SAT Solver verification

By Filip Marić

March 19, 2025

Abstract

This document contains formall correctness proofs of modern SAT solvers. Two different approaches are used — state-transition systems and shallow embedding into HOL.

Formalization based on state-transition systems follows [1, 3]. Several different SAT solver descriptions are given and their partial correctness and termination is proved. These include:

1. a solver based on classical DPLL procedure (based on backtrack-search with unit propagation),
2. a very general solver with backjumping and learning (similiar to the description given in [3]), and
3. a solver with a specific conflict analysis algorithm (similiar to the description given in [1]).

Formalization based on shallow embedding into HOL defines a SAT solver as a set or recursive HOL functions. Solver supports most state-of-the art techniques including the two-watch literal propagation scheme.

Within the SAT solver correctness proofs, a large number of lemmas about propositional logic and CNF formulae are proved. This theory is self-contained and could be used for further exploring of properties of CNF based SAT algorithms.

Contents

1 MoreList

theory *MoreList*
imports *Main HOL−Library.Multiset*
begin

Theory contains some additional lemmas and functions for the *List* datatype. Warning: some of these notions are obsolete because they already exist in *List.thy* in similiar form.

1.1 *last* and *butlast* - last element of list and elements before it

lemma *listEqualsButlastAppendLast*:
  assumes *list* ≠ []
  shows *list* = (*butlast list*) @ [*last list*]
using *assms*
by (*induct list*) *auto*

lemma *lastListInList* [*simp*]:
  assumes *list* ≠ []
  shows *last list* ∈ *set list*
using *assms*

by (*induct list*) *auto*

lemma *butlastIsSubset*:
  shows *set* (*butlast list*) $\subseteq$ *set list*
by (*induct list*) (*auto split*: *if-split-asm*)

lemma *setListIsSetButlastAndLast*:
  shows *set list* $\subseteq$ *set* (*butlast list*) $\cup$ {*last list*}
by (*induct list*) *auto*

lemma *butlastAppend*:
  shows *butlast* (*list1* @ *list2*) = (*if list2* = [] *then butlast list1 else* (*list1* @ *butlast list2*))
by (*induct list1*) *auto*

1.2 *removeAll* - element removal

lemma *removeAll-multiset*:
  assumes *distinct a x* $\in$ *set a*
  shows *mset a* = {#*x*#} + *mset* (*removeAll x a*)
using *assms*
proof (*induct a*)
  case (*Cons y a′*)
  thus *?case*
  proof (*cases x* = *y*)
    case *True*
    with ‹*distinct* (*y* # *a′*)› ‹*x* $\in$ *set* (*y* # *a′*)›
    have ¬ *x* $\in$ *set a′*
      by *auto*
    hence *removeAll x a′* = *a′*
      by (*rule removeAll-id*)
    with ‹*x* = *y*› **show** *?thesis*
      by (*simp add*: *union-commute*)
  next
    case *False*
    with ‹*x* $\in$ *set* (*y* # *a′*)›
    have *x* $\in$ *set a′*
      by *simp*
    with ‹*distinct* (*y* # *a′*)›
    have *x* $\neq$ *y distinct a′*
      by *auto*
    hence *mset a′* = {#*x*#} + *mset* (*removeAll x a′*)
      using ‹*x* $\in$ *set a′*›
      using *Cons*(*1*)
      by *simp*
    thus *?thesis*
      using ‹*x* $\neq$ *y*›
      by (*simp add*: *union-assoc*)
  qed

qed *simp*

lemma *removeAll-map*:
  assumes $\forall\ x\ y.\ x \neq y \longrightarrow f\ x \neq f\ y$
  shows *removeAll* (*f x*) (*map f a*) = *map f* (*removeAll x a*)
using *assms*
by (*induct a arbitrary*: *x*) *auto*

1.3 *uniq* - no duplicate elements.

uniq list holds iff there are no repeated elements in a list. Obsolete: same as *distinct* in *List.thy*.

primrec *uniq* :: *'a list => bool*
where
uniq [] = *True* |
uniq (*h#t*) = (*h* ∉ *set t* ∧ *uniq t*)

lemma *uniqDistinct*:
uniq l = *distinct l*
by (*induct l*) *auto*

lemma *uniqAppend*:
  assumes *uniq* (*l1* @ *l2*)
  shows *uniq l1 uniq l2*
using *assms*
by (*induct l1*) *auto*

lemma *uniqAppendIff*:
  uniq (*l1* @ *l2*) = (*uniq l1* ∧ *uniq l2* ∧ *set l1* ∩ *set l2* = {}) (**is** *?lhs* = *?rhs*)
by (*induct l1*) *auto*

lemma *uniqAppendElement*:
  assumes *uniq l*
  shows *e* ∉ *set l* = *uniq* (*l* @ [*e*])
using *assms*
by (*induct l*) (*auto split*: *if-split-asm*)

lemma *uniqImpliesNotLastMemButlast*:
  assumes *uniq l*
  shows *last l* ∉ *set* (*butlast l*)
proof (*cases l* = [])
  case *True*
  thus *?thesis*
    using *assms*
    by *simp*
next
  case *False*
  hence *l* = *butlast l* @ [*last l*]

by (*rule listEqualsButlastAppendLast*)
moreover
with ‹*uniq l*›
have *uniq* (*butlast l*)
using *uniqAppend*[*of butlast l* [*last l*]]
by *simp*
ultimately
show *?thesis*
using *assms*
using *uniqAppendElement*[*of butlast l last l*]
by *simp*
qed

lemma *uniqButlastNotUniqListImpliesLastMemButlast*:
assumes *uniq* (*butlast l*) ¬ *uniq l*
shows *last l* ∈ *set* (*butlast l*)
proof (*cases l* = [])
case *True*
thus *?thesis*
using *assms*
by *auto*
next
case *False*
hence *l* = *butlast l* @ [(*last l*)]
by (*rule listEqualsButlastAppendLast*)
thus *?thesis*
using *assms*
using *uniqAppendElement*[*of butlast l last l*]
by *auto*
qed

lemma *uniqRemdups*:
shows *uniq* (*remdups x*)
by (*induct x*) *auto*

lemma *uniqHeadTailSet*:
assumes *uniq l*
shows *set* (*tl l*) = (*set l*) − {*hd l*}
using *assms*
by (*induct l*) *auto*

lemma *uniqLengthEqCardSet*:
assumes *uniq l*
shows *length l* = *card* (*set l*)
using *assms*
by (*induct l*) *auto*

lemma *lengthGtOneTwoDistinctElements*:
assumes

```
  uniq l length l > 1 l ≠ []
shows
  ∃ a1 a2. a1 ∈ set l ∧ a2 ∈ set l ∧ a1 ≠ a2
proof−
  let ?a1 = l ! 0
  let ?a2 = l ! 1
  have ?a1 ∈ set l
    using nth-mem[of 0 l]
    using assms
    by simp
  moreover
  have ?a2 ∈ set l
    using nth-mem[of 1 l]
    using assms
    by simp
  moreover
  have ?a1 ≠ ?a2
    using nth-eq-iff-index-eq[of l 0 1]
    using assms
    by (auto simp add: uniqDistinct)
  ultimately
  show ?thesis
    by auto
qed
```

1.4 *firstPos* - first position of an element

firstPos returns the zero-based index of the first occurrence of an element int a list, or the length of the list if the element does not occur.

```
primrec firstPos :: 'a => 'a list => nat
where
firstPos a [] = 0 |
firstPos a (h # t) = (if a = h then 0 else 1 + (firstPos a t))

lemma firstPosEqualZero:
  shows (firstPos a (m # M') = 0) = (a = m)
by (induct M') auto

lemma firstPosLeLength:
  assumes a ∈ set l
  shows firstPos a l < length l
using assms
by (induct l) auto

lemma firstPosAppend:
  assumes a ∈ set l
  shows firstPos a l = firstPos a (l @ l')
using assms
```

```
by (induct l) auto

lemma firstPosAppendNonMemberFirstMemberSecond:
  assumes a ∉ set l1 and  a ∈ set l2
  shows firstPos a (l1 @ l2) = length l1 + firstPos a l2
using assms
by (induct l1) auto

lemma firstPosDomainForElements:
  shows (0 ≤ firstPos a l ∧ firstPos a l < length l) = (a ∈ set l) (is
?lhs = ?rhs)
  by (induct l) auto

lemma firstPosEqual:
  assumes a ∈ set l and b ∈ set l
  shows (firstPos a l = firstPos b l) = (a = b) (is ?lhs = ?rhs)
proof−
  {
    assume ?lhs
    hence ?rhs
      using assms
    proof (induct l)
      case (Cons m l')
      {
        assume a = m
        have b = m
        proof−
          from ‹a = m›
          have firstPos a (m # l') = 0
            by simp
          with Cons
          have firstPos b (m # l') = 0
            by simp
          with ‹b ∈ set (m # l')›
          have firstPos b (m # l') = 0
            by simp
          thus ?thesis
            using firstPosEqualZero[of b m l']
            by simp
        qed
        with ‹a = m›
        have ?case
          by simp
      }
      note ∗ = this
      moreover
      {
        assume b = m
        have a = m
```

```
  proof−
    from ‹b = m›
    have firstPos b (m # l') = 0
      by simp
    with Cons
    have firstPos a (m # l') = 0
      by simp
    with ‹a ∈ set (m # l')›
    have firstPos a (m # l') = 0
      by simp
    thus ?thesis
      using firstPosEqualZero[of a m l']
      by simp
  qed
  with ‹b = m›
  have ?case
    by simp
}
note ** = this
moreover
{
  assume Q: a ≠ m b ≠ m
  from Q ‹a ∈ set (m # l')›
  have a ∈ set l'
    by simp
  from Q ‹b ∈ set (m # l')›
  have b ∈ set l'
    by simp
  from ‹a ∈ set l'› ‹b ∈ set l'› Cons
  have firstPos a l' = firstPos b l'
    by (simp split: if-split-asm)
  with Cons
  have ?case
    by (simp split: if-split-asm)
}
note *** = this
moreover
{
  have a = m ∨ b = m ∨ a ≠ m ∧ b ≠ m
    by auto
}
ultimately
show ?thesis
proof (cases a = m)
  case True
  thus ?thesis
    by (rule *)
next
  case False
```

```
        thus ?thesis
        proof (cases b = m)
          case True
          thus ?thesis
            by (rule **)
        next
          case False
          with ‹a ≠ m› show ?thesis
            by (rule ***)
        qed
      qed
    qed simp
  } thus ?thesis
    by auto
qed

lemma firstPosLast:
  assumes l ≠ [] uniq l
  shows (firstPos x l = length l − 1) = (x = last l)
using assms
by (induct l) auto
```

1.5 *precedes* - ordering relation induced by *firstPos*

```
definition precedes :: 'a => 'a => 'a list => bool
where
precedes a b l == (a ∈ set l ∧ b ∈ set l ∧ firstPos a l <= firstPos b
l)

lemma noElementsPrecedesFirstElement:
  assumes a ≠ b
  shows ¬ precedes a b (b # list)
proof−
  {
    assume precedes a b (b # list)
    hence a ∈ set (b # list) firstPos a (b # list) <= 0
      unfolding precedes-def
      by (auto split: if-split-asm)
    hence firstPos a (b # list) = 0
      by auto
    with ‹a ≠ b›
    have False
      using firstPosEqualZero[of a b list]
      by simp
  }
  thus ?thesis
    by auto
qed
```

```
lemma lastPrecedesNoElement:
assumes uniq l
shows ¬(∃ a. a ≠ last l ∧ precedes (last l) a l)
proof−
  {
    assume ¬ ?thesis
    then obtain a
      where precedes (last l) a l a ≠ last l
      by auto
    hence a ∈ set l last l ∈ set l firstPos (last l) l ≤ firstPos a l
      unfolding precedes-def
      by auto
    hence length l − 1 ≤ firstPos a l
      using firstPosLast[of l last l]
      using ‹uniq l›
      by force
    hence firstPos a l = length l − 1
      using firstPosDomainForElements[of a l]
      using ‹a ∈ set l›
      by auto
    hence a = last l
      using firstPosLast[of l last l]
      using ‹a ∈ set l› ‹last l ∈ set l›
      using ‹uniq l›
      using firstPosEqual[of a l last l]
      by force
    with ‹a ≠ last l›
    have False
      by simp
  }
  thus ?thesis
    by auto
qed

lemma precedesAppend:
  assumes precedes a b l
  shows precedes a b (l @ l′)
proof−
  from ‹precedes a b l›
  have a ∈ set l b ∈ set l firstPos a l ≤ firstPos b l
    unfolding precedes-def
    by (auto split: if-split-asm)
  thus ?thesis
    using firstPosAppend[of a l l′]
    using firstPosAppend[of b l l′]
    unfolding precedes-def
    by simp
qed
```

```
lemma precedesMemberHeadMemberTail:
  assumes a ∈ set l1 and b ∉ set l1 and b ∈ set l2
  shows precedes a b (l1 @ l2)
proof−
  from ‹a ∈ set l1›
  have firstPos a l1 < length l1
    using firstPosLeLength [of a l1]
    by simp
  moreover
  from ‹a ∈ set l1›
  have firstPos a (l1 @ l2) = firstPos a l1
    using firstPosAppend[of a l1 l2]
    by simp
  moreover
  from ‹b ∉ set l1› ‹b ∈ set l2›
  have firstPos b (l1 @ l2) = length l1 + firstPos b l2
    by (rule firstPosAppendNonMemberFirstMemberSecond)
  moreover
  have firstPos b l2 ≥ 0
    by auto
  ultimately
  show ?thesis
    unfolding precedes-def
    using ‹a ∈ set l1› ‹b ∈ set l2›
    by simp
qed


lemma precedesReflexivity:
  assumes a ∈ set l
  shows precedes a a l
using assms
unfolding precedes-def
by simp

lemma precedesTransitivity:
  assumes
  precedes a b l and precedes b c l
  shows
  precedes a c l
using assms
unfolding precedes-def
by auto

lemma precedesAntisymmetry:
  assumes
  a ∈ set l and b ∈ set l and
  precedes a b l and precedes b a l
  shows
```

```
  a = b
proof -
  from assms
  have firstPos a l = firstPos b l
    unfolding precedes-def
    by auto
  thus ?thesis
    using firstPosEqual[of a l b]
    using assms
    by simp
qed

lemma precedesTotalOrder:
  assumes a ∈ set l and b ∈ set l
  shows a=b ∨ precedes a b l ∨ precedes b a l
using assms
unfolding precedes-def
by auto

lemma precedesMap:
  assumes precedes a b list and ∀ x y. x ≠ y ⟶ f x ≠ f y
  shows precedes (f a) (f b) (map f list)
using assms
proof (induct list)
  case (Cons l list')
    {
      assume a = l
      have ?case
      proof -
        from ‹a = l›
        have firstPos (f a) (map f (l # list')) = 0
          using firstPosEqualZero[of f a f l map f list']
          by simp
        moreover
        from ‹precedes a b (l # list')›
        have b ∈ set (l # list')
          unfolding precedes-def
          by simp
        hence f b ∈ set (map f (l # list'))
          by auto
        moreover
        hence firstPos (f b) (map f (l # list')) ≥ 0
          by auto
        ultimately
        show ?thesis
          using ‹a = l› ‹f b ∈ set (map f (l # list'))›
          unfolding precedes-def
          by simp
      qed
```

```
    }
    moreover
    {
      assume b = l
      with ‹precedes a b (l # list')›
      have a = l
        using noElementsPrecedesFirstElement[of a l list']
        by auto
      from ‹a = l› ‹b = l›
      have ?case
        unfolding precedes-def
        by simp
    }
    moreover
    {
      assume a ≠ l b ≠ l
      with ‹∀ x y. x ≠ y ⟶ f x ≠ f y›
      have f a ≠ f l f b ≠ f l
        by auto
      from ‹precedes a b (l # list')›
      have b ∈ set(l # list') a ∈ set(l # list') firstPos a (l # list') ≤
firstPos b (l # list')
        unfolding precedes-def
        by auto
      with ‹a ≠ l› ‹b ≠ l›
      have a ∈ set list' b ∈ set list' firstPos a list' ≤ firstPos b list'
        by auto
      hence precedes a b list'
        unfolding precedes-def
        by simp
      with Cons
      have precedes (f a) (f b) (map f list')
        by simp
      with ‹f a ≠ f l› ‹f b ≠ f l›
      have ?case
        unfolding precedes-def
        by auto
    }
    ultimately
    show ?case
      by auto
next
  case Nil
  thus ?case
    unfolding precedes-def
    by simp
qed

lemma precedesFilter:
```

```
  assumes precedes a b list and f a and f b
  shows precedes a b (filter f list)
using assms
proof(induct list)
  case (Cons l list')
  show ?case
  proof−
    from ‹precedes a b (l # list')›
    have a ∈ set(l # list') b ∈ set(l # list') firstPos a (l # list') ≤
firstPos b (l # list')
      unfolding precedes-def
      by auto
    from ‹f a› ‹a ∈ set(l # list')›
    have a ∈ set(filter f (l # list'))
      by auto
    moreover
    from ‹f b› ‹b ∈ set(l # list')›
    have b ∈ set(filter f (l # list'))
      by auto
    moreover
    have firstPos a (filter f (l # list')) ≤ firstPos b (filter f (l # list'))
    proof−
      {
        assume a = l
        with ‹f a›
        have firstPos a (filter f (l # list')) = 0
          by auto
        with ‹b ∈ set (filter f (l # list'))›
        have ?thesis
          by auto
      }
      moreover
      {
        assume b = l
        with ‹precedes a b (l # list')›
        have a = b
          using noElementsPrecedesFirstElement[of a b list']
          by auto
        hence ?thesis
          by (simp add: precedesReflexivity)
      }
      moreover
      {
        assume a ≠ l b ≠ l
        with ‹precedes a b (l # list')›
        have firstPos a list' ≤ firstPos b list'
          unfolding precedes-def
          by auto
        moreover
```

```
      from ‹a ≠ l› ‹a ∈ set (l # list')›
      have a ∈ set list'
        by simp
      moreover
      from ‹b ≠ l› ‹b ∈ set (l # list')›
      have b ∈ set list'
        by simp
      ultimately
      have precedes a b list'
        unfolding precedes-def
        by simp
      with ‹f a› ‹f b› Cons(1)
      have precedes a b (filter f list')
        by simp
      with ‹a ≠ l› ‹b ≠ l›
      have ?thesis
        unfolding precedes-def
        by auto
    }
    ultimately
    show ?thesis
      by blast
  qed
  ultimately
  show ?thesis
    unfolding precedes-def
    by simp
 qed
qed simp

definition
precedesOrder list == {(a, b). precedes a b list ∧ a ≠ b}

lemma transPrecedesOrder:
 trans (precedesOrder list)
proof−
 {
  fix x y z
  assume precedes x y list x ≠ y precedes y z list y ≠ z
  hence precedes x z list x ≠ z
    using precedesTransitivity[of x y list z]
    using firstPosEqual[of y list z]
    unfolding precedes-def
    by auto
 }
 thus ?thesis
  unfolding trans-def
  unfolding precedesOrder-def
  by blast
```

qed

lemma *wellFoundedPrecedesOrder*:
  shows *wf* (*precedesOrder list*)
unfolding *wf-eq-minimal*
proof−
  show ∀ *Q a*. *a*:*Q* ⟶ (∃ *aMin* ∈ *Q*. ∀ *a*′. (*a*′, *aMin*) ∈ *precedesOrder list* ⟶ *a*′ ∉ *Q*)
  proof−
    {
      fix *a* :: ′*a* **and** *Q*::′*a set*
      assume *a* ∈ *Q*
      let *?listQ* = *filter* (λ *x*. *x* ∈ *Q*) *list*
      have ∃ *aMin* ∈ *Q*. ∀ *a*′. (*a*′, *aMin*) ∈ *precedesOrder list* ⟶ *a*′ ∉ *Q*
      proof (*cases ?listQ* = [])
        case *True*
        let *?aMin* = *a*
        have ∀ *a*′. (*a*′, *?aMin*) ∈ *precedesOrder list* ⟶ *a*′ ∉ *Q*
        proof−
          {
            fix *a*′
            assume (*a*′, *?aMin*) ∈ *precedesOrder list*
            hence *a* ∈ *set list*
              unfolding *precedesOrder-def*
              unfolding *precedes-def*
              by *simp*
            with ‹*a* ∈ *Q*›
            have *a* ∈ *set ?listQ*
              by (*induct list*) *auto*
            with ‹*?listQ* = []›
            have *False*
              by *simp*
            hence *a*′ ∉ *Q*
              by *simp*
          }
          thus *?thesis*
            by *simp*
        qed
        with ‹*a* ∈ *Q*› **obtain** *aMin* **where** *aMin* ∈ *Q* ∀ *a*′. (*a*′, *aMin*) ∈ *precedesOrder list* ⟶ *a*′ ∉ *Q*
          by *auto*
        thus *?thesis*
          by *auto*
      next
        case *False*
        let *?aMin* = *hd ?listQ*
        from *False*

```
      have ?aMin ∈ Q
        by (induct list) auto
      have ∀ a'. (a', ?aMin) ∈ precedesOrder list ⟶ a' ∉ Q
      proof
        fix a'
        {
          assume (a', ?aMin) ∈ precedesOrder list
          hence a' ∈ set list precedes a' ?aMin list a' ≠ ?aMin
            unfolding precedesOrder-def
            unfolding precedes-def
            by auto
          have a' ∉ Q
          proof−
            {
              assume a' ∈ Q
              with ‹?aMin ∈ Q› ‹precedes a' ?aMin list›
              have precedes a' ?aMin ?listQ
                  using precedesFilter[of a' ?aMin list λ x. x ∈ Q]
                  by blast
              from ‹a' ≠ ?aMin›
              have ¬ precedes a' (hd ?listQ) (hd ?listQ # tl ?listQ)
                by (rule noElementsPrecedesFirstElement)
              with False ‹precedes a' ?aMin ?listQ›
              have False
                by auto
            }
            thus ?thesis
              by auto
          qed
        } thus (a', ?aMin) ∈ precedesOrder list ⟶ a' ∉ Q
          by simp
      qed
      with ‹?aMin ∈ Q›
      show ?thesis
        ..
    qed
  }
  thus ?thesis
    by simp
 qed
qed
```

1.6 *isPrefix* - prefixes of list.

Check if a list is a prefix of another list. Obsolete: similiar notion is defined in *List_prefixes.thy*.

```
definition
  isPrefix :: 'a list => 'a list => bool
  where isPrefix p t = (∃ s. p @ s = t)
```

```
lemma prefixIsSubset:
  assumes isPrefix p l
  shows set p ⊆ set l
using assms
unfolding isPrefix-def
by auto

lemma uniqListImpliesUniqPrefix:
assumes isPrefix p l and uniq l
shows uniq p
proof−
  from ‹isPrefix p l› obtain s
    where p @ s = l
    unfolding isPrefix-def
    by auto
  with ‹uniq l›
  show ?thesis
    using uniqAppend[of p s]
    by simp
qed

lemma firstPosPrefixElement:
  assumes isPrefix p l and a ∈ set p
  shows firstPos a p = firstPos a l
proof−
  from ‹isPrefix p l› obtain s
    where p @ s = l
    unfolding isPrefix-def
    by auto
  with ‹a ∈ set p›
  show ?thesis
    using firstPosAppend[of a p s]
    by simp
qed

lemma laterInPrefixRetainsPrecedes:
  assumes
  isPrefix p l and precedes a b l and b ∈ set p
  shows
  precedes a b p
proof−
  from ‹isPrefix p l› obtain s
    where p @ s = l
    unfolding isPrefix-def
    by auto
  from ‹precedes a b l›
  have a ∈ set l b ∈ set l firstPos a l ≤ firstPos b l
    unfolding precedes-def
```

by (*auto split*: *if-split-asm*)

from ‹*p* @ *s* = *l*› ‹*b* ∈ *set p*›
have *firstPos b l* = *firstPos b p*
using *firstPosAppend* [*of b p s*]
by *simp*

show *?thesis*
proof (*cases a* ∈ *set p*)
case *True*
from ‹*p* @ *s* = *l*› ‹*a* ∈ *set p*›
have *firstPos a l* = *firstPos a p*
using *firstPosAppend* [*of a p s*]
by *simp*

from ‹*firstPos a l* = *firstPos a p*› ‹*firstPos b l* = *firstPos b p*› ‹*firstPos a l* ≤ *firstPos b l*›
‹*a* ∈ *set p*› ‹*b* ∈ *set p*›
show *?thesis*
unfolding *precedes-def*
by *simp*
next
case *False*
from ‹*a* ∉ *set p*› ‹*a* ∈ *set l*› ‹*p* @ *s* = *l*›
have *a* ∈ *set s*
by *auto*
with ‹*a* ∉ *set p*› ‹*p* @ *s* = *l*›
have *firstPos a l* = *length p* + *firstPos a s*
using *firstPosAppendNonMemberFirstMemberSecond*[*of a p s*]
by *simp*
moreover
from ‹*b* ∈ *set p*›
have *firstPos b p* < *length p*
by (*rule firstPosLeLength*)
ultimately
show *?thesis*
using ‹*firstPos b l* = *firstPos b p*› ‹*firstPos a l* ≤ *firstPos b l*›
by *simp*
qed
qed

1.7 *list-diff* - the set difference operation on two lists.

primrec *list-diff* :: *'a list* ⇒ *'a list* ⇒ *'a list*
where
list-diff x [] = *x* |
list-diff x (*y#ys*) = *list-diff* (*removeAll y x*) *ys*

lemma [*simp*]:
  shows *list-diff* [] *y* = []
by (*induct y*) *auto*

lemma [*simp*]:
  shows *list-diff* (*x* # *xs*) *y* = (*if x* ∈ *set y then list-diff xs y else x* #
list-diff xs y)
proof (*induct y arbitrary*: *xs*)
  case (*Cons y ys*)
  thus *?case*
  proof (*cases x* = *y*)
    case *True*
    thus *?thesis*
      by *simp*
  next
    case *False*
    thus *?thesis*
    proof (*cases x* ∈ *set ys*)
      case *True*
      thus *?thesis*
        using *Cons*
        by *simp*
    next
      case *False*
      thus *?thesis*
        using *Cons*
        by *simp*
    qed
  qed
qed *simp*

lemma *listDiffIff*:
  shows (*x* ∈ *set a* ∧ *x* ∉ *set b*) = (*x* ∈ *set* (*list-diff a b*))
by (*induct a*) *auto*

lemma *listDiffDoubleRemoveAll*:
  assumes *x* ∈ *set a*
  shows *list-diff b a* = *list-diff b* (*x* # *a*)
using *assms*
by (*induct b*) *auto*

lemma *removeAllListDiff* [*simp*]:
  shows *removeAll x* (*list-diff a b*) = *list-diff* (*removeAll x a*) *b*
by (*induct a*) *auto*

lemma *listDiffRemoveAllNonMember*:
  assumes *x* ∉ *set a*
  shows *list-diff a b* = *list-diff a* (*removeAll x b*)
using *assms*

proof (*induct b arbitrary*: *a*)
  case (*Cons y b'*)
  from ‹$x \notin$ *set a*›
  have $x \notin$ *set* (*removeAll y a*)
    by *auto*
  thus *?case*
  proof (*cases* $x = y$)
    case *False*
    thus *?thesis*
      using *Cons*(*2*)
      using *Cons*(*1*)[*of removeAll y a*]
      using ‹$x \notin$ *set* (*removeAll y a*)›
      by *auto*
  next
    case *True*
    thus *?thesis*
      using *Cons*(*1*)[*of removeAll y a*]
      using ‹$x \notin$ *set a*›
      using ‹$x \notin$ *set* (*removeAll y a*)›
      by *auto*
  qed
qed *simp*

lemma *listDiffMap*:
  assumes $\forall\ x\ y.\ x \neq y \longrightarrow f\ x \neq f\ y$
  shows *map f* (*list-diff a b*) = *list-diff* (*map f a*) (*map f b*)
using *assms*
by (*induct b arbitrary*: *a*) (*auto simp add*: *removeAll-map*)

1.8 *remdups* - removing duplicates

lemma *remdupsRemoveAllCommute*[*simp*]:
  shows *remdups* (*removeAll a list*) = *removeAll a* (*remdups list*)
by (*induct list*) *auto*

lemma *remdupsAppend*:
  shows *remdups* (*a* @ *b*) = *remdups* (*list-diff a b*) @ *remdups b*
proof (*induct a*)
  case (*Cons x a'*)
  thus *?case*
    using *listDiffIff*[*of x a' b*]
    by *auto*
qed *simp*

lemma *remdupsAppendSet*:
  shows *set* (*remdups* (*a* @ *b*)) = *set* (*remdups a* @ *remdups* (*list-diff b a*))
proof (*induct a*)
  case *Nil*

thus *?case*
  by *auto*
next
  case (*Cons x a'*)
  thus *?case*
  proof (*cases x* ∈ *set a'*)
    case *True*
    thus *?thesis*
      using *Cons*
      using *listDiffDoubleRemoveAll*[*of x a' b*]
      by *simp*
  next
    case *False*
    thus *?thesis*
    proof (*cases x* ∈ *set b*)
      case *True*
      show *?thesis*
      proof−
        have *set* (*remdups* (*x* # *a'*) @ *remdups* (*list-diff b* (*x* # *a'*))) =

          set (*x* # *remdups a'* @ *remdups* (*list-diff b* (*x* # *a'*)))
          using ‹*x* ∉ *set a'*›
          by *auto*
          also have ... = *set* (*x* # *remdups a'* @ *remdups* (*list-diff* (*removeAll x b*) *a'*))
          by *auto*
        also have ... = *set* (*x* # *remdups a'* @ *remdups* (*removeAll x* (*list-diff b a'*)))
          by *simp*
        also have ... = *set* (*remdups a'* @ *x* # *remdups* (*removeAll x* (*list-diff b a'*)))
          by *simp*
        also have ... = *set* (*remdups a'* @ *x* # *removeAll x* (*remdups* (*list-diff b a'*)))
          by (*simp only*: *remdupsRemoveAllCommute*)
          also have ... = *set* (*remdups a'*) ∪ *set* (*x* # *removeAll x* (*remdups* (*list-diff b a'*)))
          by *simp*
         also have ... = *set* (*remdups a'*) ∪ {*x*} ∪ *set* (*removeAll x* (*remdups* (*list-diff b a'*)))
          by *auto*
       also have ... = *set* (*remdups a'*) ∪ *set* (*remdups* (*list-diff b a'*))
        proof−
          from ‹*x* ∉ *set a'*› ‹*x* ∈ *set b*›
          have *x* ∈ *set* (*list-diff b a'*)
            using *listDiffIff*[*of x b a'*]
            by *simp*
          hence *x* ∈ *set* (*remdups* (*list-diff b a'*))
            by *auto*

thus *?thesis*
by *auto*
qed
also have ... = *set* (*remdups* (a' @ *b*))
using *Cons*(*1*)
by *simp*
also have ... = *set* (*remdups* ((*x* # a') @ *b*))
using ‹$x \in$ *set b*›
by *simp*
finally show *?thesis*
by *simp*
qed
next
case *False*
thus *?thesis*
proof−
have *set* (*remdups* (*x* # a') @ *remdups* (*list-diff b* (*x* # a'))) =

set (*x* # (*remdups* a') @ *remdups* (*list-diff b* (*x* # a')))
using ‹$x \notin$ *set* a'›
by *auto*
also have ... = *set* (*x* # *remdups* a' @ *remdups* (*list-diff* (*removeAll x b*) a'))
by *auto*
also have ... = *set* (*x* # *remdups* a' @ *remdups* (*list-diff b* a'))
using ‹$x \notin$ *set b*›
by *auto*
also have ... = $\{x\} \cup$ *set* (*remdups* (a' @ *b*))
using *Cons*(*1*)
by *simp*
also have ... = *set* (*remdups* ((*x* # a') @ *b*))
by *auto*
finally show *?thesis*
by *simp*
qed
qed
qed
qed

lemma *remdupsAppendMultiSet*:
shows *mset* (*remdups* (*a* @ *b*)) = *mset* (*remdups a* @ *remdups* (*list-diff b a*))
proof (*induct a*)
case *Nil*
thus *?case*
by *auto*
next
case (*Cons x* a')
thus *?case*

```
  proof (cases x ∈ set a′)
    case True
    thus ?thesis
      using Cons
      using listDiffDoubleRemoveAll[of x a′ b]
      by simp
    next
    case False
    thus ?thesis
    proof (cases x ∈ set b)
      case True
      show ?thesis
      proof−
        have mset (remdups (x # a′) @ remdups (list-diff b (x # a′)))
=
          mset (x # remdups a′ @ remdups (list-diff b (x # a′)))
        proof−
          have remdups (x # a′) = x # remdups a′
            using ‹x ∉ set a′›
            by auto
          thus ?thesis
            by simp
        qed
        also have . . . = mset (x # remdups a′ @ remdups (list-diff
(removeAll x b) a′))
          by auto
        also have . . . = mset (x # remdups a′ @ remdups (removeAll
x (list-diff b a′)))
          by simp
        also have . . . = mset (remdups a′ @ x # remdups (removeAll
x (list-diff b a′)))
          by (simp add: union-assoc)
        also have . . . = mset (remdups a′ @ x # removeAll x (remdups
(list-diff b a′)))
          by (simp only: remdupsRemoveAllCommute)
        also have . . . = mset (remdups a′) + mset (x # removeAll x
(remdups (list-diff b a′)))
          by simp
        also have . . . = mset (remdups a′) + {#x#} + mset (removeAll
x (remdups (list-diff b a′)))
          by simp
        also have . . . = mset (remdups a′) + mset (remdups (list-diff
b a′))
        proof−
          from ‹x ∉ set a′› ‹x ∈ set b›
          have x ∈ set (list-diff b a′)
            using listDiffIff[of x b a′]
            by simp
          hence x ∈ set (remdups (list-diff b a′))
```

```
          by auto
        thus ?thesis
          using removeAll-multiset[of remdups (list-diff b a') x]
          by (simp add: union-assoc)
      qed
      also have ... = mset (remdups (a' @ b))
        using Cons(1)
        by simp
      also have ... = mset (remdups ((x # a') @ b))
        using ‹x ∈ set b›
        by simp
      finally show ?thesis
        by simp
    qed
  next
    case False
    thus ?thesis
    proof−
      have mset (remdups (x # a') @ remdups (list-diff b (x # a'))) =
        mset (x # remdups a' @ remdups (list-diff b (x # a')))
      proof−
        have remdups (x # a') = x # remdups a'
          using ‹x ∉ set a'›
          by auto
        thus ?thesis
          by simp
      qed
      also have ... = mset (x # remdups a' @ remdups (list-diff (removeAll x b) a'))
        by auto
      also have ... = mset (x # remdups a' @ remdups (list-diff b a'))
        using ‹x ∉ set b›
        using removeAll-id[of x b]
        by simp
      also have ... = {#x#} + mset (remdups (a' @ b))
        using Cons(1)
        by (simp add: union-commute)
      also have ... = mset (remdups ((x # a') @ b))
        using ‹x ∉ set a'› ‹x ∉ set b›
        by (auto simp add: union-commute)
      finally show ?thesis
        by simp
    qed
  qed
 qed
qed
```

lemma *remdupsListDiff*:
remdups (*list-diff a b*) = *list-diff* (*remdups a*) (*remdups b*)
proof(*induct a*)
  case *Nil*
  thus *?case*
    by *simp*
next
  case (*Cons x a′*)
  thus *?case*
    using *listDiffIff*[*of x a′ b*]
    by *auto*
qed

definition
multiset-le a b r == *a* = *b* ∨ (*a*, *b*) ∈ *mult r*

lemma *multisetEmptyLeI*:
  multiset-le {#} *a r*
unfolding *multiset-le-def*
using *one-step-implies-mult*[*of a* {#} *r* {#}]
by *auto*

lemma *multisetUnionLessMono2*:
shows
  trans r ⟹ (*b1*, *b2*) ∈ *mult r* ⟹ (*a* + *b1*, *a* + *b2*) ∈ *mult r*
unfolding *mult-def*
apply (*erule trancl-induct*)
apply (*blast intro*: *mult1-union transI*)
apply (*blast intro*: *mult1-union transI trancl-trans*)
done

lemma *multisetUnionLessMono1*:
shows
  trans r ⟹ (*a1*, *a2*) ∈ *mult r* ⟹ (*a1* + *b*, *a2* + *b*) ∈ *mult r*
  by (*metis multisetUnionLessMono2 union-commute*)

lemma *multisetUnionLeMono2*:
assumes
  trans r
  multiset-le b1 b2 r
shows

multiset-le (*a* + *b1*) (*a* + *b2*) *r*
using *assms*
unfolding *multiset-le-def*
using *multisetUnionLessMono2*[*of r b1 b2 a*]
by *auto*

lemma *multisetUnionLeMono1*:
assumes
trans r
multiset-le a1 a2 r
shows
multiset-le (*a1* + *b*) (*a2* + *b*) *r*
using *assms*
unfolding *multiset-le-def*
using *multisetUnionLessMono1*[*of r a1 a2 b*]
by *auto*

lemma *multisetLeTrans*:
assumes
trans r
multiset-le x y r
multiset-le y z r
shows
multiset-le x z r
using *assms*
unfolding *multiset-le-def*
unfolding *mult-def*
by (*blast intro*: *trancl-trans*)

lemma *multisetUnionLeMono*:
assumes
trans r
multiset-le a1 a2 r
multiset-le b1 b2 r
shows
multiset-le (*a1* + *b1*) (*a2* + *b2*) *r*
using *assms*
using *multisetUnionLeMono1*[*of r a1 a2 b1*]
using *multisetUnionLeMono2*[*of r b1 b2 a2*]
using *multisetLeTrans*[*of r a1* + *b1 a2* + *b1 a2* + *b2*]
by *simp*

lemma *multisetLeListDiff*:
assumes
trans r
shows
multiset-le (*mset* (*list-diff a b*)) (*mset a*) *r*
proof (*induct a*)

case *Nil*
thus *?case*
unfolding *multiset-le-def*
by *simp*
next
case (*Cons x a′*)
thus *?case*
using *assms*
using *multisetEmptyLeI*[*of* {#*x*#} *r*]
using *multisetUnionLeMono*[*of r mset* (*list-diff a′ b*) *mset a′* {#}
{#*x*#}]
using *multisetUnionLeMono1*[*of r mset* (*list-diff a′ b*) *mset a′*
{#*x*#}]
by *auto*
qed

1.9 Levi's lemma

Obsolete: these two lemmas are already proved as *append-eq-append-conv2* and *append-eq-Cons-conv*.

lemma *FullLevi*:
shows (x @ $y = z$ @ w) =
($x = z \wedge y = w \vee$
($\exists$ t. z @ $t = x \wedge t$ @ $y = w$) $\vee$
($\exists$ t. x @ $t = z \wedge t$ @ $w = y$)) (**is** *?lhs* = *?rhs*)
proof
assume *?rhs*
thus *?lhs*
by *auto*
next
assume *?lhs*
thus *?rhs*
proof (*induct x arbitrary*: *z*)
case (*Cons a x′*)
show *?case*
proof (*cases z* = [])
case *True*
with ‹(a # x') @ $y = z$ @ w›
obtain t **where** z @ $t = a$ # x' t @ $y = w$
by *auto*
thus *?thesis*
by *auto*
next
case *False*
then obtain b **and** z' **where** $z = b$ # z'
by (*auto simp add*: *neq-Nil-conv*)
with ‹(a # x') @ $y = z$ @ w›
have x' @ $y = z'$ @ w $a = b$
by *auto*

```
      with Cons(1)[of z']
      have x' = z' ∧ y = w ∨ (∃ t. z' @ t = x' ∧ t @ y = w) ∨ (∃ t.
x' @ t = z' ∧ t @ w = y)
        by simp
      with ‹a = b› ‹z = b # z'›
      show ?thesis
        by auto
    qed
  qed simp
qed

lemma SimpleLevi:
  shows (p @ s = a # list) =
             ( p = [] ∧ s = a # list ∨
              (∃ t. p = a # t ∧ t @ s = list))
by (induct p) auto
```

1.10 Single element lists

```
lemma lengthOneCharacterisation:
  shows (length l = 1) = (l = [hd l])
by (induct l) auto

lemma lengthOneImpliesOnlyElement:
  assumes length l = 1 and a : set l
  shows ∀ a'. a' : set l ⟶ a' = a
proof (cases l)
  case (Cons literal' clause')
  with assms
  show ?thesis
    by auto
qed simp


end
```

2 CNF

```
theory CNF
imports MoreList
begin
```

Theory describing formulae in Conjunctive Normal Form.

2.1 Syntax

2.1.1 Basic datatypes

```
type-synonym Variable = nat
```

datatype *Literal* = *Pos Variable* | *Neg Variable*
type-synonym *Clause* = *Literal list*
type-synonym *Formula* = *Clause list*

Notice that instead of set or multisets, lists are used in definitions of clauses and formulae. This is done because SAT solver implementation usually use list-like data structures for representing these datatypes.

2.1.2 Membership

Check if the literal is member of a clause, clause is a member of a formula or the literal is a member of a formula

consts *member* :: *'a* ⇒ *'b* ⇒ *bool* (**infixl** ‹*el*› *55*)

overloading *literalElClause* ≡ *member* :: *Literal* ⇒ *Clause* ⇒ *bool*
begin
definition [*simp*]: ((*literal*::*Literal*) *el* (*clause*::*Clause*)) == *literal* ∈ *set clause*
end

overloading *clauseElFormula* ≡ *member* :: *Clause* ⇒ *Formula* ⇒ *bool*
begin
definition [*simp*]: ((*clause*::*Clause*) *el* (*formula*::*Formula*)) == *clause* ∈ *set formula*
end

overloading *el-literal* ≡ (*el*) :: *Literal* ⇒ *Formula* ⇒ *bool*
begin

primrec *el-literal* **where**
(*literal*::*Literal*) *el* ([]::*Formula*) = *False* |
((*literal*::*Literal*) *el* ((*clause* # *formula*)::*Formula*)) = ((*literal el clause*) ∨ (*literal el formula*))

end

lemma *literalElFormulaCharacterization*:
fixes *literal* :: *Literal* **and** *formula* :: *Formula*
shows (*literal el formula*) = (∃ (*clause*::*Clause*). *clause el formula* ∧ *literal el clause*)
by (*induct formula*) *auto*

2.1.3 Variables

The variable of a given literal

primrec

var :: *Literal* ⇒ *Variable*
where
var (*Pos v*) = *v*
| *var* (*Neg v*) = *v*

Set of variables of a given clause, formula or valuation

primrec
varsClause :: (*Literal list*) ⇒ (*Variable set*)
where
varsClause [] = {}
| *varsClause* (*literal* # *list*) = {*var literal*} ∪ (*varsClause list*)

primrec
varsFormula :: *Formula* ⇒ (*Variable set*)
where
varsFormula [] = {}
| *varsFormula* (*clause* # *formula*) = (*varsClause clause*) ∪ (*varsFormula formula*)

consts *vars* :: *'a* ⇒ *Variable set*

overloading *vars-clause* ≡ *vars* :: *Clause* ⇒ *Variable set*
begin
definition [*simp*]: *vars* (*clause*::*Clause*) == *varsClause clause*
end

overloading *vars-formula* ≡ *vars* :: *Formula* ⇒ *Variable set*
begin
definition [*simp*]: *vars* (*formula*::*Formula*) == *varsFormula formula*
end

overloading *vars-set* ≡ *vars* :: *Literal set* ⇒ *Variable set*
begin
definition [*simp*]: *vars* (*s*::*Literal set*) == {*vbl*. ∃ *l*. *l* ∈ *s* ∧ *var l* = *vbl*}
end

lemma *clauseContainsItsLiteralsVariable*:
fixes *literal* :: *Literal* **and** *clause* :: *Clause*
assumes *literal el clause*
shows *var literal* ∈ *vars clause*
using *assms*
by (*induct clause*) *auto*

lemma *formulaContainsItsLiteralsVariable*:
fixes *literal* :: *Literal* **and** *formula*::*Formula*
assumes *literal el formula*
shows *var literal* ∈ *vars formula*
using *assms*

```
proof (induct formula)
  case Nil
  thus ?case
    by simp
next
  case (Cons clause formula)
  thus ?case
  proof (cases literal el clause)
    case True
    with clauseContainsItsLiteralsVariable
    have var literal ∈ vars clause
      by simp
    thus ?thesis
      by simp
  next
    case False
    with Cons
    show ?thesis
      by simp
  qed
qed

lemma formulaContainsItsClausesVariables:
  fixes clause :: Clause and formula :: Formula
  assumes clause el formula
  shows vars clause ⊆ vars formula
using assms
by (induct formula) auto

lemma varsAppendFormulae:
  fixes formula1 :: Formula and formula2 :: Formula
  shows vars (formula1 @ formula2) = vars formula1 ∪ vars formula2
by (induct formula1) auto

lemma varsAppendClauses:
  fixes clause1 :: Clause and clause2 :: Clause
  shows vars (clause1 @ clause2) = vars clause1 ∪ vars clause2
by (induct clause1) auto

lemma varsRemoveLiteral:
  fixes literal :: Literal and clause :: Clause
  shows vars (removeAll literal clause) ⊆ vars clause
by (induct clause) auto

lemma varsRemoveLiteralSuperset:
  fixes literal :: Literal and clause :: Clause
  shows vars clause − {var literal} ⊆ vars (removeAll literal clause)
by (induct clause) auto
```

```
lemma varsRemoveAllClause:
  fixes clause :: Clause and formula :: Formula
  shows vars (removeAll clause formula) ⊆ vars formula
by (induct formula) auto

lemma varsRemoveAllClauseSuperset:
  fixes clause :: Clause and formula :: Formula
  shows vars formula − vars clause ⊆ vars (removeAll clause formula)
by (induct formula) auto

lemma varInClauseVars:
  fixes variable :: Variable and clause :: Clause
  shows variable ∈ vars clause = (∃ literal. literal el clause ∧ var
literal = variable)
by (induct clause) auto

lemma varInFormulaVars:
  fixes variable :: Variable and formula :: Formula
  shows variable ∈ vars formula = (∃ literal. literal el formula ∧ var
literal = variable) (is ?lhs formula = ?rhs formula)
proof (induct formula)
  case Nil
  show ?case
    by simp
next
  case (Cons clause formula)
  show ?case
  proof
    assume P: ?lhs (clause # formula)
    thus ?rhs (clause # formula)
    proof (cases variable ∈ vars clause)
      case True
      with varInClauseVars
      have ∃ literal. literal el clause ∧ var literal = variable
        by simp
      thus ?thesis
        by auto
    next
      case False
      with P
      have variable ∈ vars formula
        by simp
      with Cons
      show ?thesis
        by auto
    qed
  next
    assume ?rhs (clause # formula)
    then obtain l
```

```
    where lEl: l el clause # formula and varL:var l = variable
      by auto
  from lEl formulaContainsItsLiteralsVariable [of l clause # formula]

    have var l ∈ vars (clause # formula)
      by auto
    with varL
    show ?lhs (clause # formula)
      by simp
  qed
qed

lemma varsSubsetFormula:
  fixes F :: Formula and F' :: Formula
  assumes ∀ c::Clause. c el F ⟶ c el F'
  shows vars F ⊆ vars F'
using assms
proof (induct F)
  case Nil
  thus ?case
    by simp
next
  case (Cons c' F'')
  thus ?case
    using formulaContainsItsClausesVariables[of c' F'']
    by simp
qed

lemma varsClauseVarsSet:
fixes
  clause :: Clause
shows
  vars clause = vars (set clause)
by (induct clause) auto
```

2.1.4 Opposite literals

```
primrec
opposite :: Literal ⇒ Literal
where
  opposite (Pos v) = (Neg v)
| opposite (Neg v) = (Pos v)

lemma oppositeIdempotency [simp]:
  fixes literal::Literal
  shows opposite (opposite literal) = literal
by (induct literal) auto

lemma oppositeSymmetry [simp]:
```

```
  fixes literal1::Literal and literal2::Literal
  shows (opposite literal1 = literal2) = (opposite literal2 = literal1)
by auto

lemma oppositeUniqueness [simp]:
  fixes literal1::Literal and literal2::Literal
  shows (opposite literal1 = opposite literal2) = (literal1 = literal2)
proof
  assume opposite literal1 = opposite literal2
  hence opposite (opposite literal1) = opposite (opposite literal2)
    by simp
  thus literal1 = literal2
    by simp
qed simp

lemma oppositeIsDifferentFromLiteral [simp]:
  fixes literal::Literal
  shows opposite literal ≠ literal
by (induct literal) auto

lemma oppositeLiteralsHaveSameVariable [simp]:
  fixes literal::Literal
  shows var (opposite literal) = var literal
by (induct literal) auto

lemma literalsWithSameVariableAreEqualOrOpposite:
  fixes literal1::Literal and literal2::Literal
  shows (var literal1 = var literal2) = (literal1 = literal2 ∨ opposite
literal1 = literal2) (is ?lhs = ?rhs)
proof
  assume ?lhs
  show ?rhs
  proof (cases literal1)
    case Pos
    note Pos1 = this
    show ?thesis
    proof (cases literal2)
      case Pos
      with ‹?lhs› Pos1 show ?thesis
        by simp
    next
      case Neg
      with ‹?lhs› Pos1 show ?thesis
        by simp
    qed
  next
    case Neg
    note Neg1 = this
    show ?thesis
```

```
    proof (cases literal2)
      case Pos
      with ‹?lhs› Neg1 show ?thesis
        by simp
    next
      case Neg
      with ‹?lhs› Neg1 show ?thesis
        by simp
    qed
  qed
next
  assume ?rhs
  thus ?lhs
    by auto
qed
```

The list of literals obtained by negating all literals of a literal list (clause, valuation). Notice that this is not a negation of a clause, because the negation of a clause is a conjunction and not a disjunction.

```
definition
oppositeLiteralList :: Literal list ⇒ Literal list
where
oppositeLiteralList clause == map opposite clause

lemma literalElListIffOppositeLiteralElOppositeLiteralList:
  fixes literal :: Literal and literalList :: Literal list
  shows literal el literalList = (opposite literal) el (oppositeLiteralList
literalList)
unfolding oppositeLiteralList-def
proof (induct literalList)
  case Nil
  thus ?case
    by simp
next
  case (Cons l literalLlist')
  show ?case
  proof (cases l = literal)
    case True
    thus ?thesis
      by simp
  next
    case False
    thus ?thesis
      by auto
  qed
qed

lemma oppositeLiteralListIdempotency [simp]:
```

fixes *literalList* :: *Literal list*
shows *oppositeLiteralList* (*oppositeLiteralList literalList*) = *literalList*
unfolding *oppositeLiteralList-def*
by (*induct literalList*) *auto*

lemma *oppositeLiteralListRemove*:
fixes *literal* :: *Literal* **and** *literalList* :: *Literal list*
shows *oppositeLiteralList* (*removeAll literal literalList*) = *removeAll* (*opposite literal*) (*oppositeLiteralList literalList*)
unfolding *oppositeLiteralList-def*
by (*induct literalList*) *auto*

lemma *oppositeLiteralListNonempty*:
fixes *literalList* :: *Literal list*
shows (*literalList* ≠ []) = ((*oppositeLiteralList literalList*) ≠ [])
unfolding *oppositeLiteralList-def*
by (*induct literalList*) *auto*

lemma *varsOppositeLiteralList*:
shows *vars* (*oppositeLiteralList clause*) = *vars clause*
unfolding *oppositeLiteralList-def*
by (*induct clause*) *auto*

2.1.5 Tautological clauses

Check if the clause contains both a literal and its opposite

primrec
clauseTautology :: *Clause* ⇒ *bool*
where
clauseTautology [] = *False*
| *clauseTautology* (*literal* # *clause*) = (*opposite literal el clause* ∨ *clauseTautology clause*)

lemma *clauseTautologyCharacterization*:
fixes *clause* :: *Clause*
shows *clauseTautology clause* = (∃ *literal*. *literal el clause* ∧ (*opposite literal*) *el clause*)
by (*induct clause*) *auto*

2.2 Semantics

2.2.1 Valuations

type-synonym *Valuation* = *Literal list*

lemma *valuationContainsItsLiteralsVariable*:
fixes *literal* :: *Literal* **and** *valuation* :: *Valuation*
assumes *literal el valuation*

shows *var literal* ∈ *vars valuation*
using *assms*
by (*induct valuation*) *auto*

lemma *varsSubsetValuation*:
fixes *valuation1* :: *Valuation* **and** *valuation2* :: *Valuation*
assumes *set valuation1* ⊆ *set valuation2*
shows *vars valuation1* ⊆ *vars valuation2*
using *assms*
proof (*induct valuation1*)
case *Nil*
show *?case*
by *simp*
next
case (*Cons literal valuation*)
note *caseCons* = *this*
hence *literal el valuation2*
by *auto*
with *valuationContainsItsLiteralsVariable* [*of literal valuation2*]
have *var literal* ∈ *vars valuation2* **.**
with *caseCons*
show *?case*
by *simp*
qed

lemma *varsAppendValuation*:
fixes *valuation1* :: *Valuation* **and** *valuation2* :: *Valuation*
shows *vars* (*valuation1* @ *valuation2*) = *vars valuation1* ∪ *vars valuation2*
by (*induct valuation1*) *auto*
lemma *varsPrefixValuation*:
fixes *valuation1* :: *Valuation* **and** *valuation2* :: *Valuation*
assumes *isPrefix valuation1 valuation2*
shows *vars valuation1* ⊆ *vars valuation2*
proof−
from *assms*
have *set valuation1* ⊆ *set valuation2*
by (*auto simp add*:*isPrefix-def*)
thus *?thesis*
by (*rule varsSubsetValuation*)
qed

2.2.2 True/False literals

Check if the literal is contained in the given valuation

definition *literalTrue* :: *Literal* ⇒ *Valuation* ⇒ *bool*
where
literalTrue-def [*simp*]: *literalTrue literal valuation* == *literal el valuation*

Check if the opposite literal is contained in the given valuation

definition *literalFalse* :: *Literal* ⇒ *Valuation* ⇒ *bool*
where
literalFalse-def [*simp*]: *literalFalse literal valuation* == *opposite literal el valuation*

lemma *variableDefinedImpliesLiteralDefined*:
  fixes *literal* :: *Literal* **and** *valuation* :: *Valuation*
  shows *var literal* ∈ *vars valuation* = (*literalTrue literal valuation* ∨ *literalFalse literal valuation*)
    (**is** (*?lhs valuation*) = (*?rhs valuation*))
proof
  assume *?rhs valuation*
  thus *?lhs valuation*
  proof
    assume *literalTrue literal valuation*
    hence *literal el valuation*
      by *simp*
    thus *?thesis*
      using *valuationContainsItsLiteralsVariable*[*of literal valuation*]
      by *simp*
  next
    assume *literalFalse literal valuation*
    hence *opposite literal el valuation*
      by *simp*
    thus *?thesis*
      using *valuationContainsItsLiteralsVariable*[*of opposite literal valuation*]
      by *simp*
  qed
next
  assume *?lhs valuation*
  thus *?rhs valuation*
  proof (*induct valuation*)
    case *Nil*
    thus *?case*
      by *simp*
  next
    case (*Cons literal′ valuation′*)
    note *ih=this*
    show *?case*
    proof (*cases var literal* ∈ *vars valuation′*)
      case *True*
      with *ih*
      show *?rhs* (*literal′* # *valuation′*)
        by *auto*
    next
      case *False*

with *ih*
have *var literal′* = *var literal*
by *simp*
hence *literal′* = *literal* ∨ *opposite literal′* = *literal*
by (*simp add*:*literalsWithSameVariableAreEqualOrOpposite*)
thus *?rhs* (*literal′* # *valuation′*)
by *auto*
qed
qed
qed

2.2.3 True/False clauses

Check if there is a literal from the clause which is true in the given valuation

primrec
clauseTrue :: *Clause* ⇒ *Valuation* ⇒ *bool*
where
clauseTrue [] *valuation* = *False*
| *clauseTrue* (*literal* # *clause*) *valuation* = (*literalTrue literal valuation* ∨ *clauseTrue clause valuation*)

Check if all the literals from the clause are false in the given valuation

primrec
clauseFalse :: *Clause* ⇒ *Valuation* ⇒ *bool*
where
clauseFalse [] *valuation* = *True*
| *clauseFalse* (*literal* # *clause*) *valuation* = (*literalFalse literal valuation* ∧ *clauseFalse clause valuation*)

lemma *clauseTrueIffContainsTrueLiteral*:
fixes *clause* :: *Clause* **and** *valuation* :: *Valuation*
shows *clauseTrue clause valuation* = (∃ *literal*. *literal el clause* ∧ *literalTrue literal valuation*)
by (*induct clause*) *auto*

lemma *clauseFalseIffAllLiteralsAreFalse*:
fixes *clause* :: *Clause* **and** *valuation* :: *Valuation*
shows *clauseFalse clause valuation* = (∀ *literal*. *literal el clause* ⟶ *literalFalse literal valuation*)
by (*induct clause*) *auto*

lemma *clauseFalseRemove*:
assumes *clauseFalse clause valuation*
shows *clauseFalse* (*removeAll literal clause*) *valuation*
proof−

{
fix *l*::*Literal*
assume *l el removeAll literal clause*
hence *l el clause*
by *simp*
with ‹*clauseFalse clause valuation*›
have *literalFalse l valuation*
by (*simp add*:*clauseFalseIffAllLiteralsAreFalse*)
}
thus *?thesis*
by (*simp add*:*clauseFalseIffAllLiteralsAreFalse*)
qed

lemma *clauseFalseAppendValuation*:
fixes *clause* :: *Clause* **and** *valuation* :: *Valuation* **and** *valuation'* :: *Valuation*
assumes *clauseFalse clause valuation*
shows *clauseFalse clause* (*valuation* @ *valuation'*)
using *assms*
by (*induct clause*) *auto*

lemma *clauseTrueAppendValuation*:
fixes *clause* :: *Clause* **and** *valuation* :: *Valuation* **and** *valuation'* :: *Valuation*
assumes *clauseTrue clause valuation*
shows *clauseTrue clause* (*valuation* @ *valuation'*)
using *assms*
by (*induct clause*) *auto*

lemma *emptyClauseIsFalse*:
fixes *valuation* :: *Valuation*
shows *clauseFalse* [] *valuation*
by *auto*

lemma *emptyValuationFalsifiesOnlyEmptyClause*:
fixes *clause* :: *Clause*
assumes *clause* ≠ []
shows ¬ *clauseFalse clause* []
using *assms*
by (*induct clause*) *auto*

lemma *valuationContainsItsFalseClausesVariables*:
fixes *clause*::*Clause* **and** *valuation*::*Valuation*
assumes *clauseFalse clause valuation*
shows *vars clause* ⊆ *vars valuation*
proof
fix *v*::*Variable*
assume *v* ∈ *vars clause*

hence ∃ *l*. *var l* = *v* ∧ *l el clause*
by (*induct clause*) *auto*
then obtain *l*
where *var l* = *v l el clause*
by *auto*
from ‹*l el clause*› ‹*clauseFalse clause valuation*›
have *literalFalse l valuation*
by (*simp add*: *clauseFalseIffAllLiteralsAreFalse*)
with ‹*var l* = *v*›
show *v* ∈ *vars valuation*
using *valuationContainsItsLiteralsVariable*[*of opposite l*]
by *simp*
qed

2.2.4 True/False formulae

Check if all the clauses from the formula are false in the given valuation

primrec
formulaTrue :: *Formula* ⇒ *Valuation* ⇒ *bool*
where
formulaTrue [] *valuation* = *True*
| *formulaTrue* (*clause* # *formula*) *valuation* = (*clauseTrue clause valuation* ∧ *formulaTrue formula valuation*)

Check if there is a clause from the formula which is false in the given valuation

primrec
formulaFalse :: *Formula* ⇒ *Valuation* ⇒ *bool*
where
formulaFalse [] *valuation* = *False*
| *formulaFalse* (*clause* # *formula*) *valuation* = (*clauseFalse clause valuation* ∨ *formulaFalse formula valuation*)

lemma *formulaTrueIffAllClausesAreTrue*:
fixes *formula* :: *Formula* **and** *valuation* :: *Valuation*
shows *formulaTrue formula valuation* = (∀ *clause*. *clause el formula* ⟶ *clauseTrue clause valuation*)
by (*induct formula*) *auto*

lemma *formulaFalseIffContainsFalseClause*:
fixes *formula* :: *Formula* **and** *valuation* :: *Valuation*
shows *formulaFalse formula valuation* = (∃ *clause*. *clause el formula* ∧ *clauseFalse clause valuation*)
by (*induct formula*) *auto*

lemma *formulaTrueAssociativity*:

fixes *f1* :: *Formula* **and** *f2* :: *Formula* **and** *f3* :: *Formula* **and** *valuation* :: *Valuation*
shows *formulaTrue* ((*f1* @ *f2*) @ *f3*) *valuation* = *formulaTrue* (*f1* @ (*f2* @ *f3*)) *valuation*
by (*auto simp add*:*formulaTrueIffAllClausesAreTrue*)

lemma *formulaTrueCommutativity*:
fixes *f1* :: *Formula* **and** *f2* :: *Formula* **and** *valuation* :: *Valuation*
shows *formulaTrue* (*f1* @ *f2*) *valuation* = *formulaTrue* (*f2* @ *f1*) *valuation*
by (*auto simp add*:*formulaTrueIffAllClausesAreTrue*)

lemma *formulaTrueSubset*:
fixes *formula* :: *Formula* **and** *formula'* :: *Formula* **and** *valuation* :: *Valuation*
assumes
formulaTrue: *formulaTrue formula valuation* **and**
subset: ∀ (*clause*::*Clause*). *clause el formula'* ⟶ *clause el formula*
shows *formulaTrue formula' valuation*
proof −
{
fix *clause* :: *Clause*
assume *clause el formula'*
with *formulaTrue subset*
have *clauseTrue clause valuation*
by (*simp add*:*formulaTrueIffAllClausesAreTrue*)
}
thus *?thesis*
by (*simp add*:*formulaTrueIffAllClausesAreTrue*)
qed

lemma *formulaTrueAppend*:
fixes *formula1* :: *Formula* **and** *formula2* :: *Formula* **and** *valuation* :: *Valuation*
shows *formulaTrue* (*formula1* @ *formula2*) *valuation* = (*formulaTrue formula1 valuation* ∧ *formulaTrue formula2 valuation*)
by (*induct formula1*) *auto*

lemma *formulaTrueRemoveAll*:
fixes *formula* :: *Formula* **and** *clause* :: *Clause* **and** *valuation* :: *Valuation*
assumes *formulaTrue formula valuation*
shows *formulaTrue* (*removeAll clause formula*) *valuation*
using *assms*
by (*induct formula*) *auto*

lemma *formulaFalseAppend*:
fixes *formula* :: *Formula* **and** *formula'* :: *Formula* **and** *valuation* :: *Valuation*

assumes *formulaFalse formula valuation*
shows *formulaFalse* (*formula* @ *formula'*) *valuation*
using *assms*
by (*induct formula*) *auto*

lemma *formulaTrueAppendValuation*:
fixes *formula* :: *Formula* **and** *valuation* :: *Valuation* **and** *valuation'* :: *Valuation*
assumes *formulaTrue formula valuation*
shows *formulaTrue formula* (*valuation* @ *valuation'*)
using *assms*
by (*induct formula*) (*auto simp add*:*clauseTrueAppendValuation*)

lemma *formulaFalseAppendValuation*:
fixes *formula* :: *Formula* **and** *valuation* :: *Valuation* **and** *valuation'* :: *Valuation*
assumes *formulaFalse formula valuation*
shows *formulaFalse formula* (*valuation* @ *valuation'*)
using *assms*
by (*induct formula*) (*auto simp add*:*clauseFalseAppendValuation*)

lemma *trueFormulaWithSingleLiteralClause*:
fixes *formula* :: *Formula* **and** *literal* :: *Literal* **and** *valuation* :: *Valuation*
assumes *formulaTrue* (*removeAll* [*literal*] *formula*) (*valuation* @ [*literal*])
shows *formulaTrue formula* (*valuation* @ [*literal*])
proof −
{
fix *clause* :: *Clause*
assume *clause el formula*
with *assms*
have *clauseTrue clause* (*valuation* @ [*literal*])
proof (*cases clause* = [*literal*])
case *True*
thus *?thesis*
by *simp*
next
case *False*
with ‹*clause el formula*›
have *clause el* (*removeAll* [*literal*] *formula*)
by *simp*
with ‹*formulaTrue* (*removeAll* [*literal*] *formula*) (*valuation* @ [*literal*])›
show *?thesis*
by (*simp add*: *formulaTrueIffAllClausesAreTrue*)
qed
}
thus *?thesis*

by (*simp add*: *formulaTrueIffAllClausesAreTrue*)
qed

2.2.5 Valuation viewed as a formula

Converts a valuation (the list of literals) into formula (list of single member lists of literals)

primrec
val2form :: *Valuation* ⇒ *Formula*
where
val2form [] = []
| *val2form* (*literal* # *valuation*) = [*literal*] # *val2form valuation*

lemma *val2FormEl*:
fixes *literal* :: *Literal* **and** *valuation* :: *Valuation*
shows *literal el valuation* = [*literal*] *el val2form valuation*
by (*induct valuation*) *auto*

lemma *val2FormAreSingleLiteralClauses*:
fixes *clause* :: *Clause* **and** *valuation* :: *Valuation*
shows *clause el val2form valuation* ⟶ (∃ *literal*. *clause* = [*literal*] ∧ *literal el valuation*)
by (*induct valuation*) *auto*

lemma *val2formOfSingleLiteralValuation*:
assumes *length v* = *1*
shows *val2form v* = [[*hd v*]]
using *assms*
by (*induct v*) *auto*

lemma *val2FormRemoveAll*:
fixes *literal* :: *Literal* **and** *valuation* :: *Valuation*
shows *removeAll* [*literal*] (*val2form valuation*) = *val2form* (*removeAll literal valuation*)
by (*induct valuation*) *auto*

lemma *val2formAppend*:
fixes *valuation1* :: *Valuation* **and** *valuation2* :: *Valuation*
shows *val2form* (*valuation1* @ *valuation2*) = (*val2form valuation1* @ *val2form valuation2*)
by (*induct valuation1*) *auto*

lemma *val2formFormulaTrue*:
fixes *valuation1* :: *Valuation* **and** *valuation2* :: *Valuation*
shows *formulaTrue* (*val2form valuation1*) *valuation2* = (∀ (*literal* :: *Literal*). *literal el valuation1* ⟶ *literal el valuation2*)
by (*induct valuation1*) *auto*

2.2.6 Consistency of valuations

Valuation is inconsistent if it contains both a literal and its opposite.

primrec
inconsistent :: *Valuation* ⇒ *bool*
where
inconsistent [] = *False*
| *inconsistent* (*literal* # *valuation*) = (*opposite literal el valuation* ∨ *inconsistent valuation*)
definition [*simp*]: *consistent valuation* == ¬ *inconsistent valuation*

lemma *inconsistentCharacterization*:
fixes *valuation* :: *Valuation*
shows *inconsistent valuation* = (∃ *literal*. *literalTrue literal valuation* ∧ *literalFalse literal valuation*)
by (*induct valuation*) *auto*

lemma *clauseTrueAndClauseFalseImpliesInconsistent*:
fixes *clause* :: *Clause* **and** *valuation* :: *Valuation*
assumes *clauseTrue clause valuation* **and** *clauseFalse clause valuation*
shows *inconsistent valuation*
proof −
from ‹*clauseTrue clause valuation*› **obtain** *literal* :: *Literal*
where *literal el clause* **and** *literalTrue literal valuation*
by (*auto simp add*: *clauseTrueIffContainsTrueLiteral*)
with ‹*clauseFalse clause valuation*›
have *literalFalse literal valuation*
by (*auto simp add*: *clauseFalseIffAllLiteralsAreFalse*)
from ‹*literalTrue literal valuation*› ‹*literalFalse literal valuation*›
show *?thesis*
by (*auto simp add*: *inconsistentCharacterization*)
qed

lemma *formulaTrueAndFormulaFalseImpliesInconsistent*:
fixes *formula* :: *Formula* **and** *valuation* :: *Valuation*
assumes *formulaTrue formula valuation* **and** *formulaFalse formula valuation*
shows *inconsistent valuation*
proof −
from ‹*formulaFalse formula valuation*› **obtain** *clause* :: *Clause*
where *clause el formula* **and** *clauseFalse clause valuation*
by (*auto simp add*: *formulaFalseIffContainsFalseClause*)
with ‹*formulaTrue formula valuation*›
have *clauseTrue clause valuation*
by (*auto simp add*: *formulaTrueIffAllClausesAreTrue*)
from ‹*clauseTrue clause valuation*› ‹*clauseFalse clause valuation*›
show *?thesis*

by (*auto simp add*: *clauseTrueAndClauseFalseImpliesInconsistent*)
qed

lemma *inconsistentAppend*:
fixes *valuation1* :: *Valuation* **and** *valuation2* :: *Valuation*
assumes *inconsistent* (*valuation1* @ *valuation2*)
shows *inconsistent valuation1* ∨ *inconsistent valuation2* ∨ (∃ *literal*. *literalTrue literal valuation1* ∧ *literalFalse literal valuation2*)
using *assms*
proof (*cases inconsistent valuation1*)
case *True*
thus *?thesis*
by *simp*
next
case *False*
thus *?thesis*
proof (*cases inconsistent valuation2*)
case *True*
thus *?thesis*
by *simp*
next
case *False*
from ‹*inconsistent* (*valuation1* @ *valuation2*)› **obtain** *literal* :: *Literal*
where *literalTrue literal* (*valuation1* @ *valuation2*) **and** *literalFalse literal* (*valuation1* @ *valuation2*)
by (*auto simp add*:*inconsistentCharacterization*)
hence (∃ *literal*. *literalTrue literal valuation1* ∧ *literalFalse literal valuation2*)
proof (*cases literalTrue literal valuation1*)
case *True*
with ‹¬ *inconsistent valuation1*›
have ¬ *literalFalse literal valuation1*
by (*auto simp add*:*inconsistentCharacterization*)
with ‹*literalFalse literal* (*valuation1* @ *valuation2*)›
have *literalFalse literal valuation2*
by *auto*
with *True*
show *?thesis*
by *auto*
next
case *False*
with ‹*literalTrue literal* (*valuation1* @ *valuation2*)›
have *literalTrue literal valuation2*
by *auto*
with ‹¬ *inconsistent valuation2*›
have ¬ *literalFalse literal valuation2*
by (*auto simp add*:*inconsistentCharacterization*)
with ‹*literalFalse literal* (*valuation1* @ *valuation2*)›

```
      have literalFalse literal valuation1
        by auto
      with ‹literalTrue literal valuation2›
      show ?thesis
        by auto
    qed
    thus ?thesis
      by simp
  qed
qed

lemma consistentAppendElement:
assumes consistent v and ¬ literalFalse l v
shows consistent (v @ [l])
proof−
  {
    assume ¬ ?thesis
    with ‹consistent v›
    have (opposite l) el v
      using inconsistentAppend[of v [l]]
      by auto
    with ‹¬ literalFalse l v›
    have False
      by simp
  }
  thus ?thesis
    by auto
qed

lemma inconsistentRemoveAll:
  fixes literal :: Literal and valuation :: Valuation
  assumes inconsistent (removeAll literal valuation)
  shows inconsistent valuation
using assms
proof −
   from ‹inconsistent (removeAll literal valuation)› obtain literal' ::
Literal
    where l'True: literalTrue literal' (removeAll literal valuation) and
l'False: literalFalse literal' (removeAll literal valuation)
    by (auto simp add:inconsistentCharacterization)
  from l'True
  have literalTrue literal' valuation
    by simp
  moreover
  from l'False
  have literalFalse literal' valuation
    by simp
  ultimately
  show ?thesis
```

by (*auto simp add*:*inconsistentCharacterization*)
qed

lemma *inconsistentPrefix*:
assumes *isPrefix valuation1 valuation2* **and** *inconsistent valuation1*
shows *inconsistent valuation2*
using *assms*
by (*auto simp add*:*inconsistentCharacterization isPrefix-def*)

lemma *consistentPrefix*:
assumes *isPrefix valuation1 valuation2* **and** *consistent valuation2*
shows *consistent valuation1*
using *assms*
by (*auto simp add*:*inconsistentCharacterization isPrefix-def*)

2.2.7 Totality of valuations

Checks if the valuation contains all the variables from the given set of variables

definition *total* **where**
[*simp*]: *total valuation variables* == *variables* ⊆ *vars valuation*

lemma *totalSubset*:
fixes *A* :: *Variable set* **and** *B* :: *Variable set* **and** *valuation* :: *Valuation*
assumes *A* ⊆ *B* **and** *total valuation B*
shows *total valuation A*
using *assms*
by *auto*

lemma *totalFormulaImpliesTotalClause*:
fixes *clause* :: *Clause* **and** *formula* :: *Formula* **and** *valuation* :: *Valuation*
assumes *clauseEl*: *clause el formula* **and** *totalFormula*: *total valuation* (*vars formula*)
shows *totalClause*: *total valuation* (*vars clause*)
proof −
from *clauseEl*
have *vars clause* ⊆ *vars formula*
using *formulaContainsItsClausesVariables* [*of clause formula*]
by *simp*
with *totalFormula*
show *?thesis*
by (*simp add*: *totalSubset*)
qed

lemma *totalValuationForClauseDefinesAllItsLiterals*:
fixes *clause* :: *Clause* **and** *valuation* :: *Valuation* **and** *literal* :: *Literal*
assumes

```
  totalClause: total valuation (vars clause) and
  literalEl: literal el clause
  shows trueOrFalse: literalTrue literal valuation ∨ literalFalse literal
valuation
proof −
  from literalEl
  have var literal ∈ vars clause
    using clauseContainsItsLiteralsVariable
    by auto
  with totalClause
  have var literal ∈ vars valuation
    by auto
  thus ?thesis
    using variableDefinedImpliesLiteralDefined [of literal valuation]
    by simp
qed

lemma totalValuationForClauseDefinesItsValue:
  fixes clause :: Clause and valuation :: Valuation
  assumes totalClause: total valuation (vars clause)
  shows clauseTrue clause valuation ∨ clauseFalse clause valuation
proof (cases clauseFalse clause valuation)
  case True
  thus ?thesis
    by (rule disjI2)
next
  case False
  hence ¬ (∀ l. l el clause ⟶ literalFalse l valuation)
    by (auto simp add:clauseFalseIffAllLiteralsAreFalse)
  then obtain l :: Literal
    where l el clause and ¬ literalFalse l valuation
    by auto
  with totalClause
  have literalTrue l valuation ∨ literalFalse l valuation
     using totalValuationForClauseDefinesAllItsLiterals [of valuation
clause l]
    by auto
  with ‹¬ literalFalse l valuation›
  have literalTrue l valuation
    by simp
  with ‹l el clause›
  have (clauseTrue clause valuation)
    by (auto simp add:clauseTrueIffContainsTrueLiteral)
  thus ?thesis
    by (rule disjI1)
qed

lemma totalValuationForFormulaDefinesAllItsLiterals:
  fixes formula::Formula and valuation::Valuation
```

assumes *totalFormula*: *total valuation* (*vars formula*) **and**
literalElFormula: *literal el formula*
shows *literalTrue literal valuation* ∨ *literalFalse literal valuation*
proof −
from *literalElFormula*
have *var literal* ∈ *vars formula*
by (*rule formulaContainsItsLiteralsVariable*)
with *totalFormula*
have *var literal* ∈ *vars valuation*
by *auto*
thus *?thesis* **using** *variableDefinedImpliesLiteralDefined* [*of literal valuation*]
by *simp*
qed

lemma *totalValuationForFormulaDefinesAllItsClauses*:
fixes *formula* :: *Formula* **and** *valuation* :: *Valuation* **and** *clause* :: *Clause*
assumes *totalFormula*: *total valuation* (*vars formula*) **and**
clauseElFormula: *clause el formula*
shows *clauseTrue clause valuation* ∨ *clauseFalse clause valuation*
proof −
from *clauseElFormula totalFormula*
have *total valuation* (*vars clause*)
by (*rule totalFormulaImpliesTotalClause*)
thus *?thesis*
by (*rule totalValuationForClauseDefinesItsValue*)
qed

lemma *totalValuationForFormulaDefinesItsValue*:
assumes *totalFormula*: *total valuation* (*vars formula*)
shows *formulaTrue formula valuation* ∨ *formulaFalse formula valuation*
proof (*cases formulaTrue formula valuation*)
case *True*
thus *?thesis*
by *simp*
next
case *False*
then obtain *clause* :: *Clause*
where *clauseElFormula*: *clause el formula* **and** *notClauseTrue*: ¬ *clauseTrue clause valuation*
by (*auto simp add*: *formulaTrueIffAllClausesAreTrue*)
from *clauseElFormula totalFormula*
have *total valuation* (*vars clause*)
using *totalFormulaImpliesTotalClause* [*of clause formula valuation*]
by *simp*
with *notClauseTrue*
have *clauseFalse clause valuation*

using *totalValuationForClauseDefinesItsValue* [*of valuation clause*]
by *simp*
with *clauseElFormula*
show *?thesis*
by (*auto simp add*:*formulaFalseIffContainsFalseClause*)
qed

lemma *totalRemoveAllSingleLiteralClause*:
fixes *literal* :: *Literal* **and** *valuation* :: *Valuation* **and** *formula* :: *Formula*
assumes *varLiteral*: *var literal* ∈ *vars valuation* **and** *totalRemoveAll*: *total valuation* (*vars* (*removeAll* [*literal*] *formula*))
shows *total valuation* (*vars formula*)
proof −
have *vars formula* − *vars* [*literal*] ⊆ *vars* (*removeAll* [*literal*] *formula*)
by (*rule varsRemoveAllClauseSuperset*)
with *assms*
show *?thesis*
by *auto*
qed

2.2.8 Models and satisfiability

Model of a formula is a consistent valuation under which formula/clause is true

consts *model* :: *Valuation* ⇒ *'a* ⇒ *bool*

overloading *modelFormula* ≡ *model* :: *Valuation* ⇒ *Formula* ⇒ *bool*
begin
definition [*simp*]: *model valuation* (*formula*::*Formula*) ==
consistent valuation ∧ (*formulaTrue formula valuation*)
end

overloading *modelClause* ≡ *model* :: *Valuation* ⇒ *Clause* ⇒ *bool*
begin
definition [*simp*]: *model valuation* (*clause*::*Clause*) ==
consistent valuation ∧ (*clauseTrue clause valuation*)
end

Checks if a formula has a model

definition *satisfiable* :: *Formula* ⇒ *bool*
where
satisfiable formula == ∃ *valuation*. *model valuation formula*

lemma *formulaWithEmptyClauseIsUnsatisfiable*:
fixes *formula* :: *Formula*
assumes ([]::*Clause*) *el formula*
shows ¬ *satisfiable formula*

```
using assms
by (auto simp add: satisfiable-def formulaTrueIffAllClausesAreTrue)

lemma satisfiableSubset:
  fixes formula0 :: Formula and formula :: Formula
  assumes subset: ∀ (clause::Clause). clause el formula0 ⟶ clause el formula
  shows satisfiable formula ⟶ satisfiable formula0
proof
  assume satisfiable formula
  show satisfiable formula0
  proof –
    from ‹satisfiable formula› obtain valuation :: Valuation
      where model valuation formula
      by (auto simp add: satisfiable-def)
    {
      fix clause :: Clause
      assume clause el formula0
      with subset
      have clause el formula
        by simp
      with ‹model valuation formula›
      have clauseTrue clause valuation
        by (simp add: formulaTrueIffAllClausesAreTrue)
    } hence formulaTrue formula0 valuation
      by (simp add: formulaTrueIffAllClausesAreTrue)
    with ‹model valuation formula›
    have model valuation formula0
      by simp
    thus ?thesis
      by (auto simp add: satisfiable-def)
  qed
qed

lemma satisfiableAppend:
  fixes formula1 :: Formula and formula2 :: Formula
  assumes satisfiable (formula1 @ formula2)
  shows satisfiable formula1 satisfiable formula2
using assms
unfolding satisfiable-def
by (auto simp add:formulaTrueAppend)

lemma modelExpand:
  fixes formula :: Formula and literal :: Literal and valuation :: Valuation
  assumes model valuation formula and var literal ∉ vars valuation
  shows model (valuation @ [literal]) formula
proof –
  from ‹model valuation formula›
```

have *formulaTrue formula* (*valuation* @ [*literal*])
  by (*simp add:formulaTrueAppendValuation*)
moreover
from ‹*model valuation formula*›
have *consistent valuation*
  by *simp*
with ‹*var literal* ∉ *vars valuation*›
have *consistent* (*valuation* @ [*literal*])
proof (*cases inconsistent* (*valuation* @ [*literal*]))
  case *True*
  hence *inconsistent valuation* ∨ *inconsistent* [*literal*] ∨ (∃ *l*. *literalTrue l valuation* ∧ *literalFalse l* [*literal*])
    by (*rule inconsistentAppend*)
  with ‹*consistent valuation*›
  have ∃ *l*. *literalTrue l valuation* ∧ *literalFalse l* [*literal*]
    by *auto*
  hence *literalFalse literal valuation*
    by *auto*
  hence *var* (*opposite literal*) ∈ (*vars valuation*)
    using *valuationContainsItsLiteralsVariable* [*of opposite literal valuation*]
    by *simp*
  with ‹*var literal* ∉ *vars valuation*›
  have *False*
    by *simp*
  thus *?thesis* **..**
qed *simp*
ultimately
show *?thesis*
  by *auto*
qed

2.2.9 Tautological clauses

lemma *tautologyNotFalse*:
  fixes *clause* :: *Clause* **and** *valuation* :: *Valuation*
  assumes *clauseTautology clause consistent valuation*
  shows ¬ *clauseFalse clause valuation*
using *assms*
  clauseTautologyCharacterization[*of clause*]
  clauseFalseIffAllLiteralsAreFalse[*of clause valuation*]
  inconsistentCharacterization
by *auto*

lemma *tautologyInTotalValuation*:
assumes
  clauseTautology clause
  vars clause ⊆ *vars valuation*

shows
clauseTrue clause valuation
proof−
from ‹*clauseTautology clause*›
obtain *literal*
where *literal el clause opposite literal el clause*
by (*auto simp add*: *clauseTautologyCharacterization*)
hence *var literal* ∈ *vars clause*
using *clauseContainsItsLiteralsVariable*[*of literal clause*]
using *clauseContainsItsLiteralsVariable*[*of opposite literal clause*]
by *simp*
hence *var literal* ∈ *vars valuation*
using ‹*vars clause* ⊆ *vars valuation*›
by *auto*
hence *literalTrue literal valuation* ∨ *literalFalse literal valuation*
using *varInClauseVars*[*of var literal valuation*]
using *varInClauseVars*[*of var* (*opposite literal*) *valuation*]
using *literalsWithSameVariableAreEqualOrOpposite*
by *auto*
thus *?thesis*
using ‹*literal el clause*› ‹*opposite literal el clause*›
by (*auto simp add*: *clauseTrueIffContainsTrueLiteral*)
qed

lemma *modelAppendTautology*:
assumes
model valuation F clauseTautology c
vars valuation ⊇ *vars F* ∪ *vars c*
shows
model valuation (*F* @ [*c*])
using *assms*
using *tautologyInTotalValuation*[*of c valuation*]
by (*auto simp add*: *formulaTrueAppend*)

lemma *satisfiableAppendTautology*:
assumes
satisfiable F clauseTautology c
shows
satisfiable (*F* @ [*c*])
proof−
from ‹*clauseTautology c*›
obtain *l*
where *l el c opposite l el c*
by (*auto simp add*: *clauseTautologyCharacterization*)
from ‹*satisfiable F*›
obtain *valuation*
where *consistent valuation formulaTrue F valuation*
unfolding *satisfiable-def*
by *auto*

show *?thesis*
proof (*cases var l* ∈ *vars valuation*)
case *True*
hence *literalTrue l valuation* ∨ *literalFalse l valuation*
using *varInClauseVars*[*of var l valuation*]
by (*auto simp add*: *literalsWithSameVariableAreEqualOrOpposite*)
hence *clauseTrue c valuation*
using ‹*l el c*› ‹*opposite l el c*›
by (*auto simp add*: *clauseTrueIffContainsTrueLiteral*)
thus *?thesis*
using ‹*consistent valuation*› ‹*formulaTrue F valuation*›
unfolding *satisfiable-def*
by (*auto simp add*: *formulaTrueIffAllClausesAreTrue*)
next
case *False*
let *?valuation'* = *valuation* @ [*l*]
have *model ?valuation' F*
using ‹*var l* ∉ *vars valuation*›
using ‹*formulaTrue F valuation*› ‹*consistent valuation*›
using *modelExpand*[*of valuation F l*]
by *simp*
moreover
have *formulaTrue* [*c*] *?valuation'*
using ‹*l el c*›
using *clauseTrueIffContainsTrueLiteral*[*of c ?valuation'*]
using *formulaTrueIffAllClausesAreTrue*[*of* [*c*] *?valuation'*]
by *auto*
ultimately
show *?thesis*
unfolding *satisfiable-def*
by (*auto simp add*: *formulaTrueAppend*)
qed
qed

lemma *modelAppendTautologicalFormula*:
fixes
F :: *Formula* **and** *F'* :: *Formula*
assumes
model valuation F ∀ *c*. *c el F'* ⟶ *clauseTautology c*
vars valuation ⊇ *vars F* ∪ *vars F'*
shows
model valuation (*F* @ *F'*)
using *assms*
proof (*induct F'*)
case *Nil*
thus *?case*
by *simp*
next
case (*Cons c F''*)

hence *model valuation* (F @ F'')
  by *simp*
hence *model valuation* ((F @ F'') @ [c])
  using *Cons*(*3*)
  using *Cons*(*4*)
  using *modelAppendTautology*[*of valuation F* @ F'' *c*]
  using *varsAppendFormulae*[*of F* F'']
  by *simp*
thus *?case*
  by (*simp add*: *formulaTrueAppend*)
qed

lemma *satisfiableAppendTautologicalFormula*:
assumes
  satisfiable F $\forall$ *c*. *c el* F' $\longrightarrow$ *clauseTautology c*
shows
  satisfiable (F @ F')
using *assms*
proof (*induct* F')
  case *Nil*
  thus *?case*
    by *simp*
next
  case (*Cons c* F'')
  hence *satisfiable* (F @ F'')
    by *simp*
  thus *?case*
    using *Cons*(*3*)
    using *satisfiableAppendTautology*[*of F* @ F'' *c*]
    unfolding *satisfiable-def*
    by (*simp add*: *formulaTrueIffAllClausesAreTrue*)
qed

lemma *satisfiableFilterTautologies*:
shows *satisfiable F* = *satisfiable* (*filter* (% *c*. $\neg$ *clauseTautology c*) *F*)
proof (*induct F*)
  case *Nil*
  thus *?case*
    by *simp*
next
  case (*Cons* c' F')
  let *?filt* = λ *F*. *filter* (% *c*. $\neg$ *clauseTautology c*) *F*
  let *?filt'* = λ *F*. *filter* (% *c*. *clauseTautology c*) *F*
  show *?case*
  proof
    assume *satisfiable* (c' # F')
    thus *satisfiable* (*?filt* (c' # F'))
      unfolding *satisfiable-def*

```
      by (auto simp add: formulaTrueIffAllClausesAreTrue)
  next
    assume satisfiable (?filt (c' # F'))
    thus satisfiable (c' # F')
    proof (cases clauseTautology c')
      case True
      hence ?filt (c' # F') = ?filt F'
        by auto
      hence satisfiable (?filt F')
        using ‹satisfiable (?filt (c' # F'))›
        by simp
      hence satisfiable F'
        using Cons
        by simp
      thus ?thesis
        using satisfiableAppendTautology[of F' c']
        using ‹clauseTautology c'›
        unfolding satisfiable-def
        by (auto simp add: formulaTrueIffAllClausesAreTrue)
    next
      case False
      hence ?filt (c' # F') = c' # ?filt F'
        by auto
      hence satisfiable (c' # ?filt F')
        using ‹satisfiable (?filt (c' # F'))›
        by simp
      moreover
      have ∀ c. c el ?filt' F' ⟶ clauseTautology c
        by simp
      ultimately
      have satisfiable ((c' # ?filt F') @ ?filt' F')
       using satisfiableAppendTautologicalFormula[of c' # ?filt F' ?filt'
F']
        by (simp (no-asm-use))
      thus ?thesis
        unfolding satisfiable-def
        by (auto simp add: formulaTrueIffAllClausesAreTrue)
    qed
  qed
qed

lemma modelFilterTautologies:
assumes
  model valuation (filter (% c. ¬ clauseTautology c) F)
  vars F ⊆ vars valuation
shows model valuation F
using assms
proof (induct F)
  case Nil
```

```
  thus ?case
    by simp
next
  case (Cons c' F')
  let ?filt = λ F. filter (% c. ¬ clauseTautology c) F
  let ?filt' = λ F. filter (% c. clauseTautology c) F
  show ?case
  proof (cases clauseTautology c')
    case True
    thus ?thesis
      using Cons
      using tautologyInTotalValuation[of c' valuation]
      by auto
  next
    case False
    hence ?filt (c' # F') = c' # ?filt F'
      by auto
    hence model valuation (c' # ?filt F')
      using ‹model valuation (?filt (c' # F'))›
      by simp
    moreover
    have ∀ c. c el ?filt' F' ⟶ clauseTautology c
      by simp
    moreover
    have vars ((c' # ?filt F') @ ?filt' F') ⊆ vars valuation
      using varsSubsetFormula[of ?filt F' F']
      using varsSubsetFormula[of ?filt' F' F']
      using varsAppendFormulae[of c' # ?filt F' ?filt' F']
      using Cons(3)
      using formulaContainsItsClausesVariables[of - ?filt F']
      by auto
    ultimately
    have model valuation ((c' # ?filt F') @ ?filt' F')
     using modelAppendTautologicalFormula[of valuation c' # ?filt F'
?filt' F']
      using varsAppendFormulae[of c' # ?filt F' ?filt' F']
      by (simp (no-asm-use)) (blast)
    thus ?thesis
      using formulaTrueAppend[of ?filt F' ?filt' F' valuation]
      using formulaTrueIffAllClausesAreTrue[of ?filt F' valuation]
      using formulaTrueIffAllClausesAreTrue[of ?filt' F' valuation]
      using formulaTrueIffAllClausesAreTrue[of F' valuation]
      by auto
  qed
qed
```

2.2.10 Entailment

Formula entails literal if it is true in all its models

definition *formulaEntailsLiteral* :: *Formula* ⇒ *Literal* ⇒ *bool*
where
formulaEntailsLiteral formula literal ==
  ∀ (*valuation*:: *Valuation*). *model valuation formula* ⟶ *literalTrue literal valuation*

Clause implies literal if it is true in all its models

definition *clauseEntailsLiteral* :: *Clause* ⇒ *Literal* ⇒ *bool*
where
clauseEntailsLiteral clause literal ==
  ∀ (*valuation*:: *Valuation*). *model valuation clause* ⟶ *literalTrue literal valuation*

Formula entails clause if it is true in all its models

definition *formulaEntailsClause* :: *Formula* ⇒ *Clause* ⇒ *bool*
where
formulaEntailsClause formula clause ==
  ∀ (*valuation*:: *Valuation*). *model valuation formula* ⟶ *model valuation clause*

Formula entails valuation if it entails its every literal

definition *formulaEntailsValuation* :: *Formula* ⇒ *Valuation* ⇒ *bool*
where
formulaEntailsValuation formula valuation ==
  ∀ *literal*. *literal el valuation* ⟶ *formulaEntailsLiteral formula literal*

Formula entails formula if it is true in all its models

definition *formulaEntailsFormula* :: *Formula* ⇒ *Formula* ⇒ *bool*
where
formulaEntailsFormula-def: *formulaEntailsFormula formula formula'* ==
  ∀ (*valuation*:: *Valuation*). *model valuation formula* ⟶ *model valuation formula'*

lemma *singleLiteralClausesEntailItsLiteral*:
  fixes *clause* :: *Clause* **and** *literal* :: *Literal*
  assumes *length clause* = *1* **and** *literal el clause*
  shows *clauseEntailsLiteral clause literal*
proof −
  from *assms*
  have *onlyLiteral*: ∀ *l*. *l el clause* ⟶ *l* = *literal*
    using *lengthOneImpliesOnlyElement*[*of clause literal*]
    by *simp*
  {
    fix *valuation* :: *Valuation*
    assume *clauseTrue clause valuation*
    with *onlyLiteral*

have *literalTrue literal valuation*
by (*auto simp add*:*clauseTrueIffContainsTrueLiteral*)
}
thus *?thesis*
by (*simp add*:*clauseEntailsLiteral-def*)
qed

lemma *clauseEntailsLiteralThenFormulaEntailsLiteral*:
fixes *clause* :: *Clause* **and** *formula* :: *Formula* **and** *literal* :: *Literal*
assumes *clause el formula* **and** *clauseEntailsLiteral clause literal*
shows *formulaEntailsLiteral formula literal*
proof −
{
fix *valuation* :: *Valuation*
assume *modelFormula*: *model valuation formula*

with ‹*clause el formula*›
have *clauseTrue clause valuation*
by (*simp add*:*formulaTrueIffAllClausesAreTrue*)
with *modelFormula* ‹*clauseEntailsLiteral clause literal*›
have *literalTrue literal valuation*
by (*auto simp add*: *clauseEntailsLiteral-def*)
}
thus *?thesis*
by (*simp add*:*formulaEntailsLiteral-def*)
qed

lemma *formulaEntailsLiteralAppend*:
fixes *formula* :: *Formula* **and** *formula'* :: *Formula* **and** *literal* :: *Literal*
assumes *formulaEntailsLiteral formula literal*
shows *formulaEntailsLiteral* (*formula* @ *formula'*) *literal*
proof −
{
fix *valuation* :: *Valuation*
assume *modelFF'*: *model valuation* (*formula* @ *formula'*)

hence *formulaTrue formula valuation*
by (*simp add*: *formulaTrueAppend*)
with *modelFF'* **and** ‹*formulaEntailsLiteral formula literal*›
have *literalTrue literal valuation*
by (*simp add*: *formulaEntailsLiteral-def*)
}
thus *?thesis*
by (*simp add*: *formulaEntailsLiteral-def*)
qed

lemma *formulaEntailsLiteralSubset*:
fixes *formula* :: *Formula* **and** *formula'* :: *Formula* **and** *literal* ::

Literal
assumes *formulaEntailsLiteral formula literal* **and** ∀ (*c*::*Clause*) . *c el formula* ⟶ *c el formula'*
shows *formulaEntailsLiteral formula' literal*
proof −
{
fix *valuation* :: *Valuation*
assume *modelF'*: *model valuation formula'*
with ‹∀ (*c*::*Clause*) . *c el formula* ⟶ *c el formula'*›
have *formulaTrue formula valuation*
by (*auto simp add*: *formulaTrueIffAllClausesAreTrue*)
with *modelF'* ‹*formulaEntailsLiteral formula literal*›
have *literalTrue literal valuation*
by (*simp add*: *formulaEntailsLiteral-def*)
}
thus *?thesis*
by (*simp add*:*formulaEntailsLiteral-def*)
qed

lemma *formulaEntailsLiteralRemoveAll*:
fixes *formula* :: *Formula* **and** *clause* :: *Clause* **and** *literal* :: *Literal*
assumes *formulaEntailsLiteral* (*removeAll clause formula*) *literal*
shows *formulaEntailsLiteral formula literal*
proof −
{
fix *valuation* :: *Valuation*
assume *modelF*: *model valuation formula*
hence *formulaTrue* (*removeAll clause formula*) *valuation*
by (*auto simp add*:*formulaTrueRemoveAll*)
with *modelF* ‹*formulaEntailsLiteral* (*removeAll clause formula*) *literal*›
have *literalTrue literal valuation*
by (*auto simp add*:*formulaEntailsLiteral-def*)
}
thus *?thesis*
by (*simp add*:*formulaEntailsLiteral-def*)
qed

lemma *formulaEntailsLiteralRemoveAllAppend*:
fixes *formula1* :: *Formula* **and** *formula2* :: *Formula* **and** *clause* :: *Clause* **and** *valuation* :: *Valuation*
assumes *formulaEntailsLiteral* ((*removeAll clause formula1*) @ *formula2*) *literal*
shows *formulaEntailsLiteral* (*formula1* @ *formula2*) *literal*
proof −
{
fix *valuation* :: *Valuation*
assume *modelF*: *model valuation* (*formula1* @ *formula2*)

hence *formulaTrue* ((*removeAll clause formula1*) @ *formula2*) *valuation*
by (*auto simp add*:*formulaTrueRemoveAll formulaTrueAppend*)
with *modelF* ‹*formulaEntailsLiteral* ((*removeAll clause formula1*) @ *formula2*) *literal*›
have *literalTrue literal valuation*
by (*auto simp add*:*formulaEntailsLiteral-def*)
}
thus *?thesis*
by (*simp add*:*formulaEntailsLiteral-def*)
qed

lemma *formulaEntailsItsClauses*:
fixes *clause* :: *Clause* **and** *formula* :: *Formula*
assumes *clause el formula*
shows *formulaEntailsClause formula clause*
using *assms*
by (*simp add*: *formulaEntailsClause-def formulaTrueIffAllClausesAreTrue*)

lemma *formulaEntailsClauseAppend*:
fixes *clause* :: *Clause* **and** *formula* :: *Formula* **and** *formula′* :: *Formula*
assumes *formulaEntailsClause formula clause*
shows *formulaEntailsClause* (*formula* @ *formula′*) *clause*
proof −
{
fix *valuation* :: *Valuation*
assume *model valuation* (*formula* @ *formula′*)
hence *model valuation formula*
by (*simp add*:*formulaTrueAppend*)
with ‹*formulaEntailsClause formula clause*›
have *clauseTrue clause valuation*
by (*simp add*:*formulaEntailsClause-def*)
}
thus *?thesis*
by (*simp add*: *formulaEntailsClause-def*)
qed

lemma *formulaUnsatIffImpliesEmptyClause*:
fixes *formula* :: *Formula*
shows *formulaEntailsClause formula* [] = (¬ *satisfiable formula*)
by (*auto simp add*: *formulaEntailsClause-def satisfiable-def*)

lemma *formulaTrueExtendWithEntailedClauses*:
fixes *formula* :: *Formula* **and** *formula0* :: *Formula* **and** *valuation* :: *Valuation*
assumes *formulaEntailed*: ∀ (*clause*::*Clause*). *clause el formula* ⟶ *formulaEntailsClause formula0 clause* **and** *consistent valuation*
shows *formulaTrue formula0 valuation* ⟶ *formulaTrue formula*

```
valuation
proof
  assume formulaTrue formula0 valuation
  {
    fix clause :: Clause
    assume clause el formula
    with formulaEntailed
    have formulaEntailsClause formula0 clause
      by simp
    with ‹formulaTrue formula0 valuation› ‹consistent valuation›
    have clauseTrue clause valuation
      by (simp add:formulaEntailsClause-def)
  }
  thus formulaTrue formula valuation
    by (simp add:formulaTrueIffAllClausesAreTrue)
qed


lemma formulaEntailsFormulaIffEntailsAllItsClauses:
  fixes formula :: Formula and formula' :: Formula
  shows formulaEntailsFormula formula formula' = (∀ clause::Clause.
clause el formula' ⟶ formulaEntailsClause formula clause)
     (is ?lhs = ?rhs)
proof
  assume ?lhs
  show ?rhs
  proof
    fix clause :: Clause
    show clause el formula' ⟶ formulaEntailsClause formula clause
    proof
      assume clause el formula'
      show formulaEntailsClause formula clause
      proof −
        {
          fix valuation :: Valuation
          assume model valuation formula
          with ‹?lhs›
          have model valuation formula'
            by (simp add:formulaEntailsFormula-def)
          with ‹clause el formula'›
          have clauseTrue clause valuation
            by (simp add:formulaTrueIffAllClausesAreTrue)
        }
        thus ?thesis
          by (simp add:formulaEntailsClause-def)
      qed
    qed
  qed
next
```

assume *?rhs*
thus *?lhs*
proof −
{
fix *valuation* :: *Valuation*
assume *model valuation formula*
{
fix *clause* :: *Clause*
assume *clause el formula'*
with ‹*?rhs*›
have *formulaEntailsClause formula clause*
by *auto*
with ‹*model valuation formula*›
have *clauseTrue clause valuation*
by (*simp add*:*formulaEntailsClause-def*)
}
hence (*formulaTrue formula' valuation*)
by (*simp add*:*formulaTrueIffAllClausesAreTrue*)
}
thus *?thesis*
by (*simp add*:*formulaEntailsFormula-def*)
qed
qed

lemma *formulaEntailsFormulaThatEntailsClause*:
fixes *formula1* :: *Formula* **and** *formula2* :: *Formula* **and** *clause* :: *Clause*
assumes *formulaEntailsFormula formula1 formula2* **and** *formulaEntailsClause formula2 clause*
shows *formulaEntailsClause formula1 clause*
using *assms*
by (*simp add*: *formulaEntailsClause-def formulaEntailsFormula-def*)

lemma
fixes *formula1* :: *Formula* **and** *formula2* :: *Formula* **and** *formula1'* :: *Formula* **and** *literal* :: *Literal*
assumes *formulaEntailsLiteral* (*formula1* @ *formula2*) *literal* **and** *formulaEntailsFormula formula1' formula1*
shows *formulaEntailsLiteral* (*formula1'* @ *formula2*) *literal*
proof −
{
fix *valuation* :: *Valuation*
assume *model valuation* (*formula1'* @ *formula2*)
hence *consistent valuation* **and** *formulaTrue formula1' valuation formulaTrue formula2 valuation*
by (*auto simp add*: *formulaTrueAppend*)
with ‹*formulaEntailsFormula formula1' formula1*›
have *model valuation formula1*

```
      by (simp add:formulaEntailsFormula-def)
    with ‹formulaTrue formula2 valuation›
    have model valuation (formula1 @ formula2)
      by (simp add: formulaTrueAppend)
    with ‹formulaEntailsLiteral (formula1 @ formula2) literal›
    have literalTrue literal valuation
      by (simp add:formulaEntailsLiteral-def)
  }
  thus ?thesis
    by (simp add:formulaEntailsLiteral-def)
qed


lemma formulaFalseInEntailedValuationIsUnsatisfiable:
  fixes formula :: Formula and valuation :: Valuation
  assumes formulaFalse formula valuation and
          formulaEntailsValuation formula valuation
  shows ¬ satisfiable formula
proof −
  from ‹formulaFalse formula valuation› obtain clause :: Clause
    where clause el formula and clauseFalse clause valuation
    by (auto simp add:formulaFalseIffContainsFalseClause)
  {
    fix valuation' :: Valuation
    assume modelV': model valuation' formula
    with ‹clause el formula› obtain literal :: Literal
      where literal el clause and literalTrue literal valuation'
    by (auto simp add: formulaTrueIffAllClausesAreTrue clauseTrueIf-
fContainsTrueLiteral)
    with ‹clauseFalse clause valuation›
    have literalFalse literal valuation
      by (auto simp add:clauseFalseIffAllLiteralsAreFalse)
    with ‹formulaEntailsValuation formula valuation›
    have formulaEntailsLiteral formula (opposite literal)
      unfolding formulaEntailsValuation-def
      by simp
    with modelV'
    have literalFalse literal valuation'
      by (auto simp add:formulaEntailsLiteral-def)
    from ‹literalTrue literal valuation'› ‹literalFalse literal valuation'›
modelV'
    have False
      by (simp add:inconsistentCharacterization)
  }
  thus ?thesis
    by (auto simp add:satisfiable-def)
qed

lemma formulaFalseInEntailedOrPureValuationIsUnsatisfiable:
```

fixes *formula* :: *Formula* **and** *valuation* :: *Valuation*
assumes *formulaFalse formula valuation* **and**
∀ *literal′*. *literal′ el valuation* ⟶ *formulaEntailsLiteral formula literal′* ∨ ¬ *opposite literal′ el formula*
shows ¬ *satisfiable formula*
proof −
from ‹*formulaFalse formula valuation*› **obtain** *clause* :: *Clause*
where *clause el formula* **and** *clauseFalse clause valuation*
by (*auto simp add*:*formulaFalseIffContainsFalseClause*)
{
fix *valuation′* :: *Valuation*
assume *modelV′*: *model valuation′ formula*
with ‹*clause el formula*› **obtain** *literal* :: *Literal*
where *literal el clause* **and** *literalTrue literal valuation′*
by (*auto simp add*: *formulaTrueIffAllClausesAreTrue clauseTrueIffContainsTrueLiteral*)
with ‹*clauseFalse clause valuation*›
have *literalFalse literal valuation*
by (*auto simp add*:*clauseFalseIffAllLiteralsAreFalse*)
with ‹∀ *literal′*. *literal′ el valuation* ⟶ *formulaEntailsLiteral formula literal′* ∨ ¬ *opposite literal′ el formula*›
have *formulaEntailsLiteral formula* (*opposite literal*) ∨ ¬ *literal el formula*
by *auto*
moreover
{
assume *formulaEntailsLiteral formula* (*opposite literal*)
with *modelV′*
have *literalFalse literal valuation′*
by (*auto simp add*:*formulaEntailsLiteral-def*)
from ‹*literalTrue literal valuation′*› ‹*literalFalse literal valuation′*› *modelV′*
have *False*
by (*simp add*:*inconsistentCharacterization*)
}
moreover
{
assume ¬ *literal el formula*
with ‹*clause el formula*› ‹*literal el clause*›
have *False*
by (*simp add*:*literalElFormulaCharacterization*)
}
ultimately
have *False*
by *auto*
}
thus *?thesis*
by (*auto simp add*:*satisfiable-def*)
qed

```
lemma unsatisfiableFormulaWithSingleLiteralClause:
  fixes formula :: Formula and literal :: Literal
  assumes ¬ satisfiable formula and [literal] el formula
  shows formulaEntailsLiteral (removeAll [literal] formula) (opposite
literal)
proof -
  {
    fix valuation :: Valuation
    assume model valuation (removeAll [literal] formula)
    hence literalFalse literal valuation
    proof (cases var literal ∈ vars valuation)
      case True
      {
        assume literalTrue literal valuation
        with ‹model valuation (removeAll [literal] formula)›
        have model valuation formula
          by (auto simp add:formulaTrueIffAllClausesAreTrue)
        with ‹¬ satisfiable formula›
        have False
          by (auto simp add:satisfiable-def)
      }
      with True
      show ?thesis
       using variableDefinedImpliesLiteralDefined [of literal valuation]
        by auto
    next
      case False
      with ‹model valuation (removeAll [literal] formula)›
      have model (valuation @ [literal]) (removeAll [literal] formula)
        by (rule modelExpand)
      hence
       formulaTrue (removeAll [literal] formula) (valuation @ [literal])
and consistent (valuation @ [literal])
        by auto
       from ‹formulaTrue (removeAll [literal] formula) (valuation @
[literal])›
      have formulaTrue formula (valuation @ [literal])
        by (rule trueFormulaWithSingleLiteralClause)
      with ‹consistent (valuation @ [literal])›
      have model (valuation @ [literal]) formula
        by simp
      with ‹¬ satisfiable formula›
      have False
        by (auto simp add:satisfiable-def)
      thus ?thesis ..
    qed
  }
```

```
  thus ?thesis
    by (simp add:formulaEntailsLiteral-def)
qed

lemma unsatisfiableFormulaWithSingleLiteralClauses:
  fixes F::Formula and c::Clause
   assumes ¬ satisfiable (F @ val2form (oppositeLiteralList c)) ¬
clauseTautology c
  shows formulaEntailsClause F c
proof−
  {
    fix v::Valuation
    assume model v F
    with ‹¬ satisfiable (F @ val2form (oppositeLiteralList c))›
    have ¬ formulaTrue (val2form (oppositeLiteralList c)) v
      unfolding satisfiable-def
      by (auto simp add: formulaTrueAppend)
    have clauseTrue c v
    proof (cases ∃ l. l el c ∧ (literalTrue l v))
      case True
      thus ?thesis
        using clauseTrueIffContainsTrueLiteral
        by simp
    next
      case False
      let ?v′ = v @ (oppositeLiteralList c)

      have ¬ inconsistent (oppositeLiteralList c)
      proof−
        {
          assume ¬ ?thesis
          then obtain l::Literal
        where l el (oppositeLiteralList c) opposite l el (oppositeLiteralList
c)
            using inconsistentCharacterization [of oppositeLiteralList c]
            by auto
          hence (opposite l) el c l el c
            using literalElListIffOppositeLiteralElOppositeLiteralList[of
l c]
            using literalElListIffOppositeLiteralElOppositeLiteralList[of
opposite l c]
            by auto
          hence clauseTautology c
            using clauseTautologyCharacterization[of c]
            by auto
          with ‹¬ clauseTautology c›
          have False
            by simp
        }
```

```
      thus ?thesis
        by auto
    qed
    with False ‹model v F›
    have consistent ?v'
      using inconsistentAppend[of v oppositeLiteralList c]
      unfolding consistent-def
      using literalElListIffOppositeLiteralElOppositeLiteralList
      by auto
    moreover
    from ‹model v F›
    have formulaTrue F ?v'
      using formulaTrueAppendValuation
      by simp
    moreover
    have formulaTrue (val2form (oppositeLiteralList c)) ?v'
      using val2formFormulaTrue[of oppositeLiteralList c v @ oppositeLiteralList c]
      by simp
    ultimately
    have model ?v' (F @ val2form (oppositeLiteralList c))
      by (simp add: formulaTrueAppend)
    with ‹¬ satisfiable (F @ val2form (oppositeLiteralList c))›
    have False
      unfolding satisfiable-def
      by auto
    thus ?thesis
      by simp
  qed
}
thus ?thesis
  unfolding formulaEntailsClause-def
  by simp
qed

lemma satisfiableEntailedFormula:
  fixes formula0 :: Formula and formula :: Formula
  assumes formulaEntailsFormula formula0 formula
  shows satisfiable formula0 ⟶ satisfiable formula
proof
  assume satisfiable formula0
  show satisfiable formula
  proof -
    from ‹satisfiable formula0› obtain valuation :: Valuation
      where model valuation formula0
      by (auto simp add: satisfiable-def)
    with ‹formulaEntailsFormula formula0 formula›
    have model valuation formula
      by (simp add: formulaEntailsFormula-def)
```

```
    thus ?thesis
      by (auto simp add: satisfiable-def)
  qed
qed

lemma val2formIsEntailed:
shows formulaEntailsValuation (F' @ val2form valuation @ F'') val-
uation
proof−
  {
    fix l::Literal
    assume l el valuation
    hence [l] el val2form valuation
      by (induct valuation) (auto)

    have formulaEntailsLiteral (F' @ val2form valuation @ F'') l
    proof−
      {
        fix valuation'::Valuation
       assume formulaTrue (F' @ val2form valuation @ F'') valuation'
        hence literalTrue l valuation'
          using ‹[l] el val2form valuation›
             using formulaTrueIffAllClausesAreTrue[of F' @ val2form
valuation @ F'' valuation']
          by (auto simp add: clauseTrueIffContainsTrueLiteral)
      } thus ?thesis
        unfolding formulaEntailsLiteral-def
        by simp
    qed
  }
  thus ?thesis
    unfolding formulaEntailsValuation-def
    by simp
qed
```

2.2.11 Equivalency

Formulas are equivalent if they have same models.

```
definition equivalentFormulae :: Formula ⇒ Formula ⇒ bool
where
equivalentFormulae formula1 formula2 ==
  ∀ (valuation::Valuation). model valuation formula1 = model valua-
tion formula2

lemma equivalentFormulaeIffEntailEachOther:
  fixes formula1 :: Formula and formula2 :: Formula
 shows equivalentFormulae formula1 formula2 = (formulaEntailsFormula
formula1 formula2 ∧ formulaEntailsFormula formula2 formula1)
by (auto simp add:formulaEntailsFormula-def equivalentFormulae-def)
```

lemma *equivalentFormulaeReflexivity*:
fixes *formula* :: *Formula*
shows *equivalentFormulae formula formula*
unfolding *equivalentFormulae-def*
by *auto*

lemma *equivalentFormulaeSymmetry*:
fixes *formula1* :: *Formula* **and** *formula2* :: *Formula*
shows *equivalentFormulae formula1 formula2* = *equivalentFormulae formula2 formula1*
unfolding *equivalentFormulae-def*
by *auto*

lemma *equivalentFormulaeTransitivity*:
fixes *formula1* :: *Formula* **and** *formula2* :: *Formula* **and** *formula3* :: *Formula*
assumes *equivalentFormulae formula1 formula2* **and** *equivalentFormulae formula2 formula3*
shows *equivalentFormulae formula1 formula3*
using *assms*
unfolding *equivalentFormulae-def*
by *auto*

lemma *equivalentFormulaeAppend*:
fixes *formula1* :: *Formula* **and** *formula1'* :: *Formula* **and** *formula2* :: *Formula*
assumes *equivalentFormulae formula1 formula1'*
shows *equivalentFormulae* (*formula1* @ *formula2*) (*formula1'* @ *formula2*)
using *assms*
unfolding *equivalentFormulae-def*
by (*auto simp add*: *formulaTrueAppend*)

lemma *satisfiableEquivalent*:
fixes *formula1* :: *Formula* **and** *formula2* :: *Formula*
assumes *equivalentFormulae formula1 formula2*
shows *satisfiable formula1* = *satisfiable formula2*
using *assms*
unfolding *equivalentFormulae-def*
unfolding *satisfiable-def*
by *auto*

lemma *satisfiableEquivalentAppend*:
fixes *formula1* :: *Formula* **and** *formula1'* :: *Formula* **and** *formula2* :: *Formula*
assumes *equivalentFormulae formula1 formula1'* **and** *satisfiable* (*formula1* @ *formula2*)
shows *satisfiable* (*formula1'* @ *formula2*)

using *assms*
proof −
from ‹*satisfiable* (*formula1* @ *formula2*)› **obtain** *valuation*:: *Valuation*
where *consistent valuation formulaTrue formula1 valuation formulaTrue formula2 valuation*
unfolding *satisfiable-def*
by (*auto simp add*: *formulaTrueAppend*)
from ‹*equivalentFormulae formula1 formula1′*› ‹*consistent valuation*› ‹*formulaTrue formula1 valuation*›
have *formulaTrue formula1′ valuation*
unfolding *equivalentFormulae-def*
by *auto*
show *?thesis*
using ‹*consistent valuation*› ‹*formulaTrue formula1′ valuation*› ‹*formulaTrue formula2 valuation*›
unfolding *satisfiable-def*
by (*auto simp add*: *formulaTrueAppend*)
qed

lemma *replaceEquivalentByEquivalent*:
fixes *formula* :: *Formula* **and** *formula′* :: *Formula* **and** *formula1* :: *Formula* **and** *formula2* :: *Formula*
assumes *equivalentFormulae formula formula′*
shows *equivalentFormulae* (*formula1* @ *formula* @ *formula2*) (*formula1* @ *formula′* @ *formula2*)
unfolding *equivalentFormulae-def*
proof
fix *v* :: *Valuation*
show *model v* (*formula1* @ *formula* @ *formula2*) = *model v* (*formula1* @ *formula′* @ *formula2*)
proof
assume *model v* (*formula1* @ *formula* @ *formula2*)
hence ∗: *consistent v formulaTrue formula1 v formulaTrue formula v formulaTrue formula2 v*
by (*auto simp add*: *formulaTrueAppend*)
from ‹*consistent v*› ‹*formulaTrue formula v*› ‹*equivalentFormulae formula formula′*›
have *formulaTrue formula′ v*
unfolding *equivalentFormulae-def*
by *auto*
thus *model v* (*formula1* @ *formula′* @ *formula2*)
using ∗
by (*simp add*: *formulaTrueAppend*)
next
assume *model v* (*formula1* @ *formula′* @ *formula2*)
hence ∗: *consistent v formulaTrue formula1 v formulaTrue formula′ v formulaTrue formula2 v*
by (*auto simp add*: *formulaTrueAppend*)

from ‹*consistent v*› ‹*formulaTrue formula′ v*› ‹*equivalentFormulae formula formula′*›
have *formulaTrue formula v*
unfolding *equivalentFormulae-def*
by *auto*
thus *model v* (*formula1* @ *formula* @ *formula2*)
using ∗
by (*simp add*: *formulaTrueAppend*)
qed
qed

lemma *clauseOrderIrrelevant*:
shows *equivalentFormulae* (*F1* @ *F* @ *F′* @ *F2*) (*F1* @ *F′* @ *F* @ *F2*)
unfolding *equivalentFormulae-def*
by (*auto simp add*: *formulaTrueIffAllClausesAreTrue*)

lemma *extendEquivalentFormulaWithEntailedClause*:
fixes *formula1* :: *Formula* **and** *formula2* :: *Formula* **and** *clause* :: *Clause*
assumes *equivalentFormulae formula1 formula2* **and** *formulaEntailsClause formula2 clause*
shows *equivalentFormulae formula1* (*formula2* @ [*clause*])
unfolding *equivalentFormulae-def*
proof
fix *valuation* :: *Valuation*
show *model valuation formula1* = *model valuation* (*formula2* @ [*clause*])
proof
assume *model valuation formula1*
hence *consistent valuation*
by *simp*
from ‹*model valuation formula1*› ‹*equivalentFormulae formula1 formula2*›
have *model valuation formula2*
unfolding *equivalentFormulae-def*
by *simp*
moreover
from ‹*model valuation formula2*› ‹*formulaEntailsClause formula2 clause*›
have *clauseTrue clause valuation*
unfolding *formulaEntailsClause-def*
by *simp*
ultimately show
model valuation (*formula2* @ [*clause*])
by (*simp add*: *formulaTrueAppend*)
next
assume *model valuation* (*formula2* @ [*clause*])
hence *consistent valuation*

by *simp*
from ‹*model valuation* (*formula2* @ [*clause*])›
have *model valuation formula2*
by (*simp add*:*formulaTrueAppend*)
with ‹*equivalentFormulae formula1 formula2*›
show *model valuation formula1*
unfolding *equivalentFormulae-def*
by *auto*
qed
qed

lemma *entailsLiteralRelpacePartWithEquivalent*:
assumes *equivalentFormulae F F′* **and** *formulaEntailsLiteral* (*F1* @ *F* @ *F2*) *l*
shows *formulaEntailsLiteral* (*F1* @ *F′* @ *F2*) *l*
proof−
{
fix *v*::*Valuation*
assume *model v* (*F1* @ *F′* @ *F2*)
hence *consistent v* **and** *formulaTrue F1 v* **and** *formulaTrue F′ v* **and** *formulaTrue F2 v*
by (*auto simp add*:*formulaTrueAppend*)
with ‹*equivalentFormulae F F′*›
have *formulaTrue F v*
unfolding *equivalentFormulae-def*
by *auto*
with ‹*consistent v*› ‹*formulaTrue F1 v*› ‹*formulaTrue F2 v*›
have *model v* (*F1* @ *F* @ *F2*)
by (*auto simp add*:*formulaTrueAppend*)
with ‹*formulaEntailsLiteral* (*F1* @ *F* @ *F2*) *l*›
have *literalTrue l v*
unfolding *formulaEntailsLiteral-def*
by *auto*
}
thus *?thesis*
unfolding *formulaEntailsLiteral-def*
by *auto*
qed

2.2.12 Remove false and duplicate literals of a clause

definition
removeFalseLiterals :: *Clause* ⇒ *Valuation* ⇒ *Clause*
where
removeFalseLiterals clause valuation = *filter* (λ *l*. ¬ *literalFalse l valuation*) *clause*

lemma *clauseTrueRemoveFalseLiterals*:
assumes *consistent v*

shows *clauseTrue c v* = *clauseTrue* (*removeFalseLiterals c v*) *v*
using *assms*
unfolding *removeFalseLiterals-def*
by (*auto simp add*: *clauseTrueIffContainsTrueLiteral inconsistentCharacterization*)

lemma *clauseTrueRemoveDuplicateLiterals*:
shows *clauseTrue c v* = *clauseTrue* (*remdups c*) *v*
by (*induct c*) (*auto simp add*: *clauseTrueIffContainsTrueLiteral*)

lemma *removeDuplicateLiteralsEquivalentClause*:
shows *equivalentFormulae* [*remdups clause*] [*clause*]
unfolding *equivalentFormulae-def*
by (*auto simp add*: *formulaTrueIffAllClausesAreTrue clauseTrueIffContainsTrueLiteral*)

lemma *falseLiteralsCanBeRemoved*:

fixes *F*::*Formula* **and** *F'*::*Formula* **and** *v*::*Valuation*
assumes *equivalentFormulae* (*F1* @ *val2form v* @ *F2*) *F'*
shows *equivalentFormulae* (*F1* @ *val2form v* @ [*removeFalseLiterals c v*] @ *F2*) (*F'* @ [*c*])
(**is** *equivalentFormulae ?lhs ?rhs*)
unfolding *equivalentFormulae-def*
proof
fix *v'* :: *Valuation*
show *model v' ?lhs* = *model v' ?rhs*
proof
assume *model v' ?lhs*
hence *consistent v'* **and**
formulaTrue (*F1* @ *val2form v* @ *F2*) *v'* **and**
clauseTrue (*removeFalseLiterals c v*) *v'*
by (*auto simp add*: *formulaTrueAppend formulaTrueIffAllClausesAreTrue*)

from ‹*consistent v'*› ‹*formulaTrue* (*F1* @ *val2form v* @ *F2*) *v'*› ‹*equivalentFormulae* (*F1* @ *val2form v* @ *F2*) *F'*›
have *model v' F'*
unfolding *equivalentFormulae-def*
by *auto*
moreover
from ‹*clauseTrue* (*removeFalseLiterals c v*) *v'*›
have *clauseTrue c v'*
unfolding *removeFalseLiterals-def*
by (*auto simp add*: *clauseTrueIffContainsTrueLiteral*)
ultimately
show *model v' ?rhs*
by (*simp add*: *formulaTrueAppend*)
next

assume *model v′ ?rhs*
hence *consistent v′* **and** *formulaTrue F′ v′* **and** *clauseTrue c v′*
by (*auto simp add*: *formulaTrueAppend formulaTrueIffAllClausesAreTrue*)

from ‹*consistent v′*› ‹*formulaTrue F′ v′*› ‹*equivalentFormulae* (*F1* @ *val2form v* @ *F2*) *F′*›
have *model v′* (*F1* @ *val2form v* @ *F2*)
unfolding *equivalentFormulae-def*
by *auto*
moreover
have *clauseTrue* (*removeFalseLiterals c v*) *v′*
proof−
from ‹*clauseTrue c v′*›
obtain *l* :: *Literal*
where *l el c* **and** *literalTrue l v′*
by (*auto simp add*: *clauseTrueIffContainsTrueLiteral*)
have ¬ *literalFalse l v*
proof−
{
assume ¬ *?thesis*
hence *opposite l el v*
by *simp*
with ‹*model v′* (*F1* @ *val2form v* @ *F2*)›
have *opposite l el v′*
using *val2formFormulaTrue*[*of v v′*]
by *auto* (*simp add*: *formulaTrueAppend*)
with ‹*literalTrue l v′*› ‹*consistent v′*›
have *False*
by (*simp add*: *inconsistentCharacterization*)
}
thus *?thesis*
by *auto*
qed
with ‹*l el c*›
have *l el* (*removeFalseLiterals c v*)
unfolding *removeFalseLiterals-def*
by *simp*
with ‹*literalTrue l v′*›
show *?thesis*
by (*auto simp add*: *clauseTrueIffContainsTrueLiteral*)
qed
ultimately
show *model v′ ?lhs*
by (*simp add*: *formulaTrueAppend*)
qed
qed

lemma *falseAndDuplicateLiteralsCanBeRemoved*:

assumes *equivalentFormulae* (*F1* @ *val2form v* @ *F2*) *F′*
shows *equivalentFormulae* (*F1* @ *val2form v* @ [*remdups* (*removeFalseLiterals*
c v)] @ *F2*) (*F′* @ [*c*])
 (**is** *equivalentFormulae ?lhs ?rhs*)
proof−
 from ‹*equivalentFormulae* (*F1* @ *val2form v* @ *F2*) *F′*›
 have *equivalentFormulae* (*F1* @ *val2form v* @ [*removeFalseLiterals*
c v] @ *F2*) (*F′* @ [*c*])
  using *falseLiteralsCanBeRemoved*
  by *simp*
 have *equivalentFormulae* [*remdups* (*removeFalseLiterals c v*)] [*removeFalseLiterals*
c v]
  using *removeDuplicateLiteralsEquivalentClause*
  by *simp*
 hence *equivalentFormulae* (*F1* @ *val2form v* @ [*remdups* (*removeFalseLiterals*
c v)] @ *F2*)
  (*F1* @ *val2form v* @ [*removeFalseLiterals c v*] @ *F2*)
  using *replaceEquivalentByEquivalent*
  [*of* [*remdups* (*removeFalseLiterals c v*)] [*removeFalseLiterals c v*]
F1 @ *val2form v F2*]
  by *auto*
 thus *?thesis*
  using ‹*equivalentFormulae* (*F1* @ *val2form v* @ [*removeFalseLiterals*
c v] @ *F2*) (*F′* @ [*c*])›
  using *equivalentFormulaeTransitivity*[*of*
   (*F1* @ *val2form v* @ [*remdups* (*removeFalseLiterals c v*)]
@ *F2*)
   (*F1* @ *val2form v* @ [*removeFalseLiterals c v*] @ *F2*)
   F′ @ [*c*]]
  by *simp*
qed

lemma *satisfiedClauseCanBeRemoved*:
assumes
 equivalentFormulae (*F* @ *val2form v*) *F′*
 clauseTrue c v
shows *equivalentFormulae* (*F* @ *val2form v*) (*F′* @ [*c*])
unfolding *equivalentFormulae-def*
proof
 fix *v′* :: *Valuation*
 show *model v′* (*F* @ *val2form v*) = *model v′* (*F′* @ [*c*])
 proof
  assume *model v′* (*F* @ *val2form v*)
  hence *consistent v′* **and** *formulaTrue* (*F* @ *val2form v*) *v′*
   by *auto*

   from ‹*model v′* (*F* @ *val2form v*)› ‹*equivalentFormulae* (*F* @
val2form v) *F′*›

```
    have model v' F'
      unfolding equivalentFormulae-def
      by auto
    moreover
    have clauseTrue c v'
    proof-
      from ‹clauseTrue c v›
      obtain l :: Literal
        where literalTrue l v and l el c
        by (auto simp add:clauseTrueIffContainsTrueLiteral)
      with ‹formulaTrue (F @ val2form v) v'›
      have literalTrue l v'
        using val2formFormulaTrue[of v v']
        using formulaTrueAppend[of F val2form v]
        by simp
      thus ?thesis
        using ‹l el c›
        by (auto simp add:clauseTrueIffContainsTrueLiteral)
    qed
    ultimately
    show model v' (F' @ [c])
      by (simp add: formulaTrueAppend)
  next
    assume model v' (F' @ [c])
    thus model v' (F @ val2form v)
      using ‹equivalentFormulae (F @ val2form v) F'›
      unfolding equivalentFormulae-def
      using formulaTrueAppend[of F' [c] v']
      by auto
  qed
qed

lemma formulaEntailsClauseRemoveEntailedLiteralOpposites:
assumes
  formulaEntailsClause F clause
  formulaEntailsValuation F valuation
shows
  formulaEntailsClause F (list-diff clause (oppositeLiteralList valuation))
proof-
  {
    fix valuation'
    assume model valuation' F
    hence consistent valuation' formulaTrue F valuation'
      by (auto simp add: formulaTrueAppend)

    have model valuation' clause
      using ‹consistent valuation'›
      using ‹formulaTrue F valuation'›
```

```
      using ‹formulaEntailsClause F clause›
      unfolding formulaEntailsClause-def
      by simp

    then obtain l::Literal
      where l el clause literalTrue l valuation'
      by (auto simp add: clauseTrueIffContainsTrueLiteral)
    moreover
    hence ¬ l el (oppositeLiteralList valuation)
    proof−
      {
        assume l el (oppositeLiteralList valuation)
        hence (opposite l) el valuation
          using literalElListIffOppositeLiteralElOppositeLiteralList[of l
oppositeLiteralList valuation]
          by simp
        hence formulaEntailsLiteral F (opposite l)
          using ‹formulaEntailsValuation F valuation›
          unfolding formulaEntailsValuation-def
          by simp
        hence literalFalse l valuation'
          using ‹consistent valuation'›
          using ‹formulaTrue F valuation'›
          unfolding formulaEntailsLiteral-def
          by simp
        with ‹literalTrue l valuation'›
          ‹consistent valuation'›
        have False
          by (simp add: inconsistentCharacterization)
      } thus ?thesis
        by auto
    qed
    ultimately
    have model valuation' (list-diff clause (oppositeLiteralList valua-
tion))
      using ‹consistent valuation'›
      using listDiffIff[of l clause oppositeLiteralList valuation]
      by (auto simp add: clauseTrueIffContainsTrueLiteral)
  } thus ?thesis
    unfolding formulaEntailsClause-def
    by simp
qed
```

2.2.13 Resolution

definition
resolve clause1 clause2 literal == removeAll literal clause1 @ removeAll (opposite literal) clause2

lemma *resolventIsEntailed*:
fixes *clause1* :: *Clause* **and** *clause2* :: *Clause* **and** *literal* :: *Literal*
shows *formulaEntailsClause* [*clause1*, *clause2*] (*resolve clause1 clause2 literal*)
proof −
{
fix *valuation* :: *Valuation*
assume *model valuation* [*clause1*, *clause2*]
from ‹*model valuation* [*clause1*, *clause2*]› **obtain** *l1* :: *Literal*
where *l1 el clause1* **and** *literalTrue l1 valuation*
by (*auto simp add*: *formulaTrueIffAllClausesAreTrue clauseTrueIffContainsTrueLiteral*)
from ‹*model valuation* [*clause1*, *clause2*]› **obtain** *l2* :: *Literal*
where *l2 el clause2* **and** *literalTrue l2 valuation*
by (*auto simp add*: *formulaTrueIffAllClausesAreTrue clauseTrueIffContainsTrueLiteral*)
have *clauseTrue* (*resolve clause1 clause2 literal*) *valuation*
proof (*cases literal* = *l1*)
case *False*
with ‹*l1 el clause1*›
have *l1 el* (*resolve clause1 clause2 literal*)
by (*auto simp add*:*resolve-def*)
with ‹*literalTrue l1 valuation*›
show *?thesis*
by (*auto simp add*: *clauseTrueIffContainsTrueLiteral*)
next
case *True*
from ‹*model valuation* [*clause1*, *clause2*]›
have *consistent valuation*
by *simp*
from *True* ‹*literalTrue l1 valuation*› ‹*literalTrue l2 valuation*› ‹*consistent valuation*›
have *literal* ≠ *opposite l2*
by (*auto simp add*:*inconsistentCharacterization*)
with ‹*l2 el clause2*›
have *l2 el* (*resolve clause1 clause2 literal*)
by (*auto simp add*:*resolve-def*)
with ‹*literalTrue l2 valuation*›
show *?thesis*
by (*auto simp add*: *clauseTrueIffContainsTrueLiteral*)
qed
}
thus *?thesis*
by (*simp add*: *formulaEntailsClause-def*)
qed

lemma *formulaEntailsResolvent*:
fixes *formula* :: *Formula* **and** *clause1* :: *Clause* **and** *clause2* :: *Clause*
assumes *formulaEntailsClause formula clause1* **and** *formulaEn-*

tailsClause formula clause2
  shows *formulaEntailsClause formula* (*resolve clause1 clause2 literal*)
proof –
  {
    fix *valuation* :: *Valuation*
    assume *model valuation formula*
    hence *consistent valuation*
      by *simp*
      from ‹*model valuation formula*› ‹*formulaEntailsClause formula clause1*›
    have *clauseTrue clause1 valuation*
      by (*simp add:formulaEntailsClause-def*)
      from ‹*model valuation formula*› ‹*formulaEntailsClause formula clause2*›
    have *clauseTrue clause2 valuation*
      by (*simp add:formulaEntailsClause-def*)
    from ‹*clauseTrue clause1 valuation*› ‹*clauseTrue clause2 valuation*› ‹*consistent valuation*›
    have *clauseTrue* (*resolve clause1 clause2 literal*) *valuation*
      using *resolventIsEntailed*
      by (*auto simp add*: *formulaEntailsClause-def*)
    with ‹*consistent valuation*›
    have *model valuation* (*resolve clause1 clause2 literal*)
      by *simp*
  }
  thus *?thesis*
    by (*simp add*: *formulaEntailsClause-def*)
qed

lemma *resolveFalseClauses*:
  fixes *literal* :: *Literal* **and** *clause1* :: *Clause* **and** *clause2* :: *Clause* **and** *valuation* :: *Valuation*
  assumes
  clauseFalse (*removeAll literal clause1*) *valuation* **and**
  clauseFalse (*removeAll* (*opposite literal*) *clause2*) *valuation*
  shows *clauseFalse* (*resolve clause1 clause2 literal*) *valuation*
proof –
  {
    fix *l* :: *Literal*
    assume *l el* (*resolve clause1 clause2 literal*)
    have *literalFalse l valuation*
    proof–
      from ‹*l el* (*resolve clause1 clause2 literal*)›
      have *l el* (*removeAll literal clause1*) ∨ *l el* (*removeAll* (*opposite literal*) *clause2*)
        unfolding *resolve-def*
        by *simp*
      thus *?thesis*
      proof

```
        assume l el (removeAll literal clause1)
        thus literalFalse l valuation
          using ‹clauseFalse (removeAll literal clause1) valuation›
          by (simp add: clauseFalseIffAllLiteralsAreFalse)
      next
        assume l el (removeAll (opposite literal) clause2)
        thus literalFalse l valuation
            using ‹clauseFalse (removeAll (opposite literal) clause2)
valuation›
          by (simp add: clauseFalseIffAllLiteralsAreFalse)
      qed
    qed
  }
  thus ?thesis
    by (simp add: clauseFalseIffAllLiteralsAreFalse)
qed
```

2.2.14 Unit clauses

Clause is unit in a valuation if all its literals but one are false, and that one is undefined.

```
definition isUnitClause :: Clause ⇒ Literal ⇒ Valuation ⇒ bool
where
isUnitClause uClause uLiteral valuation ==
   uLiteral el uClause ∧
   ¬ (literalTrue uLiteral valuation) ∧
   ¬ (literalFalse uLiteral valuation) ∧
   (∀ literal. literal el uClause ∧ literal ≠ uLiteral ⟶ literalFalse
literal valuation)
```

```
lemma unitLiteralIsEntailed:
  fixes uClause :: Clause and uLiteral :: Literal and formula :: For-
mula and valuation :: Valuation
  assumes isUnitClause uClause uLiteral valuation and formulaEn-
tailsClause formula uClause
  shows formulaEntailsLiteral (formula @ val2form valuation) uLiteral
proof -
  {
    fix valuation'
    assume model valuation' (formula @ val2form valuation)
    hence consistent valuation'
      by simp
    from ‹model valuation' (formula @ val2form valuation)›
    have formulaTrue formula valuation' and formulaTrue (val2form
valuation) valuation'
      by (auto simp add:formulaTrueAppend)
    from ‹formulaTrue formula valuation'› ‹consistent valuation'› ‹for-
mulaEntailsClause formula uClause›
```

```
    have clauseTrue uClause valuation'
      by (simp add:formulaEntailsClause-def)
    then obtain l :: Literal
      where l el uClause literalTrue l valuation'
      by (auto simp add: clauseTrueIffContainsTrueLiteral)
    hence literalTrue uLiteral valuation'
    proof (cases l = uLiteral)
      case True
      with ‹literalTrue l valuation'›
      show ?thesis
        by simp
    next
      case False
      with ‹l el uClause› ‹isUnitClause uClause uLiteral valuation›
      have literalFalse l valuation
        by (simp add: isUnitClause-def)
      from ‹formulaTrue (val2form valuation) valuation'›
      have ∀ literal :: Literal. literal el valuation ⟶ literal el valuation'
        using val2formFormulaTrue [of valuation valuation']
        by simp
      with ‹literalFalse l valuation›
      have literalFalse l valuation'
        by auto
      with ‹literalTrue l valuation'› ‹consistent valuation'›
      have False
        by (simp add:inconsistentCharacterization)
      thus ?thesis ..
    qed
  }
  thus ?thesis
    by (simp add: formulaEntailsLiteral-def)
qed

lemma isUnitClauseRemoveAllUnitLiteralIsFalse:
  fixes uClause :: Clause and uLiteral :: Literal and valuation ::
Valuation
  assumes isUnitClause uClause uLiteral valuation
  shows clauseFalse (removeAll uLiteral uClause) valuation
proof −
  {
    fix literal :: Literal
    assume literal el (removeAll uLiteral uClause)
    hence literal el uClause and literal ≠ uLiteral
      by auto
    with ‹isUnitClause uClause uLiteral valuation›
    have literalFalse literal valuation
      by (simp add: isUnitClause-def)
  }
  thus ?thesis
```

by (*simp add*: *clauseFalseIffAllLiteralsAreFalse*)
qed

lemma *isUnitClauseAppendValuation*:
assumes *isUnitClause uClause uLiteral valuation l* ≠ *uLiteral l* ≠ *opposite uLiteral*
shows *isUnitClause uClause uLiteral* (*valuation* @ [*l*])
using *assms*
unfolding *isUnitClause-def*
by *auto*

lemma *containsTrueNotUnit*:
assumes
l el c **and** *literalTrue l v* **and** *consistent v*
shows
¬ (∃ *ul*. *isUnitClause c ul v*)
using *assms*
unfolding *isUnitClause-def*
by (*auto simp add*: *inconsistentCharacterization*)

lemma *unitBecomesFalse*:
assumes
isUnitClause uClause uLiteral valuation
shows
clauseFalse uClause (*valuation* @ [*opposite uLiteral*])
using *assms*
using *isUnitClauseRemoveAllUnitLiteralIsFalse*[*of uClause uLiteral valuation*]
by (*auto simp add*: *clauseFalseIffAllLiteralsAreFalse*)

2.2.15 Reason clauses

A clause is *reason* for unit propagation of a given literal if it was a unit clause before it is asserted, and became true when it is asserted.

definition
isReason::*Clause* ⇒ *Literal* ⇒ *Valuation* ⇒ *bool*
where
(*isReason clause literal valuation*) ==
(*literal el clause*) ∧
(*clauseFalse* (*removeAll literal clause*) *valuation*) ∧
(∀ *literal'*. *literal' el* (*removeAll literal clause*)
⟶ *precedes* (*opposite literal'*) *literal valuation* ∧ *opposite literal'* ≠ *literal*)

lemma *isReasonAppend*:
fixes *clause* :: *Clause* **and** *literal* :: *Literal* **and** *valuation* :: *Valuation* **and** *valuation'* :: *Valuation*
assumes *isReason clause literal valuation*

shows *isReason clause literal* (*valuation* @ *valuation'*)
proof –
from *assms*
have *literal el clause* **and**
clauseFalse (*removeAll literal clause*) *valuation* (**is** *?false valuation*)
and
∀ *literal'*. *literal' el* (*removeAll literal clause*) ⟶
precedes (*opposite literal'*) *literal valuation* ∧ *opposite literal'* ≠ *literal* (**is** *?precedes valuation*)
unfolding *isReason-def*
by *auto*
moreover
from ‹*?false valuation*›
have *?false* (*valuation* @ *valuation'*)
by (*rule clauseFalseAppendValuation*)
moreover
from ‹*?precedes valuation*›
have *?precedes* (*valuation* @ *valuation'*)
by (*simp add*:*precedesAppend*)
ultimately
show *?thesis*
unfolding *isReason-def*
by *auto*
qed

lemma *isUnitClauseIsReason*:
fixes *uClause* :: *Clause* **and** *uLiteral* :: *Literal* **and** *valuation* :: *Valuation*
assumes *isUnitClause uClause uLiteral valuation uLiteral el valuation'*
shows *isReason uClause uLiteral* (*valuation* @ *valuation'*)
proof –
from *assms*
have *uLiteral el uClause* **and** ¬ *literalTrue uLiteral valuation* **and** ¬ *literalFalse uLiteral valuation*
and ∀ *literal*. *literal el uClause* ∧ *literal* ≠ *uLiteral* ⟶ *literalFalse literal valuation*
unfolding *isUnitClause-def*
by *auto*
hence *clauseFalse* (*removeAll uLiteral uClause*) *valuation*
by (*simp add*: *clauseFalseIffAllLiteralsAreFalse*)
hence *clauseFalse* (*removeAll uLiteral uClause*) (*valuation* @ *valuation'*)
by (*simp add*: *clauseFalseAppendValuation*)
moreover
have ∀ *literal'*. *literal' el* (*removeAll uLiteral uClause*) ⟶
precedes (*opposite literal'*) *uLiteral* (*valuation* @ *valuation'*) ∧ (*opposite literal'*) ≠ *uLiteral*
proof –

{
fix *literal′* :: *Literal*
assume *literal′ el* (*removeAll uLiteral uClause*)
with ‹*clauseFalse* (*removeAll uLiteral uClause*) *valuation*›
have *literalFalse literal′ valuation*
by (*simp add*:*clauseFalseIffAllLiteralsAreFalse*)
with ‹¬ *literalTrue uLiteral valuation*› ‹¬ *literalFalse uLiteral valuation*›
have *precedes* (*opposite literal′*) *uLiteral* (*valuation* @ *valuation′*) ∧ (*opposite literal′*) ≠ *uLiteral*
using ‹*uLiteral el valuation′*›
using *precedesMemberHeadMemberTail* [*of opposite literal′ valuation uLiteral valuation′*]
by *auto*
}
thus *?thesis*
by *simp*
qed
ultimately
show *?thesis* **using** ‹*uLiteral el uClause*›
by (*auto simp add*: *isReason-def*)
qed

lemma *isReasonHoldsInPrefix*:
fixes *prefix* :: *Valuation* **and** *valuation* :: *Valuation* **and** *clause* :: *Clause* **and** *literal* :: *Literal*
assumes
literal el prefix **and**
isPrefix prefix valuation **and**
isReason clause literal valuation
shows
isReason clause literal prefix
proof −
from ‹*isReason clause literal valuation*›
have
literal el clause **and**
clauseFalse (*removeAll literal clause*) *valuation* (**is** *?false valuation*) **and**
∀ *literal′*. *literal′ el* (*removeAll literal clause*) ⟶
precedes (*opposite literal′*) *literal valuation* ∧ *opposite literal′* ≠ *literal* (**is** *?precedes valuation*)
unfolding *isReason-def*
by *auto*
{
fix *literal′* :: *Literal*
assume *literal′ el* (*removeAll literal clause*)
with ‹*?precedes valuation*›
have *precedes* (*opposite literal′*) *literal valuation* (*opposite literal′*) ≠ *literal*

by *auto*
with ‹*literal el prefix*› ‹*isPrefix prefix valuation*›
have *precedes* (*opposite literal'*) *literal prefix* ∧ (*opposite literal'*) ≠ *literal*
using *laterInPrefixRetainsPrecedes* [*of prefix valuation opposite literal' literal*]
by *auto*
}
note ∗ = *this*
hence *?precedes prefix*
by *auto*
moreover
have *?false prefix*
proof −
{
fix *literal'* :: *Literal*
assume *literal' el* (*removeAll literal clause*)
from ‹*literal' el* (*removeAll literal clause*)› ∗
have *precedes* (*opposite literal'*) *literal prefix*
by *simp*
with ‹*literal el prefix*›
have *literalFalse literal' prefix*
unfolding *precedes-def*
by (*auto split*: *if-split-asm*)
}
thus *?thesis*
by (*auto simp add*:*clauseFalseIffAllLiteralsAreFalse*)
qed
ultimately
show *?thesis* **using** ‹*literal el clause*›
unfolding *isReason-def*
by *auto*
qed

2.2.16 Last asserted literal of a list

lastAssertedLiteral from a list is the last literal from a clause that is asserted in a valuation.

definition
isLastAssertedLiteral::*Literal* ⇒ *Literal list* ⇒ *Valuation* ⇒ *bool*
where
isLastAssertedLiteral literal clause valuation ==
literal el clause ∧
literalTrue literal valuation ∧
(∀ *literal'*. *literal' el clause* ∧ *literal'* ≠ *literal* ⟶ ¬ *precedes literal literal' valuation*)

Function that gets the last asserted literal of a list - specified only by its postcondition.

```
definition
getLastAssertedLiteral :: Literal list ⇒ Valuation ⇒ Literal
where
getLastAssertedLiteral clause valuation ==
  last (filter (λ l::Literal. l el clause) valuation)

lemma getLastAssertedLiteralCharacterization:
assumes
  clauseFalse clause valuation
  clause ≠ []
  uniq valuation
shows
  isLastAssertedLiteral (getLastAssertedLiteral (oppositeLiteralList clause)
valuation) (oppositeLiteralList clause) valuation
proof−
  let ?oppc = oppositeLiteralList clause
  let ?l = getLastAssertedLiteral ?oppc valuation
  let ?f = filter (λ l. l el ?oppc) valuation

  have ?oppc ≠ []
    using ‹clause ≠ []›
    using oppositeLiteralListNonempty[of clause]
    by simp
  then obtain l'::Literal
    where l' el ?oppc
    by force

  have ∀ l::Literal. l el ?oppc ⟶ l el valuation
  proof
    fix l::Literal
    show l el ?oppc ⟶ l el valuation
    proof
      assume l el ?oppc
      hence opposite l el clause
          using literalElListIffOppositeLiteralElOppositeLiteralList[of l
?oppc]
        by simp
      thus l el valuation
        using ‹clauseFalse clause valuation›
        using clauseFalseIffAllLiteralsAreFalse[of clause valuation]
        by auto
    qed
  qed
  hence l' el valuation
    using ‹l' el ?oppc›
    by simp
  hence l' el ?f
    using ‹l' el ?oppc›
    by simp
```

```
hence ?f ≠ []
  using set-empty[of ?f]
  by auto
hence last ?f el ?f
  using last-in-set[of ?f]
  by simp
hence ?l el ?oppc literalTrue ?l valuation
  unfolding getLastAssertedLiteral-def
  by auto
moreover
have ∀ literal'. literal' el ?oppc ∧ literal' ≠ ?l ⟶
                ¬ precedes ?l literal' valuation
proof
  fix literal'
   show literal' el ?oppc ∧ literal' ≠ ?l ⟶ ¬ precedes ?l literal'
valuation
  proof
    assume literal' el ?oppc ∧ literal' ≠ ?l
    show ¬ precedes ?l literal' valuation
    proof (cases literalTrue literal' valuation)
      case False
      thus ?thesis
        unfolding precedes-def
        by simp
    next
      case True
      with ‹literal' el ?oppc ∧ literal' ≠ ?l›
      have literal' el ?f
        by simp
      have uniq ?f
        using ‹uniq valuation›
        by (simp add: uniqDistinct)
      hence ¬ precedes ?l literal' ?f
        using lastPrecedesNoElement[of ?f]
        using ‹literal' el ?oppc ∧ literal' ≠ ?l›
        unfolding getLastAssertedLiteral-def
        by auto
      thus ?thesis
        using precedesFilter[of ?l literal' valuation λ l. l el ?oppc]
        using ‹literal' el ?oppc ∧ literal' ≠ ?l›
        using ‹?l el ?oppc›
        by auto
    qed
  qed
qed
ultimately
show ?thesis
  unfolding isLastAssertedLiteral-def
  by simp
```

qed

lemma *lastAssertedLiteralIsUniq*:
fixes *literal* :: *Literal* **and** *literal'* :: *Literal* **and** *literalList* :: *Literal list* **and** *valuation* :: *Valuation*
assumes
lastL: *isLastAssertedLiteral literal literalList valuation* **and**
lastL': *isLastAssertedLiteral literal' literalList valuation*
shows *literal* = *literal'*
using *assms*
proof −
from *lastL* **have** ∗:
literal el literalList
∀ *l*. *l el literalList* ∧ *l* ≠ *literal* ⟶ ¬ *precedes literal l valuation*
and
literalTrue literal valuation
by (*auto simp add*: *isLastAssertedLiteral-def*)
from *lastL'* **have** ∗∗:
literal' el literalList
∀ *l*. *l el literalList* ∧ *l* ≠ *literal'* ⟶ ¬ *precedes literal' l valuation*
and
literalTrue literal' valuation
by (*auto simp add*: *isLastAssertedLiteral-def*)
{
assume *literal'* ≠ *literal*
with ∗ ∗∗ **have** ¬ *precedes literal literal' valuation* **and** ¬ *precedes literal' literal valuation*
by *auto*
with ‹*literalTrue literal valuation*› ‹*literalTrue literal' valuation*›
have *False*
using *precedesTotalOrder*[*of literal valuation literal'*]
unfolding *precedes-def*
by *simp*
}
thus *?thesis*
by *auto*
qed

lemma *isLastAssertedCharacterization*:
fixes *literal* :: *Literal* **and** *literalList* :: *Literal list* **and** *v* :: *Valuation*
assumes *isLastAssertedLiteral literal* (*oppositeLiteralList literalList*) *valuation*
shows *opposite literal el literalList* **and** *literalTrue literal valuation*
proof −
from *assms* **have**
∗: *literal el* (*oppositeLiteralList literalList*) **and** ∗∗: *literalTrue literal valuation*
by (*auto simp add*: *isLastAssertedLiteral-def*)
from ∗ **show** *opposite literal el literalList*

using *literalElListIffOppositeLiteralElOppositeLiteralList* [*of literal oppositeLiteralList literalList*]
by *simp*
from ∗∗ **show** *literalTrue literal valuation*
by *simp*
qed

lemma *isLastAssertedLiteralSubset*:
assumes
isLastAssertedLiteral l c M
set c′ ⊆ *set c*
l el c′
shows
isLastAssertedLiteral l c′ M
using *assms*
unfolding *isLastAssertedLiteral-def*
by *auto*

lemma *lastAssertedLastInValuation*:
fixes *literal* :: *Literal* **and** *literalList* :: *Literal list* **and** *valuation* :: *Valuation*
assumes *literal el literalList* **and** ¬ *literalTrue literal valuation*
shows *isLastAssertedLiteral literal literalList* (*valuation* @ [*literal*])
proof −
have *literalTrue literal* [*literal*]
by *simp*
hence *literalTrue literal* (*valuation* @ [*literal*])
by *simp*
moreover
have ∀ *l*. *l el literalList* ∧ *l* ≠ *literal* ⟶ ¬ *precedes literal l* (*valuation* @ [*literal*])
proof −
{
fix *l*
assume *l el literalList l* ≠ *literal*
have ¬ *precedes literal l* (*valuation* @ [*literal*])
proof (*cases literalTrue l valuation*)
case *False*
with ‹*l* ≠ *literal*›
show *?thesis*
unfolding *precedes-def*
by *simp*
next
case *True*
from ‹¬ *literalTrue literal valuation*› ‹*literalTrue literal* [*literal*]› ‹*literalTrue l valuation*›
have *precedes l literal* (*valuation* @ [*literal*])
using *precedesMemberHeadMemberTail*[*of l valuation literal* [*literal*]]

by *auto*
with ‹*l* ≠ *literal*› ‹*literalTrue l valuation*› ‹*literalTrue literal* [*literal*]›
show *?thesis*
using *precedesAntisymmetry*[*of l valuation* @ [*literal*] *literal*]
unfolding *precedes-def*
by *auto*
qed
} **thus** *?thesis*
by *simp*
qed
ultimately
show *?thesis* **using** ‹*literal el literalList*›
by (*simp add*:*isLastAssertedLiteral-def*)
qed

end

3 Trail datatype definition and its properties

theory *Trail*
imports *MoreList*
begin

Trail is a list in which some elements can be marked.

type-synonym *'a Trail* = (*'a*∗*bool*) *list*

abbreviation
element :: (*'a*∗*bool*) ⇒ *'a*
where *element x* == *fst x*

abbreviation
marked :: (*'a*∗*bool*) ⇒ *bool*
where *marked x* == *snd x*

3.1 Trail elements

Elements of the trail with marks removed

primrec
elements :: *'a Trail* ⇒ *'a list*
where
elements [] = []
| *elements* (*h*#*t*) = (*element h*) # (*elements t*)

lemma
elements t = *map fst t*

```
by (induct t) auto

lemma eitherMarkedOrNotMarkedElement:
  shows a = (element a, True) ∨ a = (element a, False)
by (cases a) auto

lemma eitherMarkedOrNotMarked:
  assumes e ∈ set (elements M)
  shows (e, True) ∈ set M ∨ (e, False) ∈ set M
using assms
proof (induct M)
  case (Cons m M')
  thus ?case
    proof (cases e = element m)
      case True
      thus ?thesis
        using eitherMarkedOrNotMarkedElement [of m]
        by auto
    next
      case False
      with Cons
      show ?thesis
        by auto
    qed
qed simp

lemma elementMemElements [simp]:
  assumes x ∈ set M
  shows element x ∈ set (elements M)
using assms
by (induct M) (auto split: if-split-asm)

lemma elementsAppend [simp]:
  shows elements (a @ b) = elements a @ elements b
by (induct a) auto

lemma elementsEmptyIffTrailEmpty [simp]:
  shows (elements list = []) = (list = [])
by (induct list) auto

lemma elementsButlastTrailIsButlastElementsTrail [simp]:
  shows elements (butlast M) = butlast (elements M)
by (induct M) auto

lemma elementLastTrailIsLastElementsTrail [simp]:
  assumes M ≠ []
  shows element (last M) = last (elements M)
using assms
by (induct M) auto
```

lemma *isPrefixElements*:
assumes *isPrefix a b*
shows *isPrefix* (*elements a*) (*elements b*)
using *assms*
unfolding *isPrefix-def*
by *auto*

lemma *prefixElementsAreTrailElements*:
assumes
isPrefix p M
shows
set (*elements p*) ⊆ *set* (*elements M*)
using *assms*
unfolding *isPrefix-def*
by *auto*

lemma *uniqElementsTrailImpliesUniqElementsPrefix*:
assumes
isPrefix p M **and** *uniq* (*elements M*)
shows
uniq (*elements p*)
proof−
from ‹*isPrefix p M*›
obtain *s*
where *M* = *p* @ *s*
unfolding *isPrefix-def*
by *auto*
with ‹*uniq* (*elements M*)›
show *?thesis*
using *uniqAppend*[*of elements p elements s*]
by *simp*
qed

lemma [*simp*]:
assumes (*e*, *d*) ∈ *set M*
shows *e* ∈ *set* (*elements M*)
using *assms*
by (*induct M*) *auto*

lemma *uniqImpliesExclusiveTrueOrFalse*:
assumes
(*e*, *d*) ∈ *set M* **and** *uniq* (*elements M*)
shows
¬ (*e*, ¬ *d*) ∈ *set M*
using *assms*
proof (*induct M*)
case (*Cons m M'*)
{

```
    assume (e, d) = m
    hence (e, ¬ d) ≠ m
      by auto
    from ‹(e, d) = m› ‹uniq (elements (m # M'))›
    have ¬ (e, d) ∈ set M'
      by (auto simp add: uniqAppendIff)
    with Cons
    have ?case
      by (auto split: if-split-asm)
  }
  moreover
  {
    assume (e, ¬ d) = m
    hence (e, d) ≠ m
      by auto
    from ‹(e, ¬ d) = m› ‹uniq (elements (m # M'))›
    have ¬ (e, ¬ d) ∈ set M'
      by (auto simp add: uniqAppendIff)
    with Cons
    have ?case
      by (auto split: if-split-asm)
  }
  moreover
  {
    assume (e, d) ≠ m (e, ¬ d) ≠ m
    from ‹(e, d) ≠ m› ‹(e, d) ∈ set (m # M')› have
      (e, d) ∈ set M'
      by simp
    with ‹uniq (elements (m # M'))› Cons(1)
    have ¬ (e, ¬ d) ∈ set M'
      by simp
    with ‹(e, ¬ d) ≠ m›
    have ?case
      by simp
  }
  moreover
  {
    have (e, d) = m ∨ (e, ¬ d) = m ∨ (e, d) ≠ m ∧ (e, ¬ d) ≠ m
      by auto
  }
  ultimately
  show ?case
    by auto
qed simp
```

3.2 Marked trail elements

primrec
markedElements :: *'a Trail ⇒ 'a list*

where
  markedElements [] = []
| *markedElements* (*h*#*t*) = (*if* (*marked h*) *then* (*element h*) # (*markedElements t*) *else* (*markedElements t*))

lemma
markedElements t = (*elements* (*filter snd t*))
by (*induct t*) *auto*

lemma *markedElementIsMarkedTrue*:
  shows (*m* ∈ *set* (*markedElements M*)) = ((*m*, *True*) ∈ *set M*)
by (*induct M*) (*auto split*: *if-split-asm*)

lemma *markedElementsAppend*:
  shows *markedElements* (*M1* @ *M2*) = *markedElements M1* @ *markedElements M2*
by (*induct M1*) *auto*

lemma *markedElementsAreElements*:
  assumes *m* ∈ *set* (*markedElements M*)
  shows   *m* ∈ *set* (*elements M*)
using *assms markedElementIsMarkedTrue*[*of m M*]
by *auto*

lemma *markedAndMemberImpliesIsMarkedElement*:
  assumes *marked m m* ∈ *set M*
  shows (*element m*) ∈ *set* (*markedElements M*)
proof−
  have *m* = (*element m*, *marked m*)
    by *auto*
  with ‹*marked m*›
  have *m* = (*element m*, *True*)
    by *simp*
  with ‹*m* ∈ *set M*› **have**
    (*element m*, *True*) ∈ *set M*
    by *simp*
  thus *?thesis*
    using *markedElementIsMarkedTrue* [*of element m M*]
    by *simp*
qed

lemma *markedElementsPrefixAreMarkedElementsTrail*:
  assumes *isPrefix p M m* ∈ *set* (*markedElements p*)
  shows *m* ∈ *set* (*markedElements M*)
proof−
  from ‹*m* ∈ *set* (*markedElements p*)›
  have (*m*, *True*) ∈ *set p*
    by (*simp add*: *markedElementIsMarkedTrue*)
  with ‹*isPrefix p M*›

```
  have (m, True) ∈ set M
    using prefixIsSubset[of p M]
    by auto
  thus ?thesis
    by (simp add: markedElementIsMarkedTrue)
qed

lemma markedElementsTrailMemPrefixAreMarkedElementsPrefix:
  assumes
  uniq (elements M) and
  isPrefix p M and
  m ∈ set (elements p) and
  m ∈ set (markedElements M)
  shows
  m ∈ set (markedElements p)
proof−
  from ‹m ∈ set (markedElements M)› have (m, True) ∈ set M
    by (simp add: markedElementIsMarkedTrue)
  with ‹uniq (elements M)› ‹m ∈ set (elements p)›
  have (m, True) ∈ set p
  proof−
    {
      assume (m, False) ∈ set p
      with ‹isPrefix p M›
      have (m, False) ∈ set M
        using prefixIsSubset[of p M]
        by auto
      with ‹(m, True) ∈ set M› ‹uniq (elements M)›
      have False
        using uniqImpliesExclusiveTrueOrFalse[of m True M]
        by simp
    }
    with ‹m ∈ set (elements p)›
    show ?thesis
      using eitherMarkedOrNotMarked[of m p]
      by auto
  qed
  thus ?thesis
    using markedElementIsMarkedTrue[of m p]
    by simp
qed
```

3.3 Prefix before/upto a trail element

Elements of the trail before the first occurrence of a given element - not incuding it

```
primrec
prefixBeforeElement :: 'a ⇒ 'a Trail ⇒ 'a Trail
where
```

prefixBeforeElement e [] = []
| *prefixBeforeElement e* (*h*#*t*) =
(*if* (*element h*) = *e then*
[]
else
(*h* # (*prefixBeforeElement e t*))
)

lemma *prefixBeforeElement e t* = *takeWhile* (λ *e*′. *element e*′ ≠ *e*) *t*
by (*induct t*) *auto*

lemma *prefixBeforeElement e t* = *take* (*firstPos e* (*elements t*)) *t*
by (*induct t*) *auto*

Elements of the trail before the first occurrence of a given element - incuding it

primrec
prefixToElement :: ′*a* ⇒ ′*a Trail* ⇒ ′*a Trail*
where
prefixToElement e [] = []
| *prefixToElement e* (*h*#*t*) =
(*if* (*element h*) = *e then*
[*h*]
else
(*h* # (*prefixToElement e t*))
)

lemma *prefixToElement e t* = *take* ((*firstPos e* (*elements t*)) + *1*) *t*
by (*induct t*) *auto*

lemma *isPrefixPrefixToElement*:
shows *isPrefix* (*prefixToElement e t*) *t*
unfolding *isPrefix-def*
by (*induct t*) *auto*

lemma *isPrefixPrefixBeforeElement*:
shows *isPrefix* (*prefixBeforeElement e t*) *t*
unfolding *isPrefix-def*
by (*induct t*) *auto*

lemma *prefixToElementContainsTrailElement*:
assumes *e* ∈ *set* (*elements M*)
shows *e* ∈ *set* (*elements* (*prefixToElement e M*))
using *assms*
by (*induct M*) *auto*

lemma *prefixBeforeElementDoesNotContainTrailElement*:
assumes *e* ∈ *set* (*elements M*)

```
  shows e ∉ set (elements (prefixBeforeElement e M))
using assms
by (induct M) auto

lemma prefixToElementAppend:
  shows prefixToElement e (M1 @ M2) =
           (if e ∈ set (elements M1) then
               prefixToElement e M1
            else
               M1 @ prefixToElement e M2
            )
by (induct M1) auto


lemma prefixToElementToPrefixElement:
  assumes
  isPrefix p M and e ∈ set (elements p)
  shows
  prefixToElement e M = prefixToElement e p
using assms
unfolding isPrefix-def
proof (induct p arbitrary: M)
  case (Cons a p')
  then obtain s
    where (a # p') @ s = M
    by auto
  show ?case
  proof (cases (element a) = e)
    case True
    from True ‹(a # p') @ s = M› have prefixToElement e M = [a]
      by auto
    moreover
    from True have prefixToElement e (a # p') = [a]
      by auto
    ultimately
    show ?thesis
      by simp
  next
    case False
    from False ‹(a # p') @ s = M› have prefixToElement e M = a
# prefixToElement e (p' @ s)
      by auto
    moreover
    from False have prefixToElement e (a # p') = a # prefixToElement
e p'
      by simp
    moreover
    from False ‹e ∈ set (elements (a # p'))› have e ∈ set (elements
p')
```

by *simp*
have *? s* . $(p' @ s = p' @ s)$
by *simp*
from ‹$e \in set\ (elements\ p')$› ‹$?\ s.\ (p' @ s = p' @ s)$›
have *prefixToElement e* $(p' @ s)$ = *prefixToElement e* p'
using *Cons*(*1*) [*of* $p' @ s$]
by *simp*
ultimately show *?thesis*
by *simp*
qed
qed *simp*

3.4 Marked elements upto a given trail element

Marked elements of the trail upto the given element (which is also included if it is marked)

definition
markedElementsTo :: $'a \Rightarrow 'a\ Trail \Rightarrow 'a\ list$
where
markedElementsTo e t = *markedElements* (*prefixToElement e t*)

lemma *markedElementsToArePrefixOfMarkedElements*:
shows *isPrefix* (*markedElementsTo e M*) (*markedElements M*)
unfolding *isPrefix-def*
unfolding *markedElementsTo-def*
by (*induct M*) *auto*

lemma *markedElementsToAreMarkedElements*:
assumes $m \in set$ (*markedElementsTo e M*)
shows $m \in set$ (*markedElements M*)
using *assms*
using *markedElementsToArePrefixOfMarkedElements*[*of e M*]
using *prefixIsSubset*
by *auto*

lemma *markedElementsToNonMemberAreAllMarkedElements*:
assumes $e \notin set\ (elements\ M)$
shows *markedElementsTo e M* = *markedElements M*
using *assms*
unfolding *markedElementsTo-def*
by (*induct M*) *auto*

lemma *markedElementsToAppend*:
shows *markedElementsTo e* (*M1* @ *M2*) =
(*if* $e \in set$ (*elements M1*) *then*
markedElementsTo e M1
else
markedElements M1 @ *markedElementsTo e M2*
)

unfolding *markedElementsTo-def*
by (*auto simp add*: *prefixToElementAppend markedElementsAppend*)

lemma *markedElementsEmptyImpliesMarkedElementsToEmpty*:
  assumes *markedElements M* = []
  shows *markedElementsTo e M* = []
using *assms*
using *markedElementsToArePrefixOfMarkedElements* [*of e M*]
unfolding *isPrefix-def*
by *auto*

lemma *markedElementIsMemberOfItsMarkedElementsTo*:
  assumes
  uniq (*elements M*) **and** *marked e* **and** *e* ∈ *set M*
  shows
  element e ∈ *set* (*markedElementsTo* (*element e*) *M*)
using *assms*
unfolding *markedElementsTo-def*
by (*induct M*) (*auto split*: *if-split-asm*)

lemma *markedElementsToPrefixElement*:
  assumes *isPrefix p M* **and** *e* ∈ *set* (*elements p*)
  shows *markedElementsTo e M* = *markedElementsTo e p*
unfolding *markedElementsTo-def*
using *assms*
by (*simp add*: *prefixToElementToPrefixElement*)

3.5 Last marked element in a trail

definition
lastMarked :: *'a Trail* ⇒ *'a*
where
lastMarked t = *last* (*markedElements t*)

lemma *lastMarkedIsMarkedElement*:
  assumes *markedElements M* ≠ []
  shows *lastMarked M* ∈ *set* (*markedElements M*)
using *assms*
unfolding *lastMarked-def*
by *simp*

lemma *removeLastMarkedFromMarkedElementsToLastMarkedAreAll-MarkedElementsInPrefixLastMarked*:
  assumes
  markedElements M ≠ []
  shows
  removeAll (*lastMarked M*) (*markedElementsTo* (*lastMarked M*) *M*)
= *markedElements* (*prefixBeforeElement* (*lastMarked M*) *M*)
using *assms*

unfolding *lastMarked-def*
unfolding *markedElementsTo-def*
by (*induct M*) *auto*

lemma *markedElementsToLastMarkedAreAllMarkedElements*:
  assumes
  uniq (*elements M*) **and** *markedElements M* $\neq$ []
  shows
  markedElementsTo (*lastMarked M*) *M* = *markedElements M*
using *assms*
unfolding *lastMarked-def*
unfolding *markedElementsTo-def*
by (*induct M*) (*auto simp add*: *markedElementsAreElements*)

lemma *lastTrailElementMarkedImpliesMarkedElementsToLastElementAreAllMarkedElements*:
  assumes
  marked (*last M*) **and** *last* (*elements M*) $\notin$ *set* (*butlast* (*elements M*))
  shows
  markedElementsTo (*last* (*elements M*)) *M* = *markedElements M*
using *assms*
unfolding *markedElementsTo-def*
by (*induct M*) *auto*

lemma *lastMarkedIsMemberOfItsMarkedElementsTo*:
  assumes
  uniq (*elements M*) **and** *markedElements M* $\neq$ []
  shows
  lastMarked M $\in$ *set* (*markedElementsTo* (*lastMarked M*) *M*)
using *assms*
using *markedElementsToLastMarkedAreAllMarkedElements* [*of M*]
using *lastMarkedIsMarkedElement* [*of M*]
by *auto*

lemma *lastTrailElementNotMarkedImpliesMarkedElementsToLAreMarkedElementsToLInButlastTrail*:
  assumes $\neg$ *marked* (*last M*)
  shows *markedElementsTo e M* = *markedElementsTo e* (*butlast M*)
using *assms*
unfolding *markedElementsTo-def*
by (*induct M*) *auto*

3.6 Level of a trail element

Level of an element is the number of marked elements that precede it

definition
elementLevel :: $'a \Rightarrow 'a$ *Trail* $\Rightarrow$ *nat*
where

```
elementLevel e t = length (markedElementsTo e t)

lemma elementLevelMarkedGeq1:
  assumes
  uniq (elements M) and e ∈ set (markedElements M)
  shows
  elementLevel e M >= 1
proof−
  from ‹e ∈ set (markedElements M)› have (e, True) ∈ set M
    by (simp add: markedElementIsMarkedTrue)
  with ‹uniq (elements M)›  have e ∈ set (markedElementsTo e M)
     using markedElementIsMemberOfItsMarkedElementsTo[of M (e,
True)]
    by simp
  hence markedElementsTo e M ≠ []
    by auto
  thus ?thesis
    unfolding elementLevel-def
    using length-greater-0-conv[of markedElementsTo e M]
    by arith
qed

lemma elementLevelAppend:
  assumes a ∈ set (elements M)
  shows elementLevel a M = elementLevel a (M @ M')
using assms
unfolding elementLevel-def
by (simp add: markedElementsToAppend)


lemma elementLevelPrecedesLeq:
  assumes
  precedes a b (elements M)
  shows
  elementLevel a M ≤ elementLevel b M
using assms
proof (induct M)
  case (Cons m M')
  {
    assume a = element m
    hence ?case
      unfolding elementLevel-def
      unfolding markedElementsTo-def
      by simp
  }
  moreover
  {
    assume b = element m
    {
```

```
      assume a ≠ b
      hence ¬ precedes a b (b # (elements M'))
        by (rule noElementsPrecedesFirstElement)
      with ‹b = element m› ‹precedes a b (elements (m # M'))›
      have False
        by simp
    }
    hence a = b
      by auto
    hence ?case
      by simp
  }
  moreover
  {
    assume a ≠ element m b ≠ element m
    moreover
    from ‹precedes a b (elements (m # M'))›
    have a ∈ set (elements (m # M')) b ∈ set (elements (m # M'))
      unfolding precedes-def
      by (auto split: if-split-asm)
    from ‹a ≠ element m› ‹a ∈ set (elements (m # M'))›
    have a ∈ set (elements M')
      by simp
    moreover
    from ‹b ≠ element m› ‹b ∈ set (elements (m # M'))›
    have b ∈ set (elements M')
      by simp
    ultimately
    have elementLevel a M' ≤ elementLevel b M'
      using Cons
      unfolding precedes-def
      by auto
    hence ?case
      using ‹a ≠ element m› ‹b ≠ element m›
      unfolding elementLevel-def
      unfolding markedElementsTo-def
      by auto
  }
  ultimately
  show ?case
    by auto
next
  case Nil
  thus ?case
    unfolding precedes-def
    by simp
qed
```

lemma *elementLevelPrecedesMarkedElementLt*:
assumes
uniq (*elements M*) **and**
$e \neq d$ **and**
$d \in$ *set* (*markedElements M*) **and**
precedes e d (*elements M*)
shows
elementLevel e M < *elementLevel d M*
using *assms*
proof (*induct M*)
case (*Cons m M'*)
{
assume *e* = *element m*
moreover
with ‹$e \neq d$› **have** $d \neq$ *element m*
by *simp*
moreover
from ‹*uniq* (*elements* (*m* # *M'*))› ‹$d \in$ *set* (*markedElements* (*m* # *M'*))›
have *1* ≤ *elementLevel d* (*m* # *M'*)
using *elementLevelMarkedGeq1*[*of m* # *M' d*]
by *auto*
moreover
from ‹$d \neq$ *element m*› ‹$d \in$ *set* (*markedElements* (*m* # *M'*))›
have $d \in$ *set* (*markedElements M'*)
by (*simp split*: *if-split-asm*)
from ‹*uniq* (*elements* (*m* # *M'*))› ‹$d \in$ *set* (*markedElements M'*)›
have *1* ≤ *elementLevel d M'*
using *elementLevelMarkedGeq1*[*of M' d*]
by *auto*
ultimately
have *?case*
unfolding *elementLevel-def*
unfolding *markedElementsTo-def*
by (*auto split*: *if-split-asm*)
}
moreover
{
assume *d* = *element m*
from ‹$e \neq d$› **have** ¬ *precedes e d* (*d* # (*elements M'*))
using *noElementsPrecedesFirstElement*[*of e d elements M'*]
by *simp*
with ‹*d* = *element m*› ‹*precedes e d* (*elements* (*m* # *M'*))›
have *False*
by *simp*
hence *?case*
by *simp*
}
moreover

{
  assume *e* ≠ *element m d* ≠ *element m*
  moreover
  from ‹*precedes e d* (*elements* (*m* # *M'*))›
  have *e* ∈ *set* (*elements* (*m* # *M'*)) *d* ∈ *set* (*elements* (*m* # *M'*))
    unfolding *precedes-def*
    by (*auto split*: *if-split-asm*)
  from ‹*e* ≠ *element m*› ‹*e* ∈ *set* (*elements* (*m* # *M'*))›
  have *e* ∈ *set* (*elements M'*)
    by *simp*
  moreover
  from ‹*d* ≠ *element m*› ‹*d* ∈ *set* (*elements* (*m* # *M'*))›
  have *d* ∈ *set* (*elements M'*)
    by *simp*
  moreover
  from ‹*d* ≠ *element m*› ‹*d* ∈ *set* (*markedElements* (*m* # *M'*))›
  have *d* ∈ *set* (*markedElements M'*)
    by (*simp split*: *if-split-asm*)
  ultimately
  have *elementLevel e M'* < *elementLevel d M'*
    using ‹*uniq* (*elements* (*m* # *M'*))› *Cons*
    unfolding *precedes-def*
    by *auto*
  hence *?case*
    using ‹*e* ≠ *element m*› ‹*d* ≠ *element m*›
    unfolding *elementLevel-def*
    unfolding *markedElementsTo-def*
    by *auto*
}
ultimately
show *?case*
  by *auto*
qed *simp*

lemma *differentMarkedElementsHaveDifferentLevels*:
  assumes
  uniq (*elements M*) **and**
  a ∈ *set* (*markedElements M*) **and**
  b ∈ *set* (*markedElements M*) **and**
  a ≠ *b*
  shows *elementLevel a M* ≠ *elementLevel b M*
proof−
  from ‹*a* ∈ *set* (*markedElements M*)›
  have *a* ∈ *set* (*elements M*)
    by (*simp add*: *markedElementsAreElements*)
  moreover
  from ‹*b* ∈ *set* (*markedElements M*)›
  have *b* ∈ *set* (*elements M*)
    by (*simp add*: *markedElementsAreElements*)

ultimately
have *precedes a b* (*elements M*) ∨ *precedes b a* (*elements M*)
using ‹$a \neq b$›
using *precedesTotalOrder*[*of a elements M b*]
by *simp*
moreover
{
assume *precedes a b* (*elements M*)
with *assms*
have *?thesis*
using *elementLevelPrecedesMarkedElementLt*[*of M a b*]
by *auto*
}
moreover
{
assume *precedes b a* (*elements M*)
with *assms*
have *?thesis*
using *elementLevelPrecedesMarkedElementLt*[*of M b a*]
by *auto*
}
ultimately
show *?thesis*
by *auto*
qed

3.7 Current trail level

Current level is the number of marked elements in the trail

definition
currentLevel :: *'a Trail* ⇒ *nat*
where
currentLevel t = *length* (*markedElements t*)

lemma *currentLevelNonMarked*:
shows *currentLevel M* = *currentLevel* (*M* @ [(*l*, *False*)])
by (*auto simp add*:*currentLevel-def markedElementsAppend*)

lemma *currentLevelPrefix*:
assumes *isPrefix a b*
shows *currentLevel a* <= *currentLevel b*
using *assms*
unfolding *isPrefix-def*
unfolding *currentLevel-def*
by (*auto simp add*: *markedElementsAppend*)

lemma *elementLevelLeqCurrentLevel*:
shows *elementLevel a M* ≤ *currentLevel M*
proof−

```
  have isPrefix (prefixToElement a M) M
    using isPrefixPrefixToElement[of a M]
    .
  then obtain s
    where prefixToElement a M @ s = M
    unfolding isPrefix-def
    by auto
  hence M = prefixToElement a M @ s
    by (rule sym)
  hence currentLevel M = currentLevel (prefixToElement a M @ s)
    by simp
  hence currentLevel M = length (markedElements (prefixToElement
a M)) + length (markedElements s)
    unfolding currentLevel-def
    by (simp add: markedElementsAppend)
  thus ?thesis
    unfolding elementLevel-def
    unfolding markedElementsTo-def
    by simp
qed

lemma elementOnCurrentLevel:
  assumes a ∉ set (elements M)
  shows elementLevel a (M @ [(a, d)]) = currentLevel  (M @ [(a, d)])
using assms
unfolding currentLevel-def
unfolding elementLevel-def
unfolding markedElementsTo-def
by (auto simp add: prefixToElementAppend)
```

3.8 Prefix to a given trail level

Prefix is made or elements of the trail up to a given element level

```
primrec
prefixToLevel-aux :: 'a Trail ⇒ nat ⇒ nat ⇒ 'a Trail
where
  (prefixToLevel-aux [] l cl) = []
| (prefixToLevel-aux (h#t) l cl) =
  (if (marked h) then
    (if (cl >= l) then [] else (h # (prefixToLevel-aux t l (cl+1))))
  else
    (h # (prefixToLevel-aux t l cl))
  )

definition
prefixToLevel :: nat ⇒ 'a Trail ⇒ 'a Trail
where
prefixToLevel-def: (prefixToLevel l t) == (prefixToLevel-aux t l 0)
```

```
lemma isPrefixPrefixToLevel-aux:
  shows ∃ s. prefixToLevel-aux t l i @ s = t
by (induct t arbitrary: i) auto

lemma isPrefixPrefixToLevel:
  shows (isPrefix (prefixToLevel l t) t)
using isPrefixPrefixToLevel-aux[of t l]
unfolding isPrefix-def
unfolding prefixToLevel-def
by simp

lemma currentLevelPrefixToLevel-aux:
  assumes l ≥ i
  shows currentLevel (prefixToLevel-aux M l i) <= l − i
using assms
proof (induct M arbitrary: i)
  case (Cons m M')
  {
    assume marked m i = l
    hence ?case
      unfolding currentLevel-def
      by simp
  }
  moreover
  {
    assume marked m i < l
    hence ?case
      using Cons(1) [of i+1]
      unfolding currentLevel-def
      by simp
  }
  moreover
  {
    assume ¬ marked m
    hence ?case
      using Cons
      unfolding currentLevel-def
      by simp
  }
  ultimately
  show ?case
    using ‹i <= l›
    by auto
next
  case Nil
  thus ?case
    unfolding currentLevel-def
    by simp
```

```
qed

lemma currentLevelPrefixToLevel:
  shows currentLevel (prefixToLevel level M) ≤ level
using currentLevelPrefixToLevel-aux[of 0 level M]
unfolding prefixToLevel-def
by simp

lemma currentLevelPrefixToLevelEq-aux:
  assumes l ≥ i currentLevel M >= l − i
  shows currentLevel (prefixToLevel-aux M l i) = l − i
using assms
proof (induct M arbitrary: i)
  case (Cons m M')
  {
    assume marked m i = l
    hence ?case
      unfolding currentLevel-def
      by simp
  }
  moreover
  {
    assume marked m i < l
    hence ?case
      using Cons(1) [of i+1]
      using Cons(3)
      unfolding currentLevel-def
      by simp
  }
  moreover
  {
    assume ¬ marked m
    hence ?case
      using Cons
      unfolding currentLevel-def
      by simp
  }
  ultimately
  show ?case
    using ‹i <= l›
    by auto
next
  case Nil
  thus ?case
    unfolding currentLevel-def
    by simp
qed

lemma currentLevelPrefixToLevelEq:
```

assumes
  level ≤ *currentLevel M*
shows
  currentLevel (*prefixToLevel level M*) = *level*
using *assms*
unfolding *prefixToLevel-def*
using *currentLevelPrefixToLevelEq-aux*[*of 0 level M*]
by *simp*

lemma *prefixToLevel-auxIncreaseAuxilaryCounter*:
  assumes *k* ≥ *i*
  shows *prefixToLevel-aux M l i* = *prefixToLevel-aux M* (*l* + (*k* − *i*))
k
using *assms*
proof (*induct M arbitrary*: *i k*)
  case (*Cons m M′*)
  {
    assume ¬ *marked m*
    hence *?case*
      using *Cons*(*1*)[*of i k*] *Cons*(*2*)
      by *simp*
  }
  moreover
  {
    assume *i* ≥ *l marked m*
    hence *?case*
      using ‹*k* ≥ *i*›
      by *simp*
  }
  moreover
  {
    assume *i* < *l marked m*
    hence *?case*
      using *Cons*(*1*)[*of i+1 k+1*] *Cons*(*2*)
      by *simp*
  }
  ultimately
  show *?case*
    by (*auto split*: *if-split-asm*)
qed *simp*

lemma *isPrefixPrefixToLevel-auxLowerLevel*:
  assumes *i* ≤ *j*
  shows *isPrefix* (*prefixToLevel-aux M i k*) (*prefixToLevel-aux M j k*)
using *assms*
by (*induct M arbitrary*: *k*) (*auto simp add*:*isPrefix-def*)

lemma *isPrefixPrefixToLevelLowerLevel*:
assumes *level* < *level′*

```
shows isPrefix (prefixToLevel level M) (prefixToLevel level′ M)
using assms
unfolding prefixToLevel-def
using isPrefixPrefixToLevel-auxLowerLevel[of level level′ M 0]
by simp

lemma prefixToLevel-auxPrefixToLevel-auxHigherLevel:
  assumes i ≤ j
  shows prefixToLevel-aux a i k = prefixToLevel-aux (prefixToLevel-aux
a j k) i k
using assms
by (induct a arbitrary: k) auto

lemma prefixToLevelPrefixToLevelHigherLevel:
  assumes level ≤ level′
  shows prefixToLevel level M = prefixToLevel level (prefixToLevel
level′ M)
using assms
unfolding prefixToLevel-def
using prefixToLevel-auxPrefixToLevel-auxHigherLevel[of level level′ M
0]
by simp

lemma prefixToLevelAppend-aux1:
  assumes
  l ≥ i and l − i < currentLevel a
  shows
  prefixToLevel-aux (a @ b) l i = prefixToLevel-aux a l i
using assms
proof (induct a arbitrary: i)
  case (Cons a a′)
  {
    assume ¬ marked a
    hence ?case
      using Cons(1)[of i] ‹i ≤ l› ‹l − i < currentLevel (a # a′)›
      unfolding currentLevel-def
      by simp
  }
  moreover
  {
    assume marked a l = i
    hence ?case
      by simp
  }
  moreover
  {
    assume marked a l > i
    hence ?case
      using Cons(1)[of i + 1] ‹i ≤ l› ‹l − i < currentLevel (a # a′)›
```

```
      unfolding currentLevel-def
      by simp
  }
  ultimately
  show ?case
    using ‹i ≤ l›
    by auto
next
  case Nil
  thus ?case
    unfolding currentLevel-def
    by simp
qed


lemma prefixToLevelAppend-aux2:
  assumes
  i ≤ l and currentLevel a + i ≤ l
  shows prefixToLevel-aux (a @ b) l i = a @ prefixToLevel-aux b l (i
+ (currentLevel a))
using assms
proof (induct a arbitrary: i)
  case (Cons a a')
  {
    assume ¬ marked a
    hence ?case
      using Cons
      unfolding currentLevel-def
      by simp
  }
  moreover
  {
    assume marked a l = i
    hence ?case
      using ‹(currentLevel (a # a')) + i ≤ l›
      unfolding currentLevel-def
      by simp
  }
  moreover
  {
    assume marked a l > i
   hence prefixToLevel-aux (a' @ b) l (i + 1) = a' @ prefixToLevel-aux
b l (i + 1 + currentLevel a')
      using Cons(1) [of i + 1] ‹(currentLevel (a # a')) + i ≤ l›
      unfolding currentLevel-def
      by simp
    moreover
     have i + 1 + length (markedElements a') = i + (1 + length
(markedElements a'))
```

```
      by simp
    ultimately
    have ?case
      using ‹marked a› ‹l > i›
      unfolding currentLevel-def
      by simp
  }
  ultimately
  show ?case
    using ‹l ≥ i›
    by auto
next
  case Nil
  thus ?case
    unfolding currentLevel-def
    by simp
qed

lemma prefixToLevelAppend:
  shows prefixToLevel level (a @ b) =
  (if level < currentLevel a then
      prefixToLevel level a
  else
      a @ prefixToLevel-aux b level (currentLevel a)
  )
proof (cases level < currentLevel a)
  case True
  thus ?thesis
    unfolding prefixToLevel-def
    using prefixToLevelAppend-aux1[of 0 level a]
    by simp
next
  case False
  thus ?thesis
    unfolding prefixToLevel-def
    using prefixToLevelAppend-aux2[of 0 level a]
    by simp
qed

lemma isProperPrefixPrefixToLevel:
  assumes level < currentLevel t
  shows ∃ s. (prefixToLevel level t) @ s = t ∧ s ≠ [] ∧ (marked (hd
s))
proof−
  have isPrefix (prefixToLevel level t) t
    by (simp add:isPrefixPrefixToLevel)
  then obtain s::'a Trail
    where (prefixToLevel level t) @ s = t
    unfolding isPrefix-def
```

by *auto*
moreover
have $s \neq []$
proof−
{
assume $s = []$
with ‹(*prefixToLevel level t*) @ *s* = *t*›
have *prefixToLevel level t* = *t*
by *simp*
hence *currentLevel* (*prefixToLevel level t*) ≤ *level*
using *currentLevelPrefixToLevel*[*of level t*]
by *simp*
with ‹*prefixToLevel level t* = *t*› **have** *currentLevel t* ≤ *level*
by *simp*
with ‹*level* < *currentLevel t*› **have** *False*
by *simp*
}
thus *?thesis*
by *auto*
qed
moreover
have *marked* (*hd s*)
proof−
{
assume ¬ *marked* (*hd s*)
have *currentLevel* (*prefixToLevel level t*) ≤ *level*
by (*simp add*:*currentLevelPrefixToLevel*)
from ‹$s \neq []$› **have** *s* = [*hd s*] @ (*tl s*)
by *simp*
with ‹(*prefixToLevel level t*) @ *s* = *t*› **have**
t = (*prefixToLevel level t*) @ [*hd s*] @ (*tl s*)
by *simp*
hence (*prefixToLevel level t*) = (*prefixToLevel level* ((*prefixToLevel level t*) @ [*hd s*] @ (*tl s*)))
by *simp*
also
with ‹*currentLevel* (*prefixToLevel level t*) ≤ *level*›
have . . . = ((*prefixToLevel level t*) @ (*prefixToLevel-aux* ([*hd s*] @ (*tl s*)) *level* (*currentLevel* (*prefixToLevel level t*))))
by (*auto simp add*: *prefixToLevelAppend*)
also
have . . . =
((*prefixToLevel level t*) @ (*hd s*) # *prefixToLevel-aux* (*tl s*) *level* (*currentLevel* (*prefixToLevel level t*)))
proof−
from ‹*currentLevel* (*prefixToLevel level t*) <= *level*› ‹¬ *marked* (*hd s*)›
have *prefixToLevel-aux* ([*hd s*] @ (*tl s*)) *level* (*currentLevel* (*prefixToLevel level t*)) =

(*hd s*) # *prefixToLevel-aux* (*tl s*) *level* (*currentLevel* (*prefixToLevel level t*))
by *simp*
thus *?thesis*
by *simp*
qed
ultimately
have (*prefixToLevel level t*) = (*prefixToLevel level t*) @ (*hd s*) # *prefixToLevel-aux* (*tl s*) *level* (*currentLevel* (*prefixToLevel level t*))
by *simp*
hence *False*
by *auto*
}
thus *?thesis*
by *auto*
qed
ultimately
show *?thesis*
by *auto*
qed

lemma *prefixToLevelElementsElementLevel*:
assumes
e ∈ *set* (*elements* (*prefixToLevel level M*))
shows
elementLevel e M ≤ *level*
proof −
have *elementLevel e* (*prefixToLevel level M*) ≤ *currentLevel* (*prefixToLevel level M*)
by (*simp add*: *elementLevelLeqCurrentLevel*)
moreover
hence *currentLevel* (*prefixToLevel level M*) ≤ *level*
using *currentLevelPrefixToLevel*[*of level M*]
by *simp*
ultimately have *elementLevel e* (*prefixToLevel level M*) ≤ *level*
by *simp*
moreover
have *isPrefix* (*prefixToLevel level M*) *M*
by (*simp add*:*isPrefixPrefixToLevel*)
then obtain *s*
where (*prefixToLevel level M*) @ *s* = *M*
unfolding *isPrefix-def*
by *auto*
with ‹*e* ∈ *set* (*elements* (*prefixToLevel level M*))›
have *elementLevel e* (*prefixToLevel level M*) = *elementLevel e M*
using *elementLevelAppend* [*of e prefixToLevel level M s*]
by *simp*
ultimately
show *?thesis*

```
    by simp
qed

lemma elementLevelLtLevelImpliesMemberPrefixToLevel-aux:
  assumes
  e ∈ set(elements M) and
  elementLevel e M + i ≤ level and
  i ≤ level
  shows
  e ∈ set (elements (prefixToLevel-aux M level i))
using assms
proof (induct M arbitrary: i)
  case (Cons m M')
  thus ?case
  proof (cases e = element m)
    case True
    thus ?thesis
      using ‹elementLevel e (m # M') + i ≤ level›
      unfolding prefixToLevel-def
      unfolding elementLevel-def
      unfolding markedElementsTo-def
      by (simp split: if-split-asm)
  next
    case False
    with ‹e ∈ set (elements (m # M'))›
    have e ∈ set (elements M')
      by simp

    show ?thesis
    proof (cases marked m)
      case True
      with Cons ‹e ≠ element m›
      have (elementLevel e M') + i + 1 ≤ level
        unfolding elementLevel-def
        unfolding markedElementsTo-def
        by (simp split: if-split-asm)
      moreover
      have elementLevel e M' ≥ 0
        by auto
      ultimately
      have i + 1 ≤ level
        by simp
       with ‹e ∈ set (elements M')› ‹(elementLevel e M') + i + 1 ≤
level› Cons(1)[of i+1]
      have e ∈ set (elements (prefixToLevel-aux M' level (i + 1)))
        by simp
      with ‹e ≠ element m› ‹i + 1 ≤ level› True
      show ?thesis
        by simp
```

```
    next
      case False
       with ‹e ≠ element m› ‹elementLevel e (m # M') + i ≤ level›
have elementLevel e M' + i ≤ level
        unfolding elementLevel-def
        unfolding markedElementsTo-def
        by (simp split: if-split-asm)
    with ‹e ∈ set (elements M')› have e ∈ set (elements (prefixToLevel-aux
M' level i))
        using Cons
        by (auto split: if-split-asm)
      with ‹e ≠ element m› False show ?thesis
        by simp
    qed
  qed
qed simp

lemma elementLevelLtLevelImpliesMemberPrefixToLevel:
  assumes
  e ∈ set (elements M) and
  elementLevel e M ≤ level
  shows
  e ∈ set (elements (prefixToLevel level M))
using assms
using elementLevelLtLevelImpliesMemberPrefixToLevel-aux[of e M 0
level]
unfolding prefixToLevel-def
by simp

lemma literalNotInEarlierLevelsThanItsLevel:
  assumes
  level < elementLevel e M
  shows
  e ∉ set (elements (prefixToLevel level M))
proof−
  {
    assume ¬ ?thesis
    hence level ≥ elementLevel e M
      by (simp add: prefixToLevelElementsElementLevel)
    with ‹level < elementLevel e M›
    have False
      by simp
  }
  thus ?thesis
    by auto
qed

lemma elementLevelPrefixElement:
  assumes e ∈ set (elements (prefixToLevel level M))
```

shows *elementLevel e* (*prefixToLevel level M*) = *elementLevel e M*
using *assms*
proof−
have *isPrefix* (*prefixToLevel level M*) *M*
by (*simp add*: *isPrefixPrefixToLevel*)
then obtain *s* **where** (*prefixToLevel level M*) @ *s* = *M*
unfolding *isPrefix-def*
by *auto*
with *assms* **show** *?thesis*
using *elementLevelAppend*[*of e prefixToLevel level M s*]
by *auto*
qed

lemma *currentLevelZeroTrailEqualsItsPrefixToLevelZero*:
assumes *currentLevel M* = *0*
shows *M* = *prefixToLevel 0 M*
using *assms*
proof (*induct M*)
case (*Cons a M'*)
show *?case*
proof−
from *Cons*
have *currentLevel M'* = *0* **and** *markedElements M'* = [] **and** ¬
marked a
unfolding *currentLevel-def*
by (*auto split*: *if-split-asm*)
thus *?thesis*
using *Cons*
unfolding *prefixToLevel-def*
by *auto*
qed
next
case *Nil*
thus *?case*
unfolding *currentLevel-def*
unfolding *prefixToLevel-def*
by *simp*
qed

3.9 Number of literals of every trail level

primrec
levelsCounter-aux :: *'a Trail* ⇒ *nat list* ⇒ *nat list*
where
levelsCounter-aux [] *l* = *l*
| *levelsCounter-aux* (*h* # *t*) *l* =
(*if* (*marked h*) *then*
levelsCounter-aux t (*l* @ [*1*])
else

levelsCounter-aux t (*butlast l* @ [*Suc* (*last l*)])
)

definition
levelsCounter :: *'a Trail* ⇒ *nat list*
where
levelsCounter t = *levelsCounter-aux t* [*0*]

lemma *levelsCounter-aux-startIrellevant*:
∀ *y*. *y* ≠ [] ⟶ *levelsCounter-aux a* (*x* @ *y*) = (*x* @ *levelsCounter-aux a y*)
by (*induct a*) (*auto simp add*: *butlastAppend*)

lemma *levelsCounter-auxSuffixContinues*: ∀ *l*. *levelsCounter-aux* (*a* @ *b*) *l* = *levelsCounter-aux b* (*levelsCounter-aux a l*)
by (*induct a*) *auto*

lemma *levelsCounter-auxNotEmpty*: ∀ *l*. *l* ≠ [] ⟶ *levelsCounter-aux a l* ≠ []
by (*induct a*) *auto*

lemma *levelsCounter-auxIncreasesFirst*:
∀ *m n l1 l2*. *levelsCounter-aux a* (*m* # *l1*) = *n* # *l2* ⟶ *m* <= *n*
proof (*induct a*)
  case *Nil*
  {
    fix *m*::*nat* **and** *n*::*nat* **and** *l1*::*nat list* **and** *l2*::*nat list*
    assume *levelsCounter-aux* [] (*m* # *l1*) = *n* # *l2*
    hence *m* = *n*
      by *simp*
  }
  thus *?case*
    by *simp*
next
  case (*Cons a list*)
  {
    fix *m*::*nat* **and** *n*::*nat* **and** *l1*::*nat list* **and** *l2*::*nat list*
    assume *levelsCounter-aux* (*a* # *list*) (*m* # *l1*) = *n* # *l2*
    have *m* <= *n*
    proof (*cases marked a*)
      case *True*
      with ‹*levelsCounter-aux* (*a* # *list*) (*m* # *l1*) = *n* # *l2*›
      have *levelsCounter-aux list* (*m* # *l1* @ [*Suc 0*]) = *n* # *l2*
        by *simp*
      with *Cons*
      show *?thesis*
        by *auto*
    next

```
    case False
    show ?thesis
    proof (cases l1 = [])
      case True
       with ‹¬ marked a› ‹levelsCounter-aux (a # list) (m # l1) =
n # l2›
      have levelsCounter-aux list [Suc m] = n # l2
        by simp
      with Cons
      have Suc m <= n
        by auto
      thus ?thesis
        by simp
    next
      case False
       with ‹¬ marked a› ‹levelsCounter-aux (a # list) (m # l1) =
n # l2›
       have levelsCounter-aux list (m # butlast l1 @ [Suc (last l1)])
= n # l2
        by simp
      with Cons
      show ?thesis
        by auto
    qed
  qed
 }
 thus ?case
   by simp
qed

lemma levelsCounterPrefix:
 assumes (isPrefix p a)
 shows ? rest. rest ≠ [] ∧ levelsCounter a = butlast (levelsCounter
p) @ rest ∧ last (levelsCounter p) ≤ hd rest
proof−
 from assms
 obtain s :: 'a Trail where p @ s = a
   unfolding isPrefix-def
   by auto
 from ‹p @ s = a› have levelsCounter a = levelsCounter (p @ s)
   by simp
 show ?thesis
 proof (cases s = [])
   case True
  have (levelsCounter a) = (butlast (levelsCounter p)) @ [last (levelsCounter
p)] ∧
     (last (levelsCounter p)) <= hd [last (levelsCounter p)]
     using ‹p @ s = a› ‹s = []›
     unfolding levelsCounter-def
```

```
      using levelsCounter-auxNotEmpty[of p]
      by auto
    thus ?thesis
      by auto
  next
    case False
    show ?thesis
    proof (cases marked (hd s))
      case True
      from ‹p @ s = a› have levelsCounter a = levelsCounter (p @ s)
        by simp
      also
      have ... = levelsCounter-aux s (levelsCounter-aux p [0])
        unfolding levelsCounter-def
        by (simp add: levelsCounter-auxSuffixContinues)
      also
      have ... = levelsCounter-aux (tl s) ((levelsCounter-aux p [0]) @
[1])
      proof−
        from ‹s ≠ []› have s = hd s # tl s
          by simp
          then have levelsCounter-aux s (levelsCounter-aux p [0]) =
levelsCounter-aux (hd s # tl s) (levelsCounter-aux p [0])
          by simp
        with ‹marked (hd s)› show ?thesis
          by simp
      qed
      also
       have ... = levelsCounter-aux p [0] @ (levelsCounter-aux (tl s)
[1])
        by (simp add: levelsCounter-aux-startIrellevant)
      finally
      have levelsCounter a = levelsCounter p @ (levelsCounter-aux (tl
s) [1])
        unfolding levelsCounter-def
        by simp
       hence (levelsCounter a) = (butlast (levelsCounter p)) @ ([last
(levelsCounter p)] @ (levelsCounter-aux (tl s) [1])) ∧
          (last (levelsCounter p)) <= hd ([last (levelsCounter p)] @
(levelsCounter-aux (tl s) [1]))
        unfolding levelsCounter-def
        using levelsCounter-auxNotEmpty[of p]
        by auto
      thus ?thesis
        by auto
    next
      case False
      from ‹p @ s = a› have levelsCounter a = levelsCounter (p @ s)
        by simp
```

also
have ... = *levelsCounter-aux s* (*levelsCounter-aux p* [*0*])
unfolding *levelsCounter-def*
by (*simp add*: *levelsCounter-auxSuffixContinues*)
also
have ... = *levelsCounter-aux* (*tl s*) ((*butlast* (*levelsCounter-aux p* [*0*])) @ [*Suc* (*last* (*levelsCounter-aux p* [*0*]))])
proof−
from ‹*s* ≠ []› **have** *s* = *hd s* # *tl s*
by *simp*
then have *levelsCounter-aux s* (*levelsCounter-aux p* [*0*]) = *levelsCounter-aux* (*hd s* # *tl s*) (*levelsCounter-aux p* [*0*])
by *simp*
with ‹~*marked* (*hd s*)› **show** *?thesis*
by *simp*
qed
also
have ... = *butlast* (*levelsCounter-aux p* [*0*]) @ (*levelsCounter-aux* (*tl s*) [*Suc* (*last* (*levelsCounter-aux p* [*0*]))])
by (*simp add*: *levelsCounter-aux-startIrellevant*)
finally
have *levelsCounter a* = *butlast* (*levelsCounter-aux p* [*0*]) @ (*levelsCounter-aux* (*tl s*) [*Suc* (*last* (*levelsCounter-aux p* [*0*]))])
unfolding *levelsCounter-def*
by *simp*
moreover
have *hd* (*levelsCounter-aux* (*tl s*) [*Suc* (*last* (*levelsCounter-aux p* [*0*]))]) >= *Suc* (*last* (*levelsCounter-aux p* [*0*]))
proof−
have (*levelsCounter-aux* (*tl s*) [*Suc* (*last* (*levelsCounter-aux p* [*0*]))]) ≠ []
using *levelsCounter-auxNotEmpty*[*of tl s*]
by *simp*
then obtain *h t* **where** (*levelsCounter-aux* (*tl s*) [*Suc* (*last* (*levelsCounter-aux p* [*0*]))]) = *h* # *t*
using *neq-Nil-conv*[*of* (*levelsCounter-aux* (*tl s*) [*Suc* (*last* (*levelsCounter-aux p* [*0*]))])]
by *auto*
hence *h* ≥ *Suc* (*last* (*levelsCounter-aux p* [*0*]))
using *levelsCounter-auxIncreasesFirst*[*of tl s*]
by *auto*
with ‹(*levelsCounter-aux* (*tl s*) [*Suc* (*last* (*levelsCounter-aux p* [*0*]))]) = *h* # *t*›
show *?thesis*
by *simp*
qed
ultimately
have *levelsCounter a* = *butlast* (*levelsCounter p*) @ (*levelsCounter-aux* (*tl s*) [*Suc* (*last* (*levelsCounter-aux p* [*0*]))]) ∧

last (*levelsCounter p*) ≤ *hd* (*levelsCounter-aux* (*tl s*) [*Suc* (*last* (*levelsCounter-aux p* [*0*]))])
unfolding *levelsCounter-def*
by *simp*
thus *?thesis*
using *levelsCounter-auxNotEmpty*[*of tl s*]
by *auto*
qed
qed
qed

lemma *levelsCounterPrefixToLevel*:
assumes *p* = *prefixToLevel level a level* ≥ *0 level* < *currentLevel a*
shows *? rest . rest* ≠ [] ∧ (*levelsCounter a*) = (*levelsCounter p*) @ *rest*
proof−
from *assms*
obtain *s* :: ′*a Trail* **where** *p* @ *s* = *a s* ≠ [] *marked* (*hd s*)
using *isProperPrefixPrefixToLevel*[*of level a*]
by *auto*
from ‹*p* @ *s* = *a*› **have** *levelsCounter a* = *levelsCounter* (*p* @ *s*)
by *simp*
also
have . . . = *levelsCounter-aux s* (*levelsCounter-aux p* [*0*])
unfolding *levelsCounter-def*
by (*simp add*: *levelsCounter-auxSuffixContinues*)
also
have . . . = *levelsCounter-aux* (*tl s*) ((*levelsCounter-aux p* [*0*]) @ [*1*])
proof−
from ‹*s* ≠ []› **have** *s* = *hd s* # *tl s*
by *simp*
then have *levelsCounter-aux s* (*levelsCounter-aux p* [*0*]) = *levelsCounter-aux* (*hd s* # *tl s*) (*levelsCounter-aux p* [*0*])
by *simp*
with ‹*marked* (*hd s*)› **show** *?thesis*
by *simp*
qed
also
have . . . = *levelsCounter-aux p* [*0*] @ (*levelsCounter-aux* (*tl s*) [*1*])
by (*simp add*: *levelsCounter-aux-startIrellevant*)
finally
have *levelsCounter a* = *levelsCounter p* @ (*levelsCounter-aux* (*tl s*) [*1*])
unfolding *levelsCounter-def*
by *simp*
moreover
have *levelsCounter-aux* (*tl s*) [*1*] ≠ []
by (*simp add*: *levelsCounter-auxNotEmpty*)
ultimately

show *?thesis*
by *simp*
qed

3.10 Prefix before last marked element

primrec
prefixBeforeLastMarked :: *'a Trail* ⇒ *'a Trail*
where
prefixBeforeLastMarked [] = []
| *prefixBeforeLastMarked* (*h*#*t*) = (*if* (*marked h*) ∧ (*markedElements t*) = [] *then* [] *else* (*h*#(*prefixBeforeLastMarked t*)))

lemma *prefixBeforeLastMarkedIsPrefixBeforeLastLevel*:
assumes *markedElements M* ≠ []
shows *prefixBeforeLastMarked M* = *prefixToLevel* ((*currentLevel M*) − *1*) *M*
using *assms*
proof (*induct M*)
case *Nil*
thus *?case*
by *simp*
next
case (*Cons a M'*)
thus *?case*
proof (*cases marked a*)
case *True*
hence *currentLevel* (*a* # *M'*) ≥ *1*
unfolding *currentLevel-def*
by *simp*
with *True Cons* **show** *?thesis*
using *prefixToLevel-auxIncreaseAuxilaryCounter*[*of 0 1 M' currentLevel M'* − *1*]
unfolding *prefixToLevel-def*
unfolding *currentLevel-def*
by *auto*
next
case *False*
with *Cons* **show** *?thesis*
unfolding *prefixToLevel-def*
unfolding *currentLevel-def*
by *auto*
qed
qed

lemma *isPrefixPrefixBeforeLastMarked*:
shows *isPrefix* (*prefixBeforeLastMarked M*) *M*
unfolding *isPrefix-def*
by (*induct M*) *auto*

lemma *lastMarkedNotInPrefixBeforeLastMarked*:
assumes *uniq* (*elements M*) **and** *markedElements M* $\neq$ []
shows $\neg$ (*lastMarked M*) $\in$ *set* (*elements* (*prefixBeforeLastMarked M*))
using *assms*
unfolding *lastMarked-def*
by (*induct M*) (*auto split*: *if-split-asm simp add*: *markedElementsAreElements*)

lemma *uniqImpliesPrefixBeforeLastMarkedIsPrefixBeforeLastMarked*:
assumes *markedElements M* $\neq$ [] **and** (*lastMarked M*) $\notin$ *set* (*elements M*)
shows *prefixBeforeLastMarked M* = *prefixBeforeElement* (*lastMarked M*) *M*
using *assms*
unfolding *lastMarked-def*
proof (*induct M*)
case *Nil*
thus *?case*
by *auto*
next
case (*Cons a M'*)
show *?case*
proof (*cases marked a* $\wedge$ (*markedElements M'*) = [])
case *True*
thus *?thesis*
unfolding *lastMarked-def*
by *auto*
next
case *False*
hence *last* (*markedElements* (*a* # *M'*)) = *last* (*markedElements M'*)
by *auto*
thus *?thesis*
using *Cons*
by (*auto split*: *if-split-asm simp add*: *markedElementsAreElements*)
qed
qed

lemma *markedElementsAreElementsBeforeLastDecisionAndLastDecision*:
assumes *markedElements M* $\neq$ []
shows (*markedElements M*) = (*markedElements* (*prefixBeforeLastMarked M*)) @ [*lastMarked M*]
using *assms*
unfolding *lastMarked-def*
by (*induct M*) (*auto split*: *if-split-asm*)

end

4 Verification of DPLL based SAT solvers.

theory *SatSolverVerification*
imports *CNF Trail*
begin

This theory contains a number of lemmas used in verification of different SAT solvers. Although this file does not contain any theorems significant on their own, it is an essential part of the SAT solver correctness proof because it contains most of the technical details used in the proofs that follow. These lemmas serve as a basis for partial correctness proof for pseudo-code implementation of modern SAT solvers described in [2], in terms of Hoare logic.

4.1 Literal Trail

LiteralTrail is a Trail consisting of literals, where decision literals are marked.

type-synonym *LiteralTrail* = *Literal Trail*

abbreviation *isDecision* :: $('a \times bool) \Rightarrow bool$
where *isDecision l* == *marked l*

abbreviation *lastDecision* :: *LiteralTrail* ⇒ *Literal*
where *lastDecision M* == *Trail.lastMarked M*

abbreviation *decisions* :: *LiteralTrail* ⇒ *Literal list*
where *decisions M* == *Trail.markedElements M*

abbreviation *decisionsTo* :: *Literal* ⇒ *LiteralTrail* ⇒ *Literal list*
where *decisionsTo M l* == *Trail.markedElementsTo M l*

abbreviation *prefixBeforeLastDecision* :: *LiteralTrail* ⇒ *LiteralTrail*
where *prefixBeforeLastDecision M* == *Trail.prefixBeforeLastMarked M*

4.2 Invariants

In this section a number of conditions will be formulated and it will be shown that these conditions are invariant after applying different DPLL-based transition rules.

definition

InvariantConsistent (*M*::*LiteralTrail*) == *consistent* (*elements M*)

definition
InvariantUniq (*M*::*LiteralTrail*) == *uniq* (*elements M*)

definition
InvariantImpliedLiterals (*F*::*Formula*) (*M*::*LiteralTrail*) == ∀ *l*. *l el elements M* ⟶ *formulaEntailsLiteral* (*F* @ *val2form* (*decisionsTo l M*)) *l*

definition
InvariantEquivalent (*F0*::*Formula*) (*F*::*Formula*) == *equivalentFormulae F0 F*

definition
InvariantVarsM (*M*::*LiteralTrail*) (*F0*::*Formula*) (*Vbl*::*Variable set*) == *vars* (*elements M*) ⊆ *vars F0* ∪ *Vbl*

definition
InvariantVarsF (*F*::*Formula*) (*F0*::*Formula*) (*Vbl*::*Variable set*) == *vars F* ⊆ *vars F0* ∪ *Vbl*

The following invariants are used in conflict analysis.

definition
InvariantCFalse (*conflictFlag*::*bool*) (*M*::*LiteralTrail*) (*C*::*Clause*) == *conflictFlag* ⟶ *clauseFalse C* (*elements M*)

definition
InvariantCEntailed (*conflictFlag*::*bool*) (*F*::*Formula*) (*C*::*Clause*) == *conflictFlag* ⟶ *formulaEntailsClause F C*

definition
InvariantReasonClauses (*F*::*Formula*) (*M*::*LiteralTrail*) ==
∀ *literal*. *literal el* (*elements M*) ∧ ¬ *literal el decisions M* ⟶
(∃ *clause*. *formulaEntailsClause F clause* ∧ *isReason clause literal* (*elements M*))

4.2.1 Auxiliary lemmas

This section contains some auxiliary lemmas that additionally characterize some of invariants that have been defined.

Lemmas about *InvariantImpliedLiterals*.

lemma *InvariantImpliedLiteralsWeakerVariant*:
fixes *M* :: *LiteralTrail* **and** *F* :: *Formula*
assumes ∀ *l*. *l el elements M* ⟶ *formulaEntailsLiteral* (*F* @ *val2form* (*decisionsTo l M*)) *l*
shows ∀ *l*. *l el elements M* ⟶ *formulaEntailsLiteral* (*F* @ *val2form* (*decisions M*)) *l*

proof −
{
fix *l* :: *Literal*
assume *l el elements M*
with *assms*
have *formulaEntailsLiteral* (*F* @ *val2form* (*decisionsTo l M*)) *l*
by *simp*
have *isPrefix* (*decisionsTo l M*) (*decisions M*)
by (*simp add*: *markedElementsToArePrefixOfMarkedElements*)
then obtain *s* :: *Valuation*
where (*decisionsTo l M*) @ *s* = (*decisions M*)
using *isPrefix-def* [*of decisionsTo l M decisions M*]
by *auto*
hence (*decisions M*) = (*decisionsTo l M*) @ *s*
by (*rule sym*)
with ‹*formulaEntailsLiteral* (*F* @ *val2form* (*decisionsTo l M*)) *l*›
have *formulaEntailsLiteral* (*F* @ *val2form* (*decisions M*)) *l*
using *formulaEntailsLiteralAppend* [*of F* @ *val2form* (*decisionsTo l M*) *l val2form s*]
by (*auto simp add*:*formulaEntailsLiteralAppend val2formAppend*)
}
thus *?thesis*
by *simp*
qed

lemma *InvariantImpliedLiteralsAndElementsEntailLiteralThenDecisionsEntailLiteral*:
fixes *M* :: *LiteralTrail* **and** *F* :: *Formula* **and** *literal* :: *Literal*
assumes *InvariantImpliedLiterals F M* **and** *formulaEntailsLiteral* (*F* @ (*val2form* (*elements M*))) *literal*
shows *formulaEntailsLiteral* (*F* @ *val2form* (*decisions M*)) *literal*
proof −
{
fix *valuation* :: *Valuation*
assume *model valuation* (*F* @ *val2form* (*decisions M*))
hence *formulaTrue F valuation* **and** *formulaTrue* (*val2form* (*decisions M*)) *valuation* **and** *consistent valuation*
by (*auto simp add*: *formulaTrueAppend*)
{
fix *l* :: *Literal*
assume *l el* (*elements M*)
from ‹*InvariantImpliedLiterals F M*›
have ∀ *l*. *l el* (*elements M*) ⟶ *formulaEntailsLiteral* (*F* @ *val2form* (*decisions M*)) *l*
by (*simp add*: *InvariantImpliedLiteralsWeakerVariant InvariantImpliedLiterals-def*)
with ‹*l el* (*elements M*)›
have *formulaEntailsLiteral* (*F* @ *val2form* (*decisions M*)) *l*
by *simp*

```
      with ‹model valuation (F @ val2form (decisions M))›
      have literalTrue l valuation
        by (simp add: formulaEntailsLiteral-def)
    }
    hence formulaTrue (val2form (elements M)) valuation
      by (simp add: val2formFormulaTrue)
    with ‹formulaTrue F valuation› ‹consistent valuation›
    have model valuation (F @ (val2form (elements M)))
      by (auto simp add:formulaTrueAppend)
   with ‹formulaEntailsLiteral (F @ (val2form (elements M))) literal›
    have literalTrue literal valuation
      by (simp add: formulaEntailsLiteral-def)
  }
  thus ?thesis
    by (simp add: formulaEntailsLiteral-def)
qed

lemma InvariantImpliedLiteralsAndFormulaFalseThenFormulaAndDecisionsAreNotSatisfiable:
  fixes M :: LiteralTrail and F :: Formula
  assumes InvariantImpliedLiterals F M and formulaFalse F (elements M)
  shows ¬ satisfiable (F @ val2form (decisions M))
proof -
  from ‹formulaFalse F (elements M)›
  have formulaFalse (F @ val2form (decisions M)) (elements M)
    by (simp add: formulaFalseAppend)
  moreover
  from ‹InvariantImpliedLiterals F M›
  have formulaEntailsValuation (F @ val2form (decisions M)) (elements M)
    unfolding formulaEntailsValuation-def
    unfolding InvariantImpliedLiterals-def
    using InvariantImpliedLiteralsWeakerVariant[of M F]
    by simp
  ultimately
  show ?thesis
   using formulaFalseInEntailedValuationIsUnsatisfiable [of F @ val2form (decisions M) elements M]
    by simp
qed

lemma InvariantImpliedLiteralsHoldsForPrefix:
  fixes M :: LiteralTrail and prefix :: LiteralTrail and F :: Formula
  assumes InvariantImpliedLiterals F M  and isPrefix prefix M
  shows InvariantImpliedLiterals F prefix
proof -
  {
    fix l :: Literal
```

assume ∗: *l el elements prefix*

from ∗ ‹*isPrefix prefix M*›
have *l el elements M*
unfolding *isPrefix-def*
by *auto*

from ∗ **and** ‹*isPrefix prefix M*›
have *decisionsTo l prefix = decisionsTo l M*
using *markedElementsToPrefixElement* [*of prefix M l*]
by *simp*

from ‹*InvariantImpliedLiterals F M*› **and** ‹*l el elements M*›
have *formulaEntailsLiteral* (*F* @ *val2form* (*decisionsTo l M*)) *l*
by (*simp add*:*InvariantImpliedLiterals-def*)
with ‹*decisionsTo l prefix = decisionsTo l M*›
have *formulaEntailsLiteral* (*F* @ *val2form* (*decisionsTo l prefix*)) *l*
by *simp*
} **thus** *?thesis*
by (*auto simp add*: *InvariantImpliedLiterals-def*)
qed

Lemmas about *InvariantReasonClauses*.

lemma *InvariantReasonClausesHoldsForPrefix*:
fixes *F*::*Formula* **and** *M*::*LiteralTrail* **and** *p*::*LiteralTrail*
assumes *InvariantReasonClauses F M* **and** *InvariantUniq M* **and** *isPrefix p M*
shows *InvariantReasonClauses F p*
proof−
from ‹*InvariantReasonClauses F M*›
have ∗: ∀ *literal. literal el elements M* ∧ ¬ *literal el decisions M* ⟶

(∃ *clause. formulaEntailsClause F clause* ∧ *isReason clause literal* (*elements M*))
unfolding *InvariantReasonClauses-def*
by *simp*
from ‹*InvariantUniq M*›
have *uniq* (*elements M*)
unfolding *InvariantUniq-def*
by *simp*
{
fix *literal*::*Literal*
assume *literal el elements p* **and** ¬ *literal el decisions p*
from ‹*isPrefix p M*› ‹*literal el* (*elements p*)›
have *literal el* (*elements M*)
by (*auto simp add*: *isPrefix-def*)
moreover
from ‹*isPrefix p M*› ‹*literal el* (*elements p*)› ‹¬ *literal el* (*decisions p*)› ‹*uniq* (*elements M*)›

have ¬ *literal el decisions M*
using *markedElementsTrailMemPrefixAreMarkedElementsPrefix* [*of M p literal*]
by *auto*
ultimately
obtain *clause*::*Clause* **where**
formulaEntailsClause F clause isReason clause literal (*elements M*)
using ∗
by *auto*
with ‹*literal el elements p*› ‹¬ *literal el decisions p*› ‹*isPrefix p M*›
have *isReason clause literal* (*elements p*)
using *isReasonHoldsInPrefix*[*of literal elements p elements M clause*]
by (*simp add*:*isPrefixElements*)
with ‹*formulaEntailsClause F clause*›
have ∃ *clause*. *formulaEntailsClause F clause* ∧ *isReason clause literal* (*elements p*)
by *auto*
}
thus *?thesis*
unfolding *InvariantReasonClauses-def*
by *auto*
qed

lemma *InvariantReasonClausesHoldsForPrefixElements*:
fixes *F*::*Formula* **and** *M*::*LiteralTrail* **and** *p*::*LiteralTrail*
assumes *InvariantReasonClauses F p* **and**
isPrefix p M **and**
literal el (*elements p*) **and** ¬ *literal el decisions M*
shows ∃ *clause*. *formulaEntailsClause F clause* ∧ *isReason clause literal* (*elements M*)
proof −
from ‹*isPrefix p M*› ‹¬ *literal el* (*decisions M*)›
have ¬ *literal el* (*decisions p*)
using *markedElementsPrefixAreMarkedElementsTrail*[*of p M literal*]
by *auto*

from ‹*InvariantReasonClauses F p*› ‹*literal el* (*elements p*)› ‹¬ *literal el* (*decisions p*)› **obtain** *clause* :: *Clause*
where *formulaEntailsClause F clause isReason clause literal* (*elements p*)
unfolding *InvariantReasonClauses-def*
by *auto*
with ‹*isPrefix p M*›
have *isReason clause literal* (*elements M*)
using *isReasonAppend* [*of clause literal elements p*]

by (*auto simp add*: *isPrefix-def*)
with ‹*formulaEntailsClause F clause*›
show *?thesis*
by *auto*
qed

4.2.2 Transition rules preserve invariants

In this section it will be proved that the different DPLL-based transition rules preserves given invariants. Rules are implicitly given in their most general form. Explicit definition of transition rules will be done in theories that describe specific solvers.

Decide transition rule.

lemma *InvariantUniqAfterDecide*:
fixes *M* :: *LiteralTrail* **and** *literal* :: *Literal* **and** *M′* :: *LiteralTrail*
assumes *InvariantUniq M* **and**
var literal ∉ *vars* (*elements M*) **and**
M′ = *M* @ [(*literal*, *True*)]
shows *InvariantUniq M′*
proof −
from ‹*InvariantUniq M*›
have *uniq* (*elements M*)
unfolding *InvariantUniq-def*
.
{
assume ¬ *uniq* (*elements M′*)
with ‹*uniq* (*elements M*)› ‹*M′* = *M* @ [(*literal*, *True*)]›
have *literal el* (*elements M*)
using *uniqButlastNotUniqListImpliesLastMemButlast* [*of elements M′*]
by *auto*
hence *var literal* ∈ *vars* (*elements M*)
using *valuationContainsItsLiteralsVariable* [*of literal elements M*]
by *simp*
with ‹*var literal* ∉ *vars* (*elements M*)›
have *False*
by *simp*
}
thus *?thesis*
unfolding *InvariantUniq-def*
by *auto*
qed

lemma *InvariantImpliedLiteralsAfterDecide*:
fixes *F* :: *Formula* **and** *M* :: *LiteralTrail* **and** *literal* :: *Literal* **and** *M′* :: *LiteralTrail*
assumes *InvariantImpliedLiterals F M* **and**
var literal ∉ *vars* (*elements M*) **and**

$M' = M$ @ [(*literal*, *True*)]
shows *InvariantImpliedLiterals F* M'
proof −
{
fix *l* :: *Literal*
assume *l el elements* M'
have *formulaEntailsLiteral* (*F* @ *val2form* (*decisionsTo l* M')) *l*
proof (*cases l el elements M*)
case *True*
with ‹$M' = M$ @ [(*literal*, *True*)]›
have *decisionsTo l* M' = *decisionsTo l M*
by (*simp add*: *markedElementsToAppend*)
with ‹*InvariantImpliedLiterals F M*› ‹*l el elements M*›
show *?thesis*
by (*simp add*: *InvariantImpliedLiterals-def*)
next
case *False*
with ‹*l el elements* M'› **and** ‹$M' = M$ @ [(*literal*, *True*)]›
have *l* = *literal*
by (*auto split*: *if-split-asm*)
have *clauseEntailsLiteral* [*literal*] *literal*
by (*simp add*: *clauseEntailsLiteral-def*)
moreover
have [*literal*] *el* (*F* @ *val2form* (*decisions M*) @ [[*literal*]])
by *simp*
moreover
{
have *isDecision* (*last* (*M* @ [(*literal*, *True*)]))
by *simp*
moreover
from ‹*var literal* $\notin$ *vars* (*elements M*)›
have ¬ *literal el* (*elements M*)
using *valuationContainsItsLiteralsVariable*[*of literal elements* *M*]
by *auto*
ultimately
have *decisionsTo literal* (*M* @ [(*literal*, *True*)]) = ((*decisions* *M*) @ [*literal*])
using *lastTrailElementMarkedImpliesMarkedElementsTo-LastElementAreAllMarkedElements* [*of M* @ [(*literal*, *True*)]]
by (*simp add*:*markedElementsAppend*)
}
ultimately
show *?thesis*
using ‹$M' = M$ @ [(*literal*, *True*)]› ‹*l* = *literal*›
clauseEntailsLiteralThenFormulaEntailsLiteral [*of* [*literal*] *F* @ *val2form* (*decisions M*) @ [[*literal*]] *literal*]
by (*simp add*:*val2formAppend*)
qed

}
thus *?thesis*
by (*simp add*:*InvariantImpliedLiterals-def*)
qed

lemma *InvariantVarsMAfterDecide*:
fixes *F* :: *Formula* **and** *F0* :: *Formula* **and** *M* :: *LiteralTrail* **and** *literal* :: *Literal* **and** *M′* :: *LiteralTrail*
assumes *InvariantVarsM M F0 Vbl* **and**
var literal ∈ *Vbl* **and**
M′ = *M* @ [(*literal*, *True*)]
shows *InvariantVarsM M′ F0 Vbl*
proof −
from ‹*InvariantVarsM M F0 Vbl*›
have *vars* (*elements M*) ⊆ *vars F0* ∪ *Vbl*
by (*simp only*:*InvariantVarsM-def*)
from ‹*M′* = *M* @ [(*literal*, *True*)]›
have *vars* (*elements M′*) = *vars* (*elements* (*M* @ [(*literal*, *True*)]))
by *simp*
also have ... = *vars* (*elements M* @ [*literal*])
by *simp*
also have ... = *vars* (*elements M*) ∪ *vars* [*literal*]
using *varsAppendClauses* [*of elements M* [*literal*]]
by *simp*
finally
show *?thesis*
using ‹*vars* (*elements M*) ⊆ (*vars F0*) ∪ *Vbl*› ‹*var literal* ∈ *Vbl*›
unfolding *InvariantVarsM-def*
by *auto*
qed

lemma *InvariantConsistentAfterDecide*:
fixes *M* :: *LiteralTrail* **and** *literal* :: *Literal* **and** *M′* :: *LiteralTrail*
assumes *InvariantConsistent M* **and**
var literal ∉ *vars* (*elements M*) **and**
M′ = *M* @ [(*literal*, *True*)]
shows *InvariantConsistent M′*
proof −
from ‹*InvariantConsistent M*›
have *consistent* (*elements M*)
unfolding *InvariantConsistent-def*
.
{
assume *inconsistent* (*elements M′*)
with ‹*M′* = *M* @ [(*literal*, *True*)]›
have *inconsistent* (*elements M*) ∨ *inconsistent* [*literal*] ∨ (∃ *l*. *literalTrue l* (*elements M*) ∧ *literalFalse l* [*literal*])
using *inconsistentAppend* [*of elements M* [*literal*]]
by *simp*

with ‹*consistent* (*elements M*)› **obtain** *l* :: *Literal*
where *literalTrue l* (*elements M*) **and** *literalFalse l* [*literal*]
by *auto*
hence (*opposite l*) = *literal*
by *auto*
hence *var literal* = *var l*
by *auto*
with ‹*literalTrue l* (*elements M*)›
have *var l* ∈ *vars* (*elements M*)
using *valuationContainsItsLiteralsVariable* [*of l elements M*]
by *simp*
with ‹*var literal* = *var l*› ‹*var literal* ∉ *vars* (*elements M*)›
have *False*
by *simp*
}
thus *?thesis*
unfolding *InvariantConsistent-def*
by *auto*
qed

lemma *InvariantReasonClausesAfterDecide*:
fixes *F* :: *Formula* **and** *M* :: *LiteralTrail* **and** *M′* :: *LiteralTrail*
assumes *InvariantReasonClauses F M* **and** *InvariantUniq M* **and**
M′ = *M* @ [(*literal*, *True*)]
shows *InvariantReasonClauses F M′*
proof −
{
fix *literal′* :: *Literal*
assume *literal′ el elements M′* **and** ¬ *literal′ el decisions M′*

have ∃ *clause*. *formulaEntailsClause F clause* ∧ *isReason clause literal′* (*elements M′*)
proof (*cases literal′ el elements M*)
case *True*
with *assms* ‹¬ *literal′ el decisions M′*› **obtain** *clause*::*Clause*
where *formulaEntailsClause F clause* ∧ *isReason clause literal′* (*elements M′*)
using *InvariantReasonClausesHoldsForPrefixElements* [*of F M M′ literal′*]
by (*auto simp add*:*isPrefix-def*)
thus *?thesis*
by *auto*
next
case *False*
with ‹*M′* = *M* @ [(*literal*, *True*)]› ‹*literal′ el elements M′*›
have *literal* = *literal′*
by (*simp split*: *if-split-asm*)
with ‹*M′* = *M* @ [(*literal*, *True*)]›
have *literal′ el decisions M′*

using *markedElementIsMarkedTrue*[*of literal M'*]
by *simp*
with ‹¬ *literal' el decisions M'*›
have *False*
by *simp*
thus *?thesis*
by *simp*
qed
}
thus *?thesis*
unfolding *InvariantReasonClauses-def*
by *auto*
qed

lemma *InvariantCFalseAfterDecide*:
fixes *conflictFlag*::*bool* **and** *M*::*LiteralTrail* **and** *C*::*Clause*
assumes *InvariantCFalse conflictFlag M C* **and** *M'* = *M* @ [(*literal*, *True*)]
shows *InvariantCFalse conflictFlag M' C*
unfolding *InvariantCFalse-def*
proof
assume *conflictFlag*
show *clauseFalse C* (*elements M'*)
proof −
from ‹*InvariantCFalse conflictFlag M C*›
have *conflictFlag* ⟶ *clauseFalse C* (*elements M*)
unfolding *InvariantCFalse-def*
.
with ‹*conflictFlag*›
have *clauseFalse C* (*elements M*)
by *simp*
with ‹*M'* = *M* @ [(*literal*, *True*)]›
show *?thesis*
by (*simp add*:*clauseFalseAppendValuation*)
qed
qed

UnitPropagate transition rule.

lemma *InvariantImpliedLiteralsHoldsForUnitLiteral*:
fixes *M* :: *LiteralTrail* **and** *F* :: *Formula* **and** *uClause* :: *Clause* **and** *uLiteral* :: *Literal*
assumes *InvariantImpliedLiterals F M* **and**
formulaEntailsClause F uClause **and** *isUnitClause uClause uLiteral* (*elements M*) **and**
M' = *M* @ [(*uLiteral*, *False*)]
shows *formulaEntailsLiteral* (*F* @ *val2form* (*decisionsTo uLiteral M'*)) *uLiteral*
proof−
have *decisionsTo uLiteral M'* = *decisions M*

proof −
from ‹*isUnitClause uClause uLiteral* (*elements M*)›
have ¬ *uLiteral el* (*elements M*)
by (*simp add*: *isUnitClause-def*)
with ‹*M′* = *M* @ [(*uLiteral*, *False*)]›
show *?thesis*
using *markedElementsToAppend*[*of uLiteral M* [(*uLiteral*, *False*)]]
unfolding *markedElementsTo-def*
by *simp*
qed
moreover
from ‹*formulaEntailsClause F uClause*› ‹*isUnitClause uClause uLiteral* (*elements M*)›
have *formulaEntailsLiteral* (*F* @ *val2form* (*elements M*)) *uLiteral*
using *unitLiteralIsEntailed* [*of uClause uLiteral elements M F*]
by *simp*
with ‹*InvariantImpliedLiterals F M*›
have *formulaEntailsLiteral* (*F* @ *val2form* (*decisions M*)) *uLiteral*
by (*simp add*: *InvariantImpliedLiteralsAndElementsEntailLiteralThenDecisionsEntailLiteral*)
ultimately
show *?thesis*
by *simp*
qed

lemma *InvariantImpliedLiteralsAfterUnitPropagate*:
fixes *M* :: *LiteralTrail* **and** *F* :: *Formula* **and** *uClause* :: *Clause* **and** *uLiteral* :: *Literal*
assumes *InvariantImpliedLiterals F M* **and**
formulaEntailsClause F uClause **and** *isUnitClause uClause uLiteral* (*elements M*) **and**
M′ = *M* @ [(*uLiteral*, *False*)]
shows *InvariantImpliedLiterals F M′*
proof −
{
fix *l* :: *Literal*
assume *l el* (*elements M′*)
have *formulaEntailsLiteral* (*F* @ *val2form* (*decisionsTo l M′*)) *l*
proof (*cases l el elements M*)
case *True*
with ‹*InvariantImpliedLiterals F M*›
have *formulaEntailsLiteral* (*F* @ *val2form* (*decisionsTo l M*)) *l*
by (*simp add*:*InvariantImpliedLiterals-def*)
moreover
from ‹*M′* = *M* @ [(*uLiteral*, *False*)]›
have (*isPrefix M M′*)
by (*simp add*:*isPrefix-def*)
with *True*
have *decisionsTo l M′* = *decisionsTo l M*

by (*simp add*: *markedElementsToPrefixElement*)
ultimately
show *?thesis*
by *simp*
next
case *False*
with ‹*l el* (*elements M'*)› ‹*M'* = *M* @ [(*uLiteral*, *False*)]›
have *l* = *uLiteral*
by (*auto split*: *if-split-asm*)
moreover
from *assms*
have *formulaEntailsLiteral* (*F* @ *val2form* (*decisionsTo uLiteral M'*)) *uLiteral*
using *InvariantImpliedLiteralsHoldsForUnitLiteral* [*of F M uClause uLiteral M'*]
by *simp*
ultimately
show *?thesis*
by *simp*
qed
}
thus *?thesis*
by (*simp add*:*InvariantImpliedLiterals-def*)
qed

lemma *InvariantVarsMAfterUnitPropagate*:
fixes *F* :: *Formula* **and** *F0* :: *Formula* **and** *M* :: *LiteralTrail* **and** *uClause* :: *Clause* **and** *uLiteral* :: *Literal* **and** *M'* :: *LiteralTrail*
assumes *InvariantVarsM M F0 Vbl* **and**
var uLiteral ∈ *vars F0* ∪ *Vbl* **and**
M' = *M* @ [(*uLiteral*, *False*)]
shows *InvariantVarsM M' F0 Vbl*
proof −
from ‹*InvariantVarsM M F0 Vbl*›
have *vars* (*elements M*) ⊆ *vars F0* ∪ *Vbl*
unfolding *InvariantVarsM-def*
.
thus *?thesis*
unfolding *InvariantVarsM-def*
using ‹*var uLiteral* ∈ *vars F0* ∪ *Vbl*›
using ‹*M'* = *M* @ [(*uLiteral*, *False*)]›
varsAppendClauses [*of elements M* [*uLiteral*]]
by *auto*
qed

lemma *InvariantConsistentAfterUnitPropagate*:
fixes *M* :: *LiteralTrail* **and** *F* :: *Formula* **and** *M'* :: *LiteralTrail* **and** *uClause* :: *Clause* **and** *uLiteral* :: *Literal*
assumes *InvariantConsistent M* **and**

isUnitClause uClause uLiteral (*elements M*) **and**
$M' = M$ @ [(*uLiteral*, *False*)]
shows *InvariantConsistent* M'
proof −
from ‹*InvariantConsistent M*›
have *consistent* (*elements M*)
unfolding *InvariantConsistent-def*
.
from ‹*isUnitClause uClause uLiteral* (*elements M*)›
have ¬ *literalFalse uLiteral* (*elements M*)
unfolding *isUnitClause-def*
by *simp*
{
assume *inconsistent* (*elements* M')
with ‹$M' = M$ @ [(*uLiteral*, *False*)]›
have *inconsistent* (*elements M*) ∨ *inconsistent* [*unitLiteral*] ∨ (∃ *l*. *literalTrue l* (*elements M*) ∧ *literalFalse l* [*uLiteral*])
using *inconsistentAppend* [*of elements M* [*uLiteral*]]
by *simp*
with ‹*consistent* (*elements M*)› **obtain** *literal*::*Literal*
where *literalTrue literal* (*elements M*) **and** *literalFalse literal* [*uLiteral*]
by *auto*
hence *literal* = *opposite uLiteral*
by *auto*
with ‹*literalTrue literal* (*elements M*)› ‹¬ *literalFalse uLiteral* (*elements M*)›
have *False*
by *simp*
} **thus** *?thesis*
unfolding *InvariantConsistent-def*
by *auto*
qed

lemma *InvariantUniqAfterUnitPropagate*:
fixes *M* :: *LiteralTrail* **and** *F* :: *Formula* **and** M' :: *LiteralTrail* **and** *uClause* :: *Clause* **and** *uLiteral* :: *Literal*
assumes *InvariantUniq M* **and**
isUnitClause uClause uLiteral (*elements M*) **and**
$M' = M$ @ [(*uLiteral*, *False*)]
shows *InvariantUniq* M'
proof−
from ‹*InvariantUniq M*›
have *uniq* (*elements M*)
unfolding *InvariantUniq-def*
.
moreover
from ‹*isUnitClause uClause uLiteral* (*elements M*)›
have ¬ *literalTrue uLiteral* (*elements M*)

unfolding *isUnitClause-def*
by *simp*
ultimately
show *?thesis*
using ‹$M' = M$ @ [(*uLiteral*, *False*)]› *uniqAppendElement*[*of elements M uLiteral*]
unfolding *InvariantUniq-def*
by *simp*
qed

lemma *InvariantReasonClausesAfterUnitPropagate*:
fixes *M* :: *LiteralTrail* **and** *F* :: *Formula* **and** M' :: *LiteralTrail* **and** *uClause* :: *Clause* **and** *uLiteral* :: *Literal*
assumes *InvariantReasonClauses F M* **and**
formulaEntailsClause F uClause **and** *isUnitClause uClause uLiteral* (*elements M*) **and**
$M' = M$ @ [(*uLiteral*, *False*)]
shows *InvariantReasonClauses F* M'
proof −
from ‹*InvariantReasonClauses F M*›
have ∗: (∀ *literal*. (*literal el* (*elements M*)) ∧ ¬ (*literal el* (*decisions M*)) ⟶
(∃ *clause*. *formulaEntailsClause F clause* ∧ (*isReason clause literal* (*elements M*))))
unfolding *InvariantReasonClauses-def*
by *simp*
{
fix *literal*::*Literal*
assume *literal el elements* M' ¬ *literal el decisions* M'
have ∃ *clause*. *formulaEntailsClause F clause* ∧ *isReason clause literal* (*elements* M')
proof (*cases literal el elements M*)
case *True*
with *assms* ‹¬ *literal el decisions* M'› **obtain** *clause*::*Clause*
where *formulaEntailsClause F clause* ∧ *isReason clause literal* (*elements* M')
using *InvariantReasonClausesHoldsForPrefixElements* [*of F M* M' *literal*]
by (*auto simp add*:*isPrefix-def*)
thus *?thesis*
by *auto*
next
case *False*
with ‹*literal el* (*elements* M')› ‹$M' = M$ @ [(*uLiteral*, *False*)]›
have *literal* = *uLiteral*
by *simp*
with ‹$M' = M$ @ [(*uLiteral*, *False*)]› ‹*isUnitClause uClause uLiteral* (*elements M*)› ‹*formulaEntailsClause F uClause*›
show *?thesis*

using *isUnitClauseIsReason* [*of uClause uLiteral elements M*]
by *auto*
qed
} **thus** *?thesis*
unfolding *InvariantReasonClauses-def*
by *simp*
qed

lemma *InvariantCFalseAfterUnitPropagate*:
fixes *M* :: *LiteralTrail* **and** *F* :: *Formula* **and** *M'* :: *LiteralTrail* **and** *uClause* :: *Clause* **and** *uLiteral* :: *Literal*
assumes *InvariantCFalse conflictFlag M C* **and**
M' = *M* @ [(*uLiteral*, *False*)]
shows *InvariantCFalse conflictFlag M' C*
proof−
from ‹*InvariantCFalse conflictFlag M C*›
have ∗: *conflictFlag* ⟶ *clauseFalse C* (*elements M*)
unfolding *InvariantCFalse-def*
.
{
assume *conflictFlag*
with ‹*M'* = *M* @ [(*uLiteral*, *False*)]› ∗
have *clauseFalse C* (*elements M'*)
by (*simp add*:*clauseFalseAppendValuation*)
}
thus *?thesis*
unfolding *InvariantCFalse-def*
by *simp*
qed

Backtrack transition rule.

lemma *InvariantImpliedLiteralsAfterBacktrack*:
fixes *F*::*Formula* **and** *M*::*LiteralTrail*
assumes *InvariantImpliedLiterals F M* **and** *InvariantUniq M* **and** *InvariantConsistent M* **and**
decisions M ≠ [] **and** *formulaFalse F* (*elements M*)
M' = (*prefixBeforeLastDecision M*) @ [(*opposite* (*lastDecision M*), *False*)]
shows *InvariantImpliedLiterals F M'*
proof −
have *isPrefix* (*prefixBeforeLastDecision M*) *M*
by (*simp add*: *isPrefixPrefixBeforeLastMarked*)
{
fix *l'*::*Literal*
assume *l' el* (*elements M'*)
let *?p* = (*prefixBeforeLastDecision M*)
let *?l* = *lastDecision M*
have *formulaEntailsLiteral* (*F* @ *val2form* (*decisionsTo l' M'*)) *l'*
proof (*cases l' el* (*elements ?p*))

case *True*
with ‹*isPrefix ?p M*›
have l' *el* (*elements M*)
using *prefixElementsAreTrailElements*[*of ?p M*]
by *auto*

with ‹*InvariantImpliedLiterals F M*›
have *formulaEntailsLiteral* (*F* @ *val2form* (*decisionsTo* l' *M*)) l'
unfolding *InvariantImpliedLiterals-def*
by *simp*
moreover
from ‹M' = *?p* @ [(*opposite ?l*, *False*)]› *True* ‹*isPrefix ?p M*›
have (*decisionsTo* l' M') = (*decisionsTo* l' *M*)
using *prefixToElementToPrefixElement*[*of ?p M* l']
unfolding *markedElementsTo-def*
by (*auto simp add*: *prefixToElementAppend*)
ultimately
show *?thesis*
by *auto*
next
case *False*
with ‹l' *el* (*elements* M')› **and** ‹M' = *?p* @ [(*opposite ?l*, *False*)]›
have *?l* = (*opposite* l')
by (*auto split*: *if-split-asm*)
hence l' = (*opposite ?l*)
by *simp*

from ‹*InvariantUniq M*› **and** ‹*markedElements M* ≠ []›
have (*decisionsTo ?l M*) = (*decisions M*)
unfolding *InvariantUniq-def*
using *markedElementsToLastMarkedAreAllMarkedElements*
by *auto*
moreover
from ‹*decisions M* ≠ []›
have *?l el* (*elements M*)
by (*simp add*: *lastMarkedIsMarkedElement markedElementsA-reElements*)
with ‹*InvariantConsistent M*›
have ¬ (*opposite ?l*) *el* (*elements M*)
unfolding *InvariantConsistent-def*
by (*simp add*: *inconsistentCharacterization*)
with ‹*isPrefix ?p M*›
have ¬ (*opposite ?l*) *el* (*elements ?p*)
using *prefixElementsAreTrailElements*[*of ?p M*]
by *auto*
with ‹M' = *?p* @ [(*opposite ?l*, *False*)]›
have *decisionsTo* (*opposite ?l*) M' = *decisions ?p*
using *markedElementsToAppend* [*of opposite ?l ?p* [(*opposite ?l*, *False*)]]

unfolding *markedElementsTo-def*
by *simp*
moreover
from ‹*InvariantUniq M*› ‹*decisions M* ≠ []›
have ¬ *?l el* (*elements ?p*)
unfolding *InvariantUniq-def*
using *lastMarkedNotInPrefixBeforeLastMarked*[*of M*]
by *simp*
hence ¬ *?l el* (*decisions ?p*)
by (*auto simp add*: *markedElementsAreElements*)
hence (*removeAll ?l* (*decisions ?p*)) = (*decisions ?p*)
by (*simp add*: *removeAll-id*)
hence (*removeAll ?l* ((*decisions ?p*) @ [*?l*])) = (*decisions ?p*)
by *simp*
from ‹*decisions M* ≠ []› *False* ‹*l'* = (*opposite ?l*)›
have (*decisions ?p*) @ [*?l*] = (*decisions M*)
using *markedElementsAreElementsBeforeLastDecisionAndLast-Decision*[*of M*]
by *simp*
with ‹(*removeAll ?l* ((*decisions ?p*) @ [*?l*])) = (*decisions ?p*)›
have (*decisions ?p*) = (*removeAll ?l* (*decisions M*))
by *simp*
moreover
from ‹*formulaFalse F* (*elements M*)› ‹*InvariantImpliedLiterals F M*›
have ¬ *satisfiable* (*F* @ (*val2form* (*decisions M*)))
using *InvariantImpliedLiteralsAndFormulaFalseThenFormulaAndDecisionsAreNotSatisfiable*[*of F M*]
by *simp*

from ‹*decisions M* ≠ []›
have *?l el* (*decisions M*)
unfolding *lastMarked-def*
by *simp*
hence [*?l*] *el val2form* (*decisions M*)
using *val2FormEl*[*of ?l* (*decisions M*)]
by *simp*
with ‹¬ *satisfiable* (*F* @ (*val2form* (*decisions M*)))›
have *formulaEntailsLiteral* (*removeAll* [*?l*] (*F* @ *val2form* (*decisions M*))) (*opposite ?l*)
using *unsatisfiableFormulaWithSingleLiteralClause*[*of F* @ *val2form* (*decisions M*) *lastDecision M*]
by *auto*
ultimately
show *?thesis*
using ‹*l'* = (*opposite ?l*)›
using *formulaEntailsLiteralRemoveAllAppend*[*of* [*?l*] *F val2form* (*removeAll ?l* (*decisions M*)) *opposite ?l*]
by (*auto simp add*: *val2FormRemoveAll*)

```
    qed
  }
  thus ?thesis
    unfolding InvariantImpliedLiterals-def
    by auto
qed

lemma InvariantConsistentAfterBacktrack:
  fixes F::Formula and M::LiteralTrail
  assumes InvariantUniq M and InvariantConsistent M and
  decisions M ≠ [] and
  M' = (prefixBeforeLastDecision M) @ [(opposite (lastDecision M),
False)]
  shows InvariantConsistent M'
proof−
  from ‹decisions M ≠ []› ‹InvariantUniq M›
  have ¬ lastDecision M el elements (prefixBeforeLastDecision M)
    unfolding InvariantUniq-def
    using lastMarkedNotInPrefixBeforeLastMarked
    by simp
  moreover
  from ‹InvariantConsistent M›
  have consistent (elements (prefixBeforeLastDecision M))
    unfolding InvariantConsistent-def
    using isPrefixPrefixBeforeLastMarked[of M]
    using isPrefixElements[of prefixBeforeLastDecision M M]
     using consistentPrefix[of elements (prefixBeforeLastDecision M)
elements M]
    by simp
  ultimately
  show ?thesis
    unfolding InvariantConsistent-def
   using ‹M' = (prefixBeforeLastDecision M) @ [(opposite (lastDecision
M), False)]›
     using inconsistentAppend[of elements (prefixBeforeLastDecision
M) [opposite (lastDecision M)]]
    by (auto split: if-split-asm)
qed

lemma InvariantUniqAfterBacktrack:
  fixes F::Formula and M::LiteralTrail
  assumes InvariantUniq M and InvariantConsistent M and
  decisions M ≠ [] and
  M' = (prefixBeforeLastDecision M) @ [(opposite (lastDecision M),
False)]
  shows InvariantUniq M'
proof−
  from ‹InvariantUniq M›
  have uniq (elements (prefixBeforeLastDecision M))
```

unfolding *InvariantUniq-def*
using *isPrefixPrefixBeforeLastMarked*[*of M*]
using *isPrefixElements*[*of prefixBeforeLastDecision M M*]
using *uniqListImpliesUniqPrefix*
by *simp*
moreover
from ‹*decisions M* ≠ []›
have *lastDecision M el* (*elements M*)
using *lastMarkedIsMarkedElement*[*of M*]
using *markedElementsAreElements*[*of lastDecision M M*]
by *simp*
with ‹*InvariantConsistent M*›
have ¬ *opposite* (*lastDecision M*) *el* (*elements M*)
unfolding *InvariantConsistent-def*
using *inconsistentCharacterization*
by *simp*
hence ¬ *opposite* (*lastDecision M*) *el* (*elements* (*prefixBeforeLastDecision M*))
using *isPrefixPrefixBeforeLastMarked*[*of M*]
using *isPrefixElements*[*of prefixBeforeLastDecision M M*]
using *prefixIsSubset*[*of elements* (*prefixBeforeLastDecision M*) *elements M*]
by *auto*
ultimately
show *?thesis*
using
‹*M'* = (*prefixBeforeLastDecision M*) @ [(*opposite* (*lastDecision M*), *False*)]›
uniqAppendElement[*of elements* (*prefixBeforeLastDecision M*) *opposite* (*lastDecision M*)]
unfolding *InvariantUniq-def*
by *simp*
qed

lemma *InvariantVarsMAfterBacktrack*:
fixes *F*::*Formula* **and** *M*::*LiteralTrail*
assumes *InvariantVarsM M F0 Vbl*
decisions M ≠ [] **and**
M' = (*prefixBeforeLastDecision M*) @ [(*opposite* (*lastDecision M*), *False*)]
shows *InvariantVarsM M' F0 Vbl*
proof−
from ‹*decisions M* ≠ []›
have *lastDecision M el* (*elements M*)
using *lastMarkedIsMarkedElement*[*of M*]
using *markedElementsAreElements*[*of lastDecision M M*]
by *simp*
hence *var* (*lastDecision M*) ∈ *vars* (*elements M*)
using *valuationContainsItsLiteralsVariable*[*of lastDecision M ele-*

ments M]
  by *simp*
 moreover
 have *vars* (*elements* (*prefixBeforeLastDecision M*)) ⊆ *vars* (*elements M*)
  using *isPrefixPrefixBeforeLastMarked*[*of M*]
  using *isPrefixElements*[*of prefixBeforeLastDecision M M*]
  using *varsPrefixValuation*[*of elements* (*prefixBeforeLastDecision M*) *elements M*]
  by *auto*
 ultimately
 show *?thesis*
  using *assms*
  using *varsAppendValuation*[*of elements* (*prefixBeforeLastDecision M*) [*opposite* (*lastDecision M*)]]
  unfolding *InvariantVarsM-def*
  by *auto*
qed

Backjump transition rule.

lemma *InvariantImpliedLiteralsAfterBackjump*:
 fixes *F*::*Formula* **and** *M*::*LiteralTrail* **and** *p*::*LiteralTrail* **and** *bClause*::*Clause* **and** *bLiteral*::*Literal*
 assumes *InvariantImpliedLiterals F M* **and**
 isPrefix p M **and** *formulaEntailsClause F bClause* **and** *isUnitClause bClause bLiteral* (*elements p*) **and**
 M′ = *p* @ [(*bLiteral*, *False*)]
 shows *InvariantImpliedLiterals F M′*
proof −
 from ‹*InvariantImpliedLiterals F M*› ‹*isPrefix p M*›
 have *InvariantImpliedLiterals F p*
  using *InvariantImpliedLiteralsHoldsForPrefix* [*of F M p*]
  by *simp*

 with *assms*
 show *?thesis*
  using *InvariantImpliedLiteralsAfterUnitPropagate* [*of F p bClause bLiteral M′*]
  by *simp*
qed

lemma *InvariantVarsMAfterBackjump*:
 fixes *F*::*Formula* **and** *M*::*LiteralTrail* **and** *p*::*LiteralTrail* **and** *bClause*::*Clause* **and** *bLiteral*::*Literal*
 assumes *InvariantVarsM M F0 Vbl* **and**
 isPrefix p M **and** *var bLiteral* ∈ *vars F0* ∪ *Vbl* **and**
 M′ = *p* @ [(*bLiteral*, *False*)]
 shows *InvariantVarsM M′ F0 Vbl*
proof −

from ‹*InvariantVarsM M F0 Vbl*›
have *vars* (*elements M*) ⊆ *vars F0* ∪ *Vbl*
unfolding *InvariantVarsM-def*
.
moreover
from ‹*isPrefix p M*›
have *vars* (*elements p*) ⊆ *vars* (*elements M*)
using *varsPrefixValuation* [*of elements p elements M*]
by (*simp add*: *isPrefixElements*)
ultimately
have *vars* (*elements p*) ⊆ *vars F0* ∪ *Vbl*
by *simp*

with ‹*vars* (*elements p*) ⊆ *vars F0* ∪ *Vbl*› *assms*
show *?thesis*
using *InvariantVarsMAfterUnitPropagate*[*of p F0 Vbl bLiteral M'*]
unfolding *InvariantVarsM-def*
by *simp*
qed

lemma *InvariantConsistentAfterBackjump*:
fixes *F*::*Formula* **and** *M*::*LiteralTrail* **and** *p*::*LiteralTrail* **and** *bClause*::*Clause* **and** *bLiteral*::*Literal*
assumes *InvariantConsistent M* **and**
isPrefix p M **and** *isUnitClause bClause bLiteral* (*elements p*) **and**
M' = *p* @ [(*bLiteral*, *False*)]
shows *InvariantConsistent M'*
proof−
from ‹*InvariantConsistent M*›
have *consistent* (*elements M*)
unfolding *InvariantConsistent-def*
.
with ‹*isPrefix p M*›
have *consistent* (*elements p*)
using *consistentPrefix* [*of elements p elements M*]
by (*simp add*: *isPrefixElements*)

with *assms*
show *?thesis*
using *InvariantConsistentAfterUnitPropagate* [*of p bClause bLiteral M'*]
unfolding *InvariantConsistent-def*
by *simp*
qed

lemma *InvariantUniqAfterBackjump*:
fixes *F*::*Formula* **and** *M*::*LiteralTrail* **and** *p*::*LiteralTrail* **and** *bClause*::*Clause* **and** *bLiteral*::*Literal*
assumes *InvariantUniq M* **and**

isPrefix p M **and** *isUnitClause bClause bLiteral* (*elements p*) **and**
M' = *p* @ [(*bLiteral*, *False*)]
shows *InvariantUniq* M'
proof –
from ‹*InvariantUniq M*›
have *uniq* (*elements M*)
unfolding *InvariantUniq-def*
.
with ‹*isPrefix p M*›
have *uniq* (*elements p*)
using *uniqElementsTrailImpliesUniqElementsPrefix* [*of p M*]
by *simp*
with *assms*
show *?thesis*
using *InvariantUniqAfterUnitPropagate*[*of p bClause bLiteral M'*]
unfolding *InvariantUniq-def*
by *simp*
qed

lemma *InvariantReasonClausesAfterBackjump*:
fixes *F*::*Formula* **and** *M*::*LiteralTrail* **and** *p*::*LiteralTrail* **and** *bClause*::*Clause* **and** *bLiteral*::*Literal*
assumes *InvariantReasonClauses F M* **and** *InvariantUniq M* **and**
isPrefix p M **and** *isUnitClause bClause bLiteral* (*elements p*) **and**
formulaEntailsClause F bClause **and**
M' = *p* @ [(*bLiteral*, *False*)]
shows *InvariantReasonClauses F* M'
proof –
from ‹*InvariantReasonClauses F M*› ‹*InvariantUniq M*› ‹*isPrefix p M*›
have *InvariantReasonClauses F p*
by (*rule InvariantReasonClausesHoldsForPrefix*)
with *assms*
show *?thesis*
using *InvariantReasonClausesAfterUnitPropagate* [*of F p bClause bLiteral M'*]
by *simp*
qed

Learn transition rule.

lemma *InvariantImpliedLiteralsAfterLearn*:
fixes *F* :: *Formula* **and** F' :: *Formula* **and** *M* :: *LiteralTrail* **and** *C* :: *Clause*
assumes *InvariantImpliedLiterals F M* **and**
F' = *F* @ [*C*]
shows *InvariantImpliedLiterals* F' *M*
proof –
from ‹*InvariantImpliedLiterals F M*›

```
  have *: ∀ l. l el (elements M) ⟶ formulaEntailsLiteral (F @
val2form (decisionsTo l M)) l
    unfolding InvariantImpliedLiterals-def
    .
  {
    fix literal :: Literal
    assume literal el (elements M)
    with *
    have formulaEntailsLiteral (F @ val2form (decisionsTo literal M))
literal
      by simp
     hence formulaEntailsLiteral (F @ [C] @ val2form (decisionsTo
literal M)) literal
    proof−
       have ∀ clause::Clause. clause el (F @ val2form (decisionsTo
literal M)) ⟶ clause el (F @ [C] @ val2form (decisionsTo literal
M))
      proof−
        {
          fix clause :: Clause
           have clause el (F @ val2form (decisionsTo literal M)) ⟶
clause el (F @ [C] @ val2form (decisionsTo literal M))
          proof
            assume clause el (F @ val2form (decisionsTo literal M))
           thus clause el (F @ [C] @ val2form (decisionsTo literal M))
              by auto
          qed
        } thus ?thesis
          by auto
      qed
       with ‹formulaEntailsLiteral (F @ val2form (decisionsTo literal
M)) literal›
      show ?thesis
        by (rule formulaEntailsLiteralSubset)
    qed
  }
  thus ?thesis
    unfolding InvariantImpliedLiterals-def
    using ‹F' = F @ [C]›
    by auto
qed

lemma InvariantReasonClausesAfterLearn:
  fixes F :: Formula and F' :: Formula and M :: LiteralTrail and C
:: Clause
  assumes InvariantReasonClauses F M and
  formulaEntailsClause F C and
  F' = F @ [C]
  shows InvariantReasonClauses F' M
```

proof –
  {
    fix *literal* :: *Literal*
    assume *literal el elements M* ∧ ¬ *literal el decisions M*
    with ‹*InvariantReasonClauses F M*› **obtain** *clause*::*Clause*
        where *formulaEntailsClause F clause isReason clause literal* (*elements M*)
      unfolding *InvariantReasonClauses-def*
      by *auto*
    from ‹*formulaEntailsClause F clause*› ‹*F′* = *F* @ [*C*]›
    have *formulaEntailsClause F′ clause*
      by (*simp add:formulaEntailsClauseAppend*)
    with ‹*isReason clause literal* (*elements M*)›
    have ∃ *clause*. *formulaEntailsClause F′ clause* ∧ *isReason clause literal* (*elements M*)
      by *auto*
  } **thus** *?thesis*
    unfolding *InvariantReasonClauses-def*
    by *simp*
qed

lemma *InvariantVarsFAfterLearn*:
  fixes *F0* :: *Formula* **and** *F* :: *Formula* **and** *F′* :: *Formula* **and** *C* :: *Clause*
  assumes *InvariantVarsF F F0 Vbl* **and**
  vars C ⊆ (*vars F0*) ∪ *Vbl* **and**
  F′ = *F* @ [*C*]
  shows *InvariantVarsF F′ F0 Vbl*
using *assms*
using *varsAppendFormulae*[*of F* [*C*]]
unfolding *InvariantVarsF-def*
by *auto*

lemma *InvariantEquivalentAfterLearn*:
  fixes *F0* :: *Formula* **and** *F* :: *Formula* **and** *F′* :: *Formula* **and** *C* :: *Clause*
  assumes *InvariantEquivalent F0 F* **and**
  formulaEntailsClause F C **and**
  F′ = *F* @ [*C*]
  shows *InvariantEquivalent F0 F′*
proof–
  from ‹*InvariantEquivalent F0 F*›
  have *equivalentFormulae F0 F*
    unfolding *InvariantEquivalent-def*
    .
  with ‹*formulaEntailsClause F C*› ‹*F′* = *F* @ [*C*]›
  have *equivalentFormulae F0* (*F* @ [*C*])
    using *extendEquivalentFormulaWithEntailedClause* [*of F0 F C*]

by *simp*
thus *?thesis*
unfolding *InvariantEquivalent-def*
using ‹$F' = F$ @ $[C]$›
by *simp*
qed

lemma *InvariantCEntailedAfterLearn*:
fixes *F0* :: *Formula* **and** *F* :: *Formula* **and** F' :: *Formula* **and** *C* :: *Clause*
assumes *InvariantCEntailed conflictFlag F C* **and**
$F' = F$ @ $[C]$
shows *InvariantCEntailed conflictFlag F′ C*
using *assms*
unfolding *InvariantCEntailed-def*
by (*auto simp add*:*formulaEntailsClauseAppend*)

Explain transition rule.

lemma *InvariantCFalseAfterExplain*:
fixes *conflictFlag*::*bool* **and** *M*::*LiteralTrail* **and** *C*::*Clause* **and** *literal* :: *Literal*
assumes *InvariantCFalse conflictFlag M C* **and**
opposite literal el C **and** *isReason reason literal* (*elements M*) **and**
C' = *resolve C reason* (*opposite literal*)
shows *InvariantCFalse conflictFlag M C′*
unfolding *InvariantCFalse-def*
proof
assume *conflictFlag*
with ‹*InvariantCFalse conflictFlag M C*›
have *clauseFalse C* (*elements M*)
unfolding *InvariantCFalse-def*
by *simp*
hence *clauseFalse* (*removeAll* (*opposite literal*) *C*) (*elements M*)
by (*simp add*: *clauseFalseIffAllLiteralsAreFalse*)
moreover
from ‹*isReason reason literal* (*elements M*)›
have *clauseFalse* (*removeAll literal reason*) (*elements M*)
unfolding *isReason-def*
by *simp*
ultimately
show *clauseFalse C′* (*elements M*)
using ‹C' = *resolve C reason* (*opposite literal*)›
resolveFalseClauses [*of opposite literal C elements M reason*]
by *simp*
qed

lemma *InvariantCEntailedAfterExplain*:
fixes *conflictFlag*::*bool* **and** *M*::*LiteralTrail* **and** *C*::*Clause* **and** *literal* :: *Literal* **and** *reason* :: *Clause*

assumes *InvariantCEntailed conflictFlag F C* **and**
formulaEntailsClause F reason **and** $C' = (resolve\ C\ reason\ (opposite\ l))$
shows *InvariantCEntailed conflictFlag F C'*
unfolding *InvariantCEntailed-def*
proof
assume *conflictFlag*
with ‹*InvariantCEntailed conflictFlag F C*›
have *formulaEntailsClause F C*
unfolding *InvariantCEntailed-def*
by *simp*
with ‹*formulaEntailsClause F reason*›
show *formulaEntailsClause F C'*
using ‹$C' = (resolve\ C\ reason\ (opposite\ l))$›
by (*simp add*:*formulaEntailsResolvent*)
qed

Conflict transition rule.

lemma *invariantCFalseAfterConflict*:
fixes *conflictFlag*::*bool* **and** *conflictFlag'*::*bool* **and** *M*::*LiteralTrail* **and** *F* :: *Formula* **and** *clause* :: *Clause* **and** *C'* :: *Clause*
assumes *conflictFlag* = *False* **and**
formulaFalse F (*elements M*) **and** *clause el F clauseFalse clause* (*elements M*) **and**
C' = *clause* **and** *conflictFlag'* = *True*
shows *InvariantCFalse conflictFlag' M C'*
unfolding *InvariantCFalse-def*
proof
from ‹*conflictFlag'* = *True*›
show *clauseFalse C'* (*elements M*)
using ‹*clauseFalse clause* (*elements M*)› ‹*C'* = *clause*›
by *simp*
qed

lemma *invariantCEntailedAfterConflict*:
fixes *conflictFlag*::*bool* **and** *conflictFlag'*::*bool* **and** *M*::*LiteralTrail* **and** *F* :: *Formula* **and** *clause* :: *Clause* **and** *C'* :: *Clause*
assumes *conflictFlag* = *False* **and**
formulaFalse F (*elements M*) **and** *clause el F* **and** *clauseFalse clause* (*elements M*) **and**
C' = *clause* **and** *conflictFlag'* = *True*
shows *InvariantCEntailed conflictFlag' F C'*
unfolding *InvariantCEntailed-def*
proof
from ‹*conflictFlag'* = *True*›
show *formulaEntailsClause F C'*
using ‹*clause el F*› ‹*C'* = *clause*›
by (*simp add*:*formulaEntailsItsClauses*)
qed

UNSAT report

```
lemma unsatReport:
  fixes F :: Formula and M :: LiteralTrail and F0 :: Formula
  assumes InvariantImpliedLiterals F M and InvariantEquivalent F0
F and
  decisions M = [] and formulaFalse F (elements M)
  shows ¬ satisfiable F0
proof−
  have formulaEntailsValuation F (elements M)
  proof−
    {
      fix literal::Literal
      assume literal el (elements M)
      from ‹decisions M = []›
      have decisionsTo literal M = []
      by (simp add:markedElementsEmptyImpliesMarkedElementsToEmpty)
      with ‹literal el (elements M)› ‹InvariantImpliedLiterals F M›
      have formulaEntailsLiteral F literal
        unfolding InvariantImpliedLiterals-def
        by auto
    }
    thus ?thesis
      unfolding formulaEntailsValuation-def
      by simp
  qed
  with ‹formulaFalse F (elements M)›
  have ¬ satisfiable F
    by (simp add:formulaFalseInEntailedValuationIsUnsatisfiable)
  with ‹InvariantEquivalent F0 F›
  show ?thesis
    unfolding InvariantEquivalent-def
    by (simp add:satisfiableEquivalent)
qed

lemma unsatReportExtensiveExplain:
  fixes F :: Formula and M :: LiteralTrail and F0 :: Formula and C
:: Clause and conflictFlag :: bool
  assumes InvariantEquivalent F0 F and InvariantCEntailed conflict-
Flag F C and
  conflictFlag and C = []
  shows ¬ satisfiable F0
proof−
  from ‹conflictFlag› ‹InvariantCEntailed conflictFlag F C›
  have formulaEntailsClause F C
    unfolding InvariantCEntailed-def
    by simp
  with ‹C=[]›
  have ¬ satisfiable F
    by (simp add:formulaUnsatIffImpliesEmptyClause)
```

with ‹*InvariantEquivalent F0 F*›
show *?thesis*
unfolding *InvariantEquivalent-def*
by (*simp add*:*satisfiableEquivalent*)
qed

SAT Report

lemma *satReport*:
fixes *F0* :: *Formula* **and** *F* :: *Formula* **and** *M*::*LiteralTrail*
assumes *vars F0* ⊆ *Vbl* **and** *InvariantVarsF F F0 Vbl* **and** *InvariantConsistent M* **and** *InvariantEquivalent F0 F* **and**
¬ *formulaFalse F* (*elements M*) **and** *vars* (*elements M*) ⊇ *Vbl*
shows *model* (*elements M*) *F0*
proof−
from ‹*InvariantConsistent M*›
have *consistent* (*elements M*)
unfolding *InvariantConsistent-def*
.
moreover
from ‹*InvariantVarsF F F0 Vbl*›
have *vars F* ⊆ *vars F0* ∪ *Vbl*
unfolding *InvariantVarsF-def*
.
with ‹*vars F0* ⊆ *Vbl*›
have *vars F* ⊆ *Vbl*
by *auto*
with ‹*vars* (*elements M*) ⊇ *Vbl*›
have *vars F* ⊆ *vars* (*elements M*)
by *simp*
hence *formulaTrue F* (*elements M*) ∨ *formulaFalse F* (*elements M*)
by (*simp add*:*totalValuationForFormulaDefinesItsValue*)
with ‹¬ *formulaFalse F* (*elements M*)›
have *formulaTrue F* (*elements M*)
by *simp*
ultimately
have *model* (*elements M*) *F*
by *simp*
with ‹*InvariantEquivalent F0 F*›
show *?thesis*
unfolding *InvariantEquivalent-def*
unfolding *equivalentFormulae-def*
by *auto*
qed

4.3 Different characterizations of backjumping

In this section, different characterization of applicability of backjumping will be given.

The clause satisfies the *Unique Implication Point UIP* condition if the level of all its literals is stricly lower then the level of its last asserted literal

definition
isUIP l c M ==
isLastAssertedLiteral (*opposite l*) (*oppositeLiteralList c*)(*elements M*) ∧
(∀ *l'*. *l' el c* ∧ *l'* ≠ *l* ⟶ *elementLevel* (*opposite l'*) *M* < *elementLevel* (*opposite l*) *M*)

Backjump level is a nonegative integer such that it is stricly lower than the level of the last asserted literal of a clause, and greater or equal then levels of all its other literals.

definition
isBackjumpLevel level l c M ==
isLastAssertedLiteral (*opposite l*) (*oppositeLiteralList c*)(*elements M*) ∧
0 ≤ *level* ∧ *level* < *elementLevel* (*opposite l*) *M* ∧
(∀ *l'*. *l' el c* ∧ *l'* ≠ *l* ⟶ *elementLevel* (*opposite l'*) *M* ≤ *level*)

lemma *lastAssertedLiteralHasHighestElementLevel*:
fixes *literal* :: *Literal* **and** *clause* :: *Clause* **and** *M* :: *LiteralTrail*
assumes *isLastAssertedLiteral literal clause* (*elements M*) **and** *uniq* (*elements M*)
shows ∀ *l'*. *l' el clause* ∧ *l' el elements M* ⟶ *elementLevel l' M* <= *elementLevel literal M*
proof –
{
fix *l'* :: *Literal*
assume *l' el clause l' el elements M*
hence *elementLevel l' M* <= *elementLevel literal M*
proof (*cases l'* = *literal*)
case *True*
thus *?thesis*
by *simp*
next
case *False*
from ‹*isLastAssertedLiteral literal clause* (*elements M*)›
have *literalTrue literal* (*elements M*)
∀ *l*. *l el clause* ∧ *l* ≠ *literal* ⟶ ¬ *precedes literal l* (*elements M*)
by (*auto simp add:isLastAssertedLiteral-def*)
with ‹*l' el clause*› *False*
have ¬ *precedes literal l'* (*elements M*)
by *simp*
with *False* ‹*l' el* (*elements M*)› ‹*literalTrue literal* (*elements M*)›
have *precedes l' literal* (*elements M*)
using *precedesTotalOrder* [*of l' elements M literal*]

```
      by simp
    with ‹uniq (elements M)›
    show ?thesis
      using elementLevelPrecedesLeq [of l' literal M]
      by auto
  qed
 }
 thus ?thesis
   by simp
qed
```

When backjump clause contains only a single literal, then the backjump level is 0.

```
lemma backjumpLevelZero:
 fixes M :: LiteralTrail and C :: Clause and l :: Literal
 assumes
  isLastAssertedLiteral (opposite l) (oppositeLiteralList C) (elements
M) and
  elementLevel (opposite l) M > 0 and
  set C = {l}
 shows
  isBackjumpLevel 0 l C M
proof−
 have ∀ l'. l' el C ∧ l' ≠ l ⟶ elementLevel (opposite l') M ≤ 0
 proof−
   {
    fix l'::Literal
    assume l' el C ∧ l' ≠ l
    hence False
      using ‹set C = {l}›
      by auto
   } thus ?thesis
    by auto
 qed
 with ‹elementLevel (opposite l) M > 0›
  ‹isLastAssertedLiteral (opposite l) (oppositeLiteralList C) (elements
M)›
 show ?thesis
   unfolding isBackjumpLevel-def
   by auto
qed
```

When backjump clause contains more than one literal, then the level of the second last asserted literal can be taken as a backjump level.

```
lemma backjumpLevelLastLast:
 fixes M :: LiteralTrail and C :: Clause and l :: Literal
 assumes
 isUIP l C M and
```

```
  uniq (elements M) and
  clauseFalse C (elements M) and
  isLastAssertedLiteral (opposite ll) (removeAll (opposite l) (oppositeLiteralList
C)) (elements M)
  shows
  isBackjumpLevel (elementLevel (opposite ll) M) l C M
proof−
  from ‹isUIP l C M›
  have isLastAssertedLiteral (opposite l) (oppositeLiteralList C) (elements
M)
    unfolding isUIP-def
    by simp

  from ‹isLastAssertedLiteral (opposite ll) (removeAll (opposite l)
(oppositeLiteralList C)) (elements M)›
  have literalTrue (opposite ll) (elements M) (opposite ll) el (removeAll
(opposite l) (oppositeLiteralList C))
    unfolding isLastAssertedLiteral-def
    by auto

  have ∀ l'. l' el (oppositeLiteralList C) ⟶ literalTrue l' (elements
M)
  proof−
    {
      fix l'::Literal
      assume l' el oppositeLiteralList C
      hence opposite l' el C
        using literalElListIffOppositeLiteralElOppositeLiteralList[of op-
posite l' C]
        by simp
      with ‹clauseFalse C (elements M)›
      have literalTrue l' (elements M)
        by (auto simp add: clauseFalseIffAllLiteralsAreFalse)
    }
    thus ?thesis
      by simp
  qed

  have ∀ l'. l' el C ∧ l' ≠ l ⟶
    elementLevel (opposite l') M <= elementLevel (opposite ll) M
  proof−
    {
      fix l' :: Literal
      assume l' el C ∧ l' ≠ l
      hence (opposite l') el (oppositeLiteralList C) opposite l' ≠ opposite
l
        using literalElListIffOppositeLiteralElOppositeLiteralList
        by auto
      hence opposite l' el (removeAll (opposite l) (oppositeLiteralList
```

C))
    by *simp*

  from ‹*opposite* l' *el* (*oppositeLiteralList* C)›
    ‹∀ l'. l' *el* (*oppositeLiteralList* C) ⟶ *literalTrue* l' (*elements* M)›
  have *literalTrue* (*opposite* l') (*elements* M)
    by *simp*

  with ‹*opposite* l' *el* (*removeAll* (*opposite* l) (*oppositeLiteralList* C))›
    ‹*isLastAssertedLiteral* (*opposite* *ll*) (*removeAll* (*opposite* l) (*oppositeLiteralList* C)) (*elements* M)›
    ‹*uniq* (*elements* M)›
  have *elementLevel* (*opposite* l') M <= *elementLevel* (*opposite* *ll*) M
    using *lastAssertedLiteralHasHighestElementLevel*[*of opposite ll removeAll* (*opposite* l) (*oppositeLiteralList* C) M]
    by *auto*
  }
  thus *?thesis*
    by *simp*
qed
moreover
from ‹*literalTrue* (*opposite* *ll*) (*elements* M)›
have *elementLevel* (*opposite* *ll*) $M \geq 0$
  by *simp*
moreover
from ‹(*opposite* *ll*) *el* (*removeAll* (*opposite* l) (*oppositeLiteralList* C))›
have *ll el* C **and** $ll \neq l$
  using *literalElListIffOppositeLiteralElOppositeLiteralList*[*of ll* C]
  by *auto*
from ‹*isUIP* l C M›
have ∀ l'. l' *el* C ∧ $l' \neq l$ ⟶ *elementLevel* (*opposite* l') M < *elementLevel* (*opposite* l) M
  unfolding *isUIP-def*
  by *simp*
with ‹*ll el* C› ‹$ll \neq l$›
have *elementLevel* (*opposite* *ll*) M < *elementLevel* (*opposite* l) M
  by *simp*
ultimately
show *?thesis*
  using ‹*isLastAssertedLiteral* (*opposite* l) (*oppositeLiteralList* C) (*elements* M)›
  unfolding *isBackjumpLevel-def*
  by *simp*
qed

if UIP is reached then there exists correct backjump level.

lemma *isUIPExistsBackjumpLevel*:
fixes *M* :: *LiteralTrail* **and** *c* :: *Clause* **and** *l* :: *Literal*
assumes
clauseFalse c (*elements M*) **and**
isUIP l c M **and**
uniq (*elements M*) **and**
elementLevel (*opposite l*) *M* > *0*
shows
∃ *level*. (*isBackjumpLevel level l c M*)
proof−
from ‹*isUIP l c M*›
have *isLastAssertedLiteral* (*opposite l*) (*oppositeLiteralList c*) (*elements M*)
unfolding *isUIP-def*
by *simp*
show *?thesis*
proof (*cases set c* = {*l*})
case *True*
with ‹*elementLevel* (*opposite l*) *M* > *0*› ‹*isLastAssertedLiteral* (*opposite l*) (*oppositeLiteralList c*) (*elements M*)›
have *isBackjumpLevel 0 l c M*
using *backjumpLevelZero*[*of l c M*]
by *auto*
thus *?thesis*
by *auto*
next
case *False*
have ∃ *literal*. *isLastAssertedLiteral literal* (*removeAll* (*opposite l*) (*oppositeLiteralList c*)) (*elements M*)
proof−
let *?ll* = *getLastAssertedLiteral* (*oppositeLiteralList* (*removeAll l c*)) (*elements M*)

from ‹*clauseFalse c* (*elements M*)›
have *clauseFalse* (*removeAll l c*) (*elements M*)
by (*simp add*:*clauseFalseRemove*)
moreover
have *removeAll l c* ≠ []
proof−
have (*set c*) ⊆ {*l*} ∪ *set* (*removeAll l c*)
by *auto*

from ‹*isLastAssertedLiteral* (*opposite l*) (*oppositeLiteralList c*) (*elements M*)›
have (*opposite l*) *el oppositeLiteralList c*
unfolding *isLastAssertedLiteral-def*
by *simp*
hence *l el c*
using *literalElListIffOppositeLiteralElOppositeLiteralList*[*of l*

```
c]
          by simp
        hence l ∈ set c
          by simp
        {
          assume ¬ ?thesis
          hence set (removeAll l c) = {}
            by simp
          with ‹(set c) ⊆ {l} ∪ set (removeAll l c)›
          have set c ⊆ {l}
            by simp
          with ‹l ∈ set c›
          have set c = {l}
            by auto
          with False
          have False
            by simp
        }
        thus ?thesis
          by auto
      qed
      ultimately
       have isLastAssertedLiteral ?ll (oppositeLiteralList (removeAll l
c)) (elements M)
        using ‹uniq (elements M)›
        using getLastAssertedLiteralCharacterization [of removeAll l c
elements M]
        by simp
    hence isLastAssertedLiteral ?ll (removeAll (opposite l) (oppositeLiteralList
c)) (elements M)
        using oppositeLiteralListRemove[of l c]
        by simp
      thus ?thesis
        by auto
    qed
    then obtain ll::Literal where isLastAssertedLiteral ll (removeAll
(opposite l) (oppositeLiteralList c)) (elements M)
      by auto

    with ‹uniq (elements M)› ‹clauseFalse c (elements M)› ‹isUIP l c
M›
    have isBackjumpLevel (elementLevel ll M) l c M
      using backjumpLevelLastLast[of l c M opposite ll]
      by auto
    thus ?thesis
      by auto
  qed
qed
```

Backjump level condition ensures that the backjump clause is unit in the prefix to backjump level.

lemma *isBackjumpLevelEnsuresIsUnitInPrefix*:
fixes *M* :: *LiteralTrail* **and** *conflictFlag* :: *bool* **and** *c* :: *Clause* **and** *l* :: *Literal*
assumes *consistent* (*elements M*) **and** *uniq* (*elements M*) **and** *clauseFalse c* (*elements M*) **and** *isBackjumpLevel level l c M*
shows *isUnitClause c l* (*elements* (*prefixToLevel level M*))
proof −
from ‹*isBackjumpLevel level l c M*›
have *isLastAssertedLiteral* (*opposite l*) (*oppositeLiteralList c*)(*elements M*)
$0 \leq level$ $level < elementLevel\ (opposite\ l)\ M$ **and**
$*$: $\forall\ l'.\ l'\ el\ c \wedge l' \neq l \longrightarrow elementLevel\ (opposite\ l')\ M \leq level$
unfolding *isBackjumpLevel-def*
by *auto*

from ‹*isLastAssertedLiteral* (*opposite l*)(*oppositeLiteralList c*) (*elements M*)›
have *l el c literalTrue* (*opposite l*) (*elements M*)
using *isLastAssertedCharacterization* [*of opposite l c elements M*]
by *auto*

have ¬ *literalFalse l* (*elements* (*prefixToLevel level M*))
using ‹*level* < *elementLevel* (*opposite l*) *M*› ‹*0* <= *level*› ‹*uniq* (*elements M*)›
by (*simp add*: *literalNotInEarlierLevelsThanItsLevel*)
moreover
have ¬ *literalTrue l* (*elements* (*prefixToLevel level M*))
proof −
from ‹*consistent* (*elements M*)› ‹*literalTrue* (*opposite l*) (*elements M*)›
have ¬ *literalFalse* (*opposite l*) (*elements M*)
by (*auto simp add*:*inconsistentCharacterization*)
thus *?thesis*
using *isPrefixPrefixToLevel*[*of level M*]
prefixElementsAreTrailElements[*of prefixToLevel level M M*]
unfolding *prefixToLevel-def*
by *auto*
qed
moreover
have $\forall\ l'.\ l'\ el\ c \wedge l' \neq l \longrightarrow literalFalse\ l'$ (*elements* (*prefixToLevel level M*))
proof −
{
fix *l'* :: *Literal*
assume $l'\ el\ c$ $l' \neq l$

from ‹*l' el c*› ‹*clauseFalse c* (*elements M*)›
have *literalFalse l'* (*elements M*)

by (*simp add*:*clauseFalseIffAllLiteralsAreFalse*)

have *literalFalse* l' (*elements* (*prefixToLevel level M*))
proof −
from ‹l' *el c*› ‹$l' \neq l$›
have *elementLevel* (*opposite* l') *M* <= *level*
using ∗
by *auto*

thus *?thesis*
using ‹*literalFalse* l' (*elements M*)›
‹*0* <= *level*›
elementLevelLtLevelImpliesMemberPrefixToLevel[*of opposite* l' *M level*]
by *simp*
qed
} **thus** *?thesis*
by *auto*
qed
ultimately
show *?thesis*
using ‹*l el c*›
unfolding *isUnitClause-def*
by *simp*
qed

Backjump level is minimal if there is no smaller level which satisfies the backjump level condition. The following definition gives operative characterization of this notion.

definition
isMinimalBackjumpLevel level l c M ==
isBackjumpLevel level l c M ∧
(*if set c* ≠ {*l*} *then*
(∃ *ll*. *ll el c* ∧ *elementLevel* (*opposite ll*) *M* = *level*)
else
level = *0*
)

lemma *isMinimalBackjumpLevelCharacterization*:
assumes
isUIP l c M
clauseFalse c (*elements M*)
uniq (*elements M*)
shows
isMinimalBackjumpLevel level l c M =
(*isBackjumpLevel level l c M* ∧
(∀ *level'*. *level'* < *level* ⟶ ¬ *isBackjumpLevel level' l c M*)) (**is** *?lhs* = *?rhs*)
proof

assume *?lhs*
show *?rhs*
proof (*cases set c* = {*l*})
case *True*
thus *?thesis*
using ‹*?lhs*›
unfolding *isMinimalBackjumpLevel-def*
by *auto*
next
case *False*
with ‹*?lhs*›
obtain *ll*
where *ll el c elementLevel* (*opposite ll*) *M* = *level isBackjumpLevel level l c M*
unfolding *isMinimalBackjumpLevel-def*
by *auto*
have *l* ≠ *ll*
using ‹*isMinimalBackjumpLevel level l c M*›
using ‹*elementLevel* (*opposite ll*) *M* = *level*›
unfolding *isMinimalBackjumpLevel-def*
unfolding *isBackjumpLevel-def*
by *auto*

show *?thesis*
using ‹*isBackjumpLevel level l c M*›
using ‹*elementLevel* (*opposite ll*) *M* = *level*›
using ‹*ll el c*› ‹*l* ≠ *ll*›
unfolding *isBackjumpLevel-def*
by *force*
qed
next
assume *?rhs*
show *?lhs*
proof (*cases set c* = {*l*})
case *True*
thus *?thesis*
using ‹*?rhs*›
using *backjumpLevelZero*[*of l c M*]
unfolding *isMinimalBackjumpLevel-def*
unfolding *isBackjumpLevel-def*
by *auto*
next
case *False*
from ‹*?rhs*›
have *l el c*
unfolding *isBackjumpLevel-def*
using *literalElListIffOppositeLiteralElOppositeLiteralList*[*of l c*]
unfolding *isLastAssertedLiteral-def*
by *simp*

let *?oll* = *getLastAssertedLiteral* (*removeAll* (*opposite l*) (*oppositeLiteralList c*)) (*elements M*)

have *clauseFalse* (*removeAll l c*) (*elements M*)
using ‹*clauseFalse c* (*elements M*)›
by (*simp add*: *clauseFalseIffAllLiteralsAreFalse*)
moreover
have *removeAll l c* ≠ []
proof−
{
assume ¬ *?thesis*
hence *set* (*removeAll l c*) = {}
by *simp*
hence *set c* ⊆ {*l*}
by *simp*
hence *False*
using ‹*set c* ≠ {*l*}›
using ‹*l el c*›
by *auto*
} **thus** *?thesis*
by *auto*
qed
ultimately
have *isLastAssertedLiteral ?oll* (*removeAll* (*opposite l*) (*oppositeLiteralList c*)) (*elements M*)
using ‹*uniq* (*elements M*)›
using *getLastAssertedLiteralCharacterization*[*of removeAll l c elements M*]
using *oppositeLiteralListRemove*[*of l c*]
by *simp*
hence *isBackjumpLevel* (*elementLevel ?oll M*) *l c M*
using *assms*
using *backjumpLevelLastLast*[*of l c M opposite ?oll*]
by *auto*

have *?oll el* (*removeAll* (*opposite l*) (*oppositeLiteralList c*))
using ‹*isLastAssertedLiteral ?oll* (*removeAll* (*opposite l*) (*oppositeLiteralList c*)) (*elements M*)›
unfolding *isLastAssertedLiteral-def*
by *simp*
hence *?oll el* (*oppositeLiteralList c*) *?oll* ≠ *opposite l*
by *auto*
hence *opposite ?oll el c*
using *literalElListIffOppositeLiteralElOppositeLiteralList*[*of ?oll oppositeLiteralList c*]
by *simp*
from ‹*?oll* ≠ *opposite l*›
have *opposite ?oll* ≠ *l*

```
    using oppositeSymmetry[of ?oll l]
    by simp

  have elementLevel ?oll M ≥ level
  proof−
    {
      assume elementLevel ?oll M < level
      hence ¬ isBackjumpLevel  (elementLevel ?oll M) l c M
        using ‹?rhs›
        by simp
      with ‹isBackjumpLevel  (elementLevel ?oll M) l c M›
      have False
        by simp
    } thus ?thesis
      by force
  qed
  moreover
  from ‹?rhs›
  have elementLevel ?oll M ≤ level
    using ‹opposite ?oll el c›
    using ‹opposite ?oll ≠ l›
    unfolding isBackjumpLevel-def
    by auto
  ultimately
  have elementLevel ?oll M = level
    by simp
  show ?thesis
    using ‹opposite ?oll el c›
    using ‹elementLevel ?oll M = level›
    using ‹?rhs›
    using ‹set c ≠ {l}›
    unfolding isMinimalBackjumpLevel-def
    by (auto simp del: set-removeAll)
 qed
qed

lemma isMinimalBackjumpLevelEnsuresIsNotUnitBeforePrefix:
 fixes M :: LiteralTrail and conflictFlag :: bool and c :: Clause and
l :: Literal
 assumes consistent (elements M) and uniq (elements M) and
 clauseFalse c (elements M) isMinimalBackjumpLevel level l c M and
 level′ < level
 shows ¬ (∃ l′. isUnitClause c l′ (elements (prefixToLevel level′ M)))
proof−
 from ‹isMinimalBackjumpLevel level l c M›
 have isUnitClause c l (elements (prefixToLevel level M))
  using assms
  using isBackjumpLevelEnsuresIsUnitInPrefix[of M c level l]
  unfolding isMinimalBackjumpLevel-def
```

by *simp*
hence ¬ *literalFalse l* (*elements* (*prefixToLevel level M*))
unfolding *isUnitClause-def*
by *auto*
hence ¬ *literalFalse l* (*elements M*) ∨ *elementLevel* (*opposite l*) *M* > *level*
using *elementLevelLtLevelImpliesMemberPrefixToLevel*[*of l M level*]
using *elementLevelLtLevelImpliesMemberPrefixToLevel*[*of opposite l M level*]
by (*force*)+

have ¬ *literalFalse l* (*elements* (*prefixToLevel level′ M*))
proof (*cases* ¬ *literalFalse l* (*elements M*))
case *True*
thus *?thesis*
using *prefixIsSubset*[*of elements* (*prefixToLevel level′ M*) *elements M*]
using *isPrefixPrefixToLevel*[*of level′ M*]
using *isPrefixElements*[*of prefixToLevel level′ M M*]
by *auto*
next
case *False*
with ‹¬ *literalFalse l* (*elements M*) ∨ *elementLevel* (*opposite l*) *M* > *level*›
have *level* < *elementLevel* (*opposite l*) *M*
by *simp*
thus *?thesis*
using *prefixToLevelElementsElementLevel*[*of opposite l level′ M*]
using ‹*level′* < *level*›
by *auto*
qed

show *?thesis*
proof (*cases set c* ≠ {*l*})
case *True*
from ‹*isMinimalBackjumpLevel level l c M*›
obtain *ll*
where *ll el c elementLevel* (*opposite ll*) *M* = *level*
using ‹*set c* ≠ {*l*}›
unfolding *isMinimalBackjumpLevel-def*
by *auto*
hence ¬ *literalFalse ll* (*elements* (*prefixToLevel level′ M*))
using *literalNotInEarlierLevelsThanItsLevel*[*of level′ opposite ll M*]
using ‹*level′* < *level*›
by *simp*

have *l* ≠ *ll*
using ‹*isMinimalBackjumpLevel level l c M*›

```
      using ‹elementLevel (opposite ll) M = level›
      unfolding isMinimalBackjumpLevel-def
      unfolding isBackjumpLevel-def
      by auto

    {
      assume ¬ ?thesis
      then obtain l'
        where isUnitClause c l' (elements (prefixToLevel level' M))
        by auto
      have False
      proof (cases l = l')
        case True
        thus ?thesis
          using ‹l ≠ ll› ‹ll el c›
          using ‹¬ literalFalse ll (elements (prefixToLevel level' M))›
          using ‹isUnitClause c l' (elements (prefixToLevel level' M))›
          unfolding isUnitClause-def
          by auto
      next
        case False
        have l el c
          using ‹isMinimalBackjumpLevel level l c M›
          unfolding isMinimalBackjumpLevel-def
          unfolding isBackjumpLevel-def
          unfolding isLastAssertedLiteral-def
          using literalElListIffOppositeLiteralElOppositeLiteralList[of l
c]
          by simp
        thus ?thesis
          using False
          using ‹¬ literalFalse l (elements (prefixToLevel level' M))›
          using ‹isUnitClause c l' (elements (prefixToLevel level' M))›
          unfolding isUnitClause-def
          by auto
      qed
    } thus ?thesis
      by auto
  next
    case False
    with ‹isMinimalBackjumpLevel level l c M›
    have level = 0
      unfolding isMinimalBackjumpLevel-def
      by simp
    with ‹level' < level›
    show ?thesis
      by simp
  qed
qed
```

If all literals in a clause are decision literals, then UIP is reached.

lemma *allDecisionsThenUIP*:
fixes *M* :: *LiteralTrail* **and** *c*:: *Clause*
assumes (*uniq* (*elements M*)) **and**
∀ *l'*. *l' el c* ⟶ (*opposite l'*) *el* (*decisions M*)
isLastAssertedLiteral (*opposite l*) (*oppositeLiteralList c*) (*elements M*)
shows *isUIP l c M*
proof−
from ‹*isLastAssertedLiteral* (*opposite l*) (*oppositeLiteralList c*) (*elements M*)›
have *l el c* (*opposite l*) *el* (*elements M*)
and ∗: ∀ *l'*. *l' el* (*oppositeLiteralList c*) ∧ *l'* ≠ *opposite l* ⟶ ¬ *precedes* (*opposite l*) *l'* (*elements M*)
unfolding *isLastAssertedLiteral-def*
using *literalElListIffOppositeLiteralElOppositeLiteralList*
by *auto*
with ‹∀ *l'*. *l' el c* ⟶ (*opposite l'*) *el* (*decisions M*)›
have (*opposite l*) *el* (*decisions M*)
by *simp*
{
fix *l'* :: *Literal*
assume *l' el c l'* ≠ *l*
hence *opposite l' el* (*oppositeLiteralList c*) **and** *opposite l'* ≠ *opposite l*
using *literalElListIffOppositeLiteralElOppositeLiteralList*[*of l' c*]
by *auto*
with ∗
have ¬ *precedes* (*opposite l*) (*opposite l'*) (*elements M*)
by *simp*

from ‹*l' el c*› ‹∀ *l*. *l el c* ⟶ (*opposite l*) *el* (*decisions M*)›
have (*opposite l'*) *el* (*decisions M*)
by *auto*
hence (*opposite l'*) *el* (*elements M*)
by (*simp add*:*markedElementsAreElements*)

from ‹(*opposite l*) *el* (*elements M*)› ‹(*opposite l'*) *el* (*elements M*)› ‹*l'* ≠ *l*›
‹¬ *precedes* (*opposite l*) (*opposite l'*) (*elements M*)›
have *precedes* (*opposite l'*) (*opposite l*) (*elements M*)
using *precedesTotalOrder* [*of opposite l elements M opposite l'*]
by *simp*
with ‹*uniq* (*elements M*)›
have *elementLevel* (*opposite l'*) *M* <= *elementLevel* (*opposite l*) *M*
by (*auto simp add*:*elementLevelPrecedesLeq*)
moreover
from ‹*uniq* (*elements M*)› ‹(*opposite l*) *el* (*decisions M*)› ‹(*opposite*

l') *el* (*decisions M*)› ‹$l' \neq l$›
have *elementLevel* (*opposite l*) $M \neq$ *elementLevel* (*opposite l'*) *M*
using *differentMarkedElementsHaveDifferentLevels*[*of M opposite l opposite l'*]
by *simp*
ultimately
have *elementLevel* (*opposite l'*) $M <$ *elementLevel* (*opposite l*) *M*
by *simp*
}
thus *?thesis*
using ‹*isLastAssertedLiteral* (*opposite l*) (*oppositeLiteralList c*) (*elements M*)›
unfolding *isUIP-def*
by *simp*
qed

If last asserted literal of a clause is a decision literal, then UIP is reached.

lemma *lastDecisionThenUIP*:
fixes *M* :: *LiteralTrail* **and** *c*:: *Clause*
assumes (*uniq* (*elements M*)) **and**
(*opposite l*) *el* (*decisions M*)
clauseFalse c (*elements M*)
isLastAssertedLiteral (*opposite l*) (*oppositeLiteralList c*) (*elements M*)
shows *isUIP l c M*
proof−
from ‹*isLastAssertedLiteral* (*opposite l*) (*oppositeLiteralList c*) (*elements M*)›
have *l el c* (*opposite l*) *el* (*elements M*)
and ∗: $\forall l'.\ l'$ *el* (*oppositeLiteralList c*) $\wedge\ l' \neq$ *opposite l* $\longrightarrow \neg$ *precedes* (*opposite l*) l' (*elements M*)
unfolding *isLastAssertedLiteral-def*
using *literalElListIffOppositeLiteralElOppositeLiteralList*
by *auto*
{
fix l' :: *Literal*
assume l' *el c* $l' \neq l$
hence *opposite* l' *el* (*oppositeLiteralList c*) **and** *opposite* $l' \neq$ *opposite l*
using *literalElListIffOppositeLiteralElOppositeLiteralList*[*of* l' *c*]
by *auto*
with ∗
have ¬ *precedes* (*opposite l*) (*opposite l'*) (*elements M*)
by *simp*

have (*opposite l'*) *el* (*elements M*)
using ‹l' *el c*› ‹*clauseFalse c* (*elements M*)›
by (*simp add*: *clauseFalseIffAllLiteralsAreFalse*)

from ‹(*opposite l*) *el* (*elements M*)› ‹(*opposite l'*) *el* (*elements M*)›
‹$l' \neq l$›
‹¬ *precedes* (*opposite l*) (*opposite l'*) (*elements M*)›
have *precedes* (*opposite l'*) (*opposite l*) (*elements M*)
using *precedesTotalOrder* [*of opposite l elements M opposite l'*]
by *simp*

hence *elementLevel* (*opposite l'*) $M <$ *elementLevel* (*opposite l*) *M*
using *elementLevelPrecedesMarkedElementLt*[*of M opposite l'*
opposite l]
using ‹*uniq* (*elements M*)›
using ‹*opposite l el* (*decisions M*)›
using ‹$l' \neq l$›
by *simp*
}
thus *?thesis*
using ‹*isLastAssertedLiteral* (*opposite l*) (*oppositeLiteralList c*)
(*elements M*)›
unfolding *SatSolverVerification.isUIP-def*
by *simp*
qed

If all literals in a clause are decision literals, then there exists a backjump level for that clause.

lemma *allDecisionsThenExistsBackjumpLevel*:
fixes *M* :: *LiteralTrail* **and** *c*:: *Clause*
assumes (*uniq* (*elements M*)) **and**
∀ *l'*. *l' el c* ⟶ (*opposite l'*) *el* (*decisions M*)
isLastAssertedLiteral (*opposite l*) (*oppositeLiteralList c*) (*elements*
M)
shows ∃ *level*. (*isBackjumpLevel level l c M*)
proof−
from *assms*
have *isUIP l c M*
using *allDecisionsThenUIP*
by *simp*
moreover
from ‹*isLastAssertedLiteral* (*opposite l*) (*oppositeLiteralList c*) (*elements*
M)›
have *l el c*
unfolding *isLastAssertedLiteral-def*
using *literalElListIffOppositeLiteralElOppositeLiteralList*
by *simp*
with ‹∀ *l'*. *l' el c* ⟶ (*opposite l'*) *el* (*decisions M*)›
have (*opposite l*) *el* (*decisions M*)
by *simp*
hence *elementLevel* (*opposite l*) $M > 0$
using ‹*uniq* (*elements M*)›

elementLevelMarkedGeq1[*of M opposite l*]
by *auto*
moreover
have *clauseFalse c* (*elements M*)
proof−
{
fix *l*′::*Literal*
assume *l*′ *el c*
with ‹∀ *l*′. *l*′ *el c* ⟶ (*opposite l*′) *el* (*decisions M*)›
have (*opposite l*′) *el* (*decisions M*)
by *simp*
hence *literalFalse l*′ (*elements M*)
using *markedElementsAreElements*
by *simp*
}
thus *?thesis*
using *clauseFalseIffAllLiteralsAreFalse*
by *simp*
qed
ultimately
show *?thesis*
using ‹*uniq* (*elements M*)›
using *isUIPExistsBackjumpLevel*
by *simp*
qed

Explain is applicable to each non-decision literal in a clause.

lemma *explainApplicableToEachNonDecision*:
fixes *F* :: *Formula* **and** *M* :: *LiteralTrail* **and** *conflictFlag* :: *bool* **and** *C* :: *Clause* **and** *literal* :: *Literal*
assumes *InvariantReasonClauses F M* **and** *InvariantCFalse conflictFlag M C* **and**
conflictFlag = *True* **and** *opposite literal el C* **and** ¬ *literal el* (*decisions M*)
shows ∃ *clause*. *formulaEntailsClause F clause* ∧ *isReason clause literal* (*elements M*)
proof−
from ‹*conflictFlag* = *True*› ‹*InvariantCFalse conflictFlag M C*›
have *clauseFalse C* (*elements M*)
unfolding *InvariantCFalse-def*
by *simp*
with ‹*opposite literal el C*›
have *literalTrue literal* (*elements M*)
by (*auto simp add*:*clauseFalseIffAllLiteralsAreFalse*)
with ‹¬ *literal el* (*decisions M*)› ‹*InvariantReasonClauses F M*›
show *?thesis*
unfolding *InvariantReasonClauses-def*
by *auto*
qed

4.4 Termination

In this section different ordering relations will be defined. These well-founded orderings will be the basic building blocks of termination orderings that will prove the termination of the SAT solving procedures

First we prove a simple lemma about acyclic orderings.

lemma *transIrreflexiveOrderingIsAcyclic*:
  assumes *trans r* **and** ∀ *x*. (*x*, *x*) ∉ *r*
  shows *acyclic r*
proof (*rule acyclicI*)
  {
    assume ∃ *x*. (*x*, *x*) ∈ *r*^+
    then obtain *x* **where** (*x*, *x*) ∈ *r*^+
      by *auto*
    moreover
    from ‹*trans r*›
    have *r*^+ = *r*
      by (*rule trancl-id*)
    ultimately
    have (*x*, *x*) ∈ *r*
      by *simp*
    with ‹∀ *x*. (*x*, *x*) ∉ *r*›
    have *False*
      by *simp*
  }
  thus ∀ *x*. (*x*, *x*) ∉ *r*^+
    by *auto*
qed

4.4.1 Trail ordering

We define a lexicographic ordering of trails, based on the number of literals on the different decision levels. It will be used for transition rules that change the trail, i.e., for *Decide*, *UnitPropagate*, *Backjump* and *Backtrack* transition rules.

definition
decisionLess = {(*l1*::(′*a*∗*bool*), *l2*::(′*a*∗*bool*)). *isDecision l1* ∧ ¬ *isDecision l2*}
definition
lexLess = {(*M1*::′*a Trail*, *M2*::′*a Trail*). (*M2*, *M1*) ∈ *lexord decisionLess*}

Following several lemmas will help prove that application of some DPLL-based transition rules decreases the trail in the *lexLess* ordering.

lemma *lexLessAppend*:

```
  assumes b ≠ []
  shows (a @ b, a) ∈ lexLess
proof−
  from ‹b ≠ []›
  have ∃ aa list. b = aa # list
    by (simp add: neq-Nil-conv)
  then obtain aa::'a × bool and list :: 'a Trail
    where b = aa # list
    by auto
  thus ?thesis
    unfolding lexLess-def
    unfolding lexord-def
    by simp
qed

lemma lexLessBackjump:
  assumes p = prefixToLevel level a and level >= 0 and level <
currentLevel a
  shows (p @ [(x, False)], a) ∈ lexLess
proof−
  from assms
  have ∃ rest. prefixToLevel level a @ rest = a ∧ rest ≠ [] ∧ isDecision
(hd rest)
    using isProperPrefixPrefixToLevel
    by auto
  with ‹p = prefixToLevel level a›
  obtain rest
    where p @ rest = a ∧ rest ≠ [] ∧ isDecision (hd rest)
    by auto
  thus ?thesis
    unfolding lexLess-def
    using lexord-append-left-rightI[of hd rest (x, False) decisionLess p
tl rest []]
    unfolding decisionLess-def
    by simp
qed

lemma lexLessBacktrack:
  assumes p = prefixBeforeLastDecision a decisions a ≠ []
  shows (p @ [(x, False)], a) ∈ lexLess
using assms
using prefixBeforeLastMarkedIsPrefixBeforeLastLevel[of a]
using lexLessBackjump[of p currentLevel a − 1 a]
unfolding currentLevel-def
by auto
```

The following several lemmas prover that *lexLess* is acyclic. This property will play an important role in building a well-founded ordering based on *lexLess*.

lemma *transDecisionLess*:
  shows *trans decisionLess*
proof−
  {
    fix x::(′*a*∗*bool*) **and** y::(′*a*∗*bool*) **and** z::(′*a*∗*bool*)
    assume $(x, y) \in$ *decisionLess*
    hence ¬ *isDecision y*
      unfolding *decisionLess-def*
      by *simp*
    moreover
    assume $(y, z) \in$ *decisionLess*
    hence *isDecision y*
      unfolding *decisionLess-def*
      by *simp*
    ultimately
    have *False*
      by *simp*
    hence $(x, z) \in$ *decisionLess*
      by *simp*
  }
  thus *?thesis*
    unfolding *trans-def*
    by *blast*
qed

lemma *translexLess*:
  shows *trans lexLess*
proof−
  {
    fix x :: ′*a Trail* **and** y :: ′*a Trail* **and** z :: ′*a Trail*
    assume $(x, y) \in$ *lexLess* **and** $(y, z) \in$ *lexLess*
    hence $(x, z) \in$ *lexLess*
      using *lexord-trans transDecisionLess*
      unfolding *lexLess-def*
      by *simp*
  }
  thus *?thesis*
    unfolding *trans-def*
    by *blast*
qed

lemma *irreflexiveDecisionLess*:
  shows $(x, x) \notin$ *decisionLess*
unfolding *decisionLess-def*
by *simp*

lemma *irreflexiveLexLess*:
  shows $(x, x) \notin$ *lexLess*

using *lexord-irreflexive*[*of decisionLess x*] *irreflexiveDecisionLess*
unfolding *lexLess-def*
by *auto*

lemma *acyclicLexLess*:
  shows *acyclic lexLess*
proof (*rule transIrreflexiveOrderingIsAcyclic*)
  show *trans lexLess*
    using *translexLess*
    .
  show ∀ *x*. (*x*, *x*) ∉ *lexLess*
    using *irreflexiveLexLess*
    by *auto*
qed

The *lexLess* ordering is not well-founded. In order to get a well-founded ordering, we restrict the *lexLess* ordering to cosistent and uniq trails with fixed variable set.

definition *lexLessRestricted* (*Vbl*::*Variable set*) == {(*M1*, *M2*).
 vars (*elements M1*) ⊆ *Vbl* ∧ *consistent* (*elements M1*) ∧ *uniq* (*elements M1*) ∧
 vars (*elements M2*) ⊆ *Vbl* ∧ *consistent* (*elements M2*) ∧ *uniq* (*elements M2*) ∧
  (*M1*, *M2*) ∈ *lexLess*}

First we show that the set of those trails is finite.

lemma *finiteVarsClause*:
  fixes *c* :: *Clause*
  shows *finite* (*vars c*)
by (*induct c*) *auto*

lemma *finiteVarsFormula*:
  fixes *F* :: *Formula*
  shows *finite* (*vars F*)
proof (*induct F*)
  case (*Cons c F*)
  thus *?case*
    using *finiteVarsClause*[*of c*]
    by *simp*
qed *simp*

lemma *finiteListDecompose*:
  shows *finite* {(*a*, *b*). *l* = *a* @ *b*}
proof (*induct l*)
  case *Nil*
  thus *?case*
    by *simp*
next
  case (*Cons x l'*)

```
thus ?case
proof−
  let ?S l = {(a, b). l = a @ b}
  let ?S' x l' = {(a', b). a' = [] ∧ b = (x # l') ∨
                            (∃ a. a' = x # a ∧ (a, b) ∈ (?S l'))}
  have ?S (x # l') = ?S' x l'
  proof
    show ?S (x # l') ⊆ ?S' x l'
    proof
      fix k
      assume k ∈ ?S (x # l')
      then obtain a and b
        where k = (a, b) x # l' = a @ b
        by auto
      then obtain a' where a' = x # a
        by auto
      from ‹k = (a, b)› ‹x # l' = a @ b›
      show k ∈ ?S' x l'
        using SimpleLevi[of a b x l']
        by auto
    qed
  next
    show ?S' x l' ⊆ ?S (x # l')
    proof
      fix k
      assume k ∈ ?S' x l'
      then obtain a' and b where
        k = (a', b) a' = [] ∧ b = x # l' ∨ (∃ a . a' = x # a ∧ (a, b)
∈ ?S l')
        by auto
      moreover
      {
        assume a' = [] b = x # l'
        with ‹k = (a', b)›
        have k ∈ ?S (x # l')
          by simp
      }
      moreover
      {
        assume ∃ a. a' = x # a ∧ (a, b) ∈ ?S l'
        then obtain a where
          a' = x # a ∧ (a, b) ∈ ?S l'
          by auto
        with ‹k = (a', b)›
        have k ∈ ?S (x # l')
          by auto
      }
      ultimately
      show k ∈ ?S (x # l')
```

```
          by auto
        qed
      qed
      moreover
      have ?S′ x l′ =
        {(a′, b). a′ = [] ∧ b = x # l′} ∪ {(a′, b). ∃ a. a′ = x # a ∧ (a,
b) ∈ ?S l′}
        by auto
      moreover
      have finite {(a′, b). ∃ a. a′ = x # a ∧ (a, b) ∈ ?S l′}
      proof−
        let ?h = λ (a, b). (x # a, b)
        have {(a′, b). ∃ a. a′ = x # a ∧ (a, b) ∈ ?S l′} = ?h ‘ {(a, b).
l′ = a @ b}
          by auto
        thus ?thesis
          using Cons(1)
          by auto
      qed
      moreover
      have finite {(a′, b). a′ = [] ∧ b = x # l′}
        by auto
      ultimately
      show ?thesis
        by auto
    qed
qed

lemma finiteListDecomposeSet:
  fixes L :: ′a list set
  assumes finite L
  shows finite {(a, b). ∃ l. l ∈ L ∧ l = a @ b}
proof−
  have {(a, b). ∃ l. l ∈ L ∧ l = a @ b} = (⋃ l ∈ L. {(a, b). l = a @
b})
    by auto
  moreover
  have finite (⋃ l ∈ L. {(a, b). l = a @ b})
  proof (rule finite-UN-I)
    from ‹finite L›
    show finite L
      .
  next
    fix l
    assume l ∈ L
    show finite {(a, b). l = a @ b}
      by (rule finiteListDecompose)
  qed
  ultimately
```

```
  show ?thesis
    by simp
qed

lemma finiteUniqAndConsistentTrailsWithGivenVariableSet:
  fixes V :: Variable set
  assumes finite V
   shows finite {(M::LiteralTrail). vars (elements M) = V ∧ uniq
(elements M) ∧ consistent (elements M)}
        (is finite (?trails V))
using assms
proof induct
  case empty
  thus ?case
  proof−
    have ?trails {} = {M. M = []} (is ?lhs = ?rhs)
    proof
      show ?lhs ⊆ ?rhs
      proof
        fix M::LiteralTrail
        assume M ∈ ?lhs
        hence M = []
          by (induct M) auto
        thus M ∈ ?rhs
          by simp
      qed
    next
      show ?rhs ⊆ ?lhs
      proof
        fix M::LiteralTrail
        assume M ∈ ?rhs
        hence M = []
          by simp
        thus M ∈ ?lhs
          by (induct M) auto
      qed
    qed
    moreover
    have finite {M. M = []}
      by auto
    ultimately
    show ?thesis
      by auto
  qed
next
  case (insert v V')
  thus ?case
  proof−
    let ?trails' V' = {(M::LiteralTrail). ∃ M' l d M''.
```

```
                              M = M' @ [(l, d)] @ M'' ∧
                              M' @ M'' ∈ (?trails V') ∧
                              l ∈ {Pos v, Neg v} ∧
                              d ∈ {True, False}}
  have ?trails (insert v V') = ?trails' V'
   (is ?lhs = ?rhs)
  proof
   show ?lhs ⊆ ?rhs
   proof
    fix M::LiteralTrail
    assume M ∈ ?lhs
     hence vars (elements M) = insert v V' uniq (elements M)
consistent (elements M)
     by auto
    hence v ∈ vars (elements M)
     by simp
    hence ∃ l. l el elements M ∧ var l = v
     by (induct M) auto
    then obtain l where l el elements M var l = v
     by auto
    hence ∃ M' M'' d. M = M' @ [(l, d)] @ M''
    proof (induct M)
     case (Cons m M1)
     thus ?case
     proof (cases l = (element m))
      case True
      then obtain d where m = (l, d)
       using eitherMarkedOrNotMarkedElement[of m]
       by auto
      hence m # M1 = [] @ [(l, d)] @ M1
       by simp
      then obtain M' M'' d where m # M1 = M' @ [(l, d)] @
M''
       ..
      thus ?thesis
       by auto
     next
      case False
      with ‹l el elements (m # M1)›
      have l el elements M1
       by simp
      with Cons(1) ‹var l = v›
      obtain M1' M'' d where M1 = M1' @ [(l, d)] @ M''
       by auto
      hence m # M1 = (m # M1') @ [(l, d)] @ M''
       by simp
      then obtain M' M'' d where m # M1 = M' @ [(l, d)] @
M''
       ..
```

thus *?thesis*
by *auto*
qed
qed *simp*
then obtain M' M'' d **where** $M = M'$ @ $[(l, d)]$ @ M''
by *auto*
moreover
from ‹*var* $l = v$›
have $l : \{Pos\ v, Neg\ v\}$
by (*cases* l) *auto*
moreover
have ∗: *vars* (*elements* (M' @ M'')) = *vars* (*elements* M') ∪ *vars* (*elements* M'')
using *varsAppendClauses*[*of elements* M' *elements* M'']
by *simp*
from ‹$M = M'$ @ $[(l, d)]$ @ M''› ‹*var* $l = v$›
have ∗∗: *vars* (*elements* M) = (*vars* (*elements* M')) ∪ $\{v\}$ ∪ (*vars* (*elements* M''))
using *varsAppendClauses*[*of elements* M' *elements* ([(l, d)] @ M'')]
using *varsAppendClauses*[*of elements* [(l, d)] *elements* M'']
by *simp*
have ∗∗∗: *vars* (*elements* M) = *vars* (*elements* (M' @ M'')) ∪ $\{v\}$
using ∗ ∗∗
by *simp*
have M' @ M'' ∈ (*?trails* V')
proof−
from ‹*uniq* (*elements* M)› ‹$M = M'$ @ $[(l, d)]$ @ M''›
have *uniq* (*elements* (M' @ M''))
by (*auto iff*: *uniqAppendIff*)
moreover
have *consistent* (*elements* (M' @ M''))
proof−
{
assume ¬ *consistent* (*elements* (M' @ M''))
then obtain l' **where** *literalTrue* l' (*elements* (M' @ M'')) *literalFalse* l' (*elements* (M' @ M''))
by (*auto simp add*:*inconsistentCharacterization*)
with ‹$M = M'$ @ $[(l, d)]$ @ M''›
have *literalTrue* l' (*elements* M) *literalFalse* l' (*elements* M)
by *auto*
hence ¬ *consistent* (*elements* M)
by (*auto simp add*: *inconsistentCharacterization*)
with ‹*consistent* (*elements* M)›
have *False*
by *simp*
}

```
    thus ?thesis
      by auto
  qed
  moreover
  have v ∉ vars (elements (M′ @ M′′))
  proof−
    {
      assume v ∈ vars (elements (M′ @ M′′))
      with ∗
      have v ∈ vars (elements M′) ∨ v ∈ vars (elements M′′)
        by simp
      moreover
      {
        assume v ∈ (vars (elements M′))
        hence ∃ l. var l = v ∧ l el elements M′
          by (induct M′) auto
        then obtain l′ where var l′ = v l′ el elements M′
          by auto
        from ‹var l = v› ‹var l′ = v›
        have l = l′ ∨ opposite l = l′
          using literalsWithSameVariableAreEqualOrOpposite[of
l l′]
          by simp
        moreover
        {
          assume l = l′
          with ‹l′ el elements M′› ‹M = M′ @ [(l, d)] @ M′′›
          have ¬ uniq (elements M)
            by (auto iff: uniqAppendIff)
          with ‹uniq (elements M)›
          have False
            by simp
        }
        moreover
        {
          assume opposite l = l′
          have ¬ consistent (elements M)
          proof−
            from ‹l′ el elements M′› ‹M = M′ @ [(l, d)] @ M′′›
            have literalTrue l′ (elements M)
              by simp
            moreover
             from ‹l′ el elements M′› ‹opposite l = l′› ‹M = M′
@ [(l, d)] @ M′′›
            have literalFalse l′ (elements M)
              by simp
            ultimately
            show ?thesis
              by (auto simp add: inconsistentCharacterization)
```

```
          qed
          with ‹consistent (elements M)›
          have False
            by simp
        }
        ultimately
        have False
          by auto
      }
      moreover
      {
        assume v ∈ (vars (elements M''))
        hence ∃ l. var l = v ∧ l el elements M''
          by (induct M'') auto
        then obtain l' where var l' = v l' el (elements M'')
          by auto
        from ‹var l = v› ‹var l' = v›
        have l = l' ∨ opposite l = l'
          using literalsWithSameVariableAreEqualOrOpposite[of
l l']
          by simp
        moreover
        {
          assume l = l'
          with ‹l' el elements M''› ‹M = M' @ [(l, d)] @ M''›
          have ¬ uniq (elements M)
            by (auto iff: uniqAppendIff)
          with ‹uniq (elements M)›
          have False
            by simp
        }
        moreover
        {
          assume opposite l = l'
          have ¬ consistent (elements M)
          proof−
            from ‹l' el elements M''› ‹M = M' @ [(l, d)] @ M''›
            have literalTrue l' (elements M)
              by simp
            moreover
            from ‹l' el elements M''› ‹opposite l = l'› ‹M = M'
@ [(l, d)] @ M''›
            have literalFalse l' (elements M)
              by simp
            ultimately
            show ?thesis
              by (auto simp add: inconsistentCharacterization)
          qed
          with ‹consistent (elements M)›
```

have *False*
by *simp*
}
ultimately
have *False*
by *auto*
}
ultimately
have *False*
by *auto*
}
thus *?thesis*
by *auto*
qed
from
$*$ $**$ $***$
‹$v \notin vars\ (elements\ (M'\ @\ M''))$›
‹$vars\ (elements\ M) = insert\ v\ V'$›
‹$\neg\ v \in V'$›
have $vars\ (elements\ (M'\ @\ M'')) = V'$
by (*auto simp del*: *vars-clause-def*)
ultimately
show *?thesis*
by *simp*
qed
ultimately
show $M \in ?rhs$
by *auto*
qed
next
show $?rhs \subseteq ?lhs$
proof
fix M :: *LiteralTrail*
assume $M \in ?rhs$
then obtain $M'\ M''\ l\ d$ **where**
$M = M'\ @\ [(l,\ d)]\ @\ M''$
$vars\ (elements\ (M'\ @\ M'')) = V'$
$uniq\ (elements\ (M'\ @\ M''))$ $consistent\ (elements\ (M'\ @\ M''))$
$l \in \{Pos\ v,\ Neg\ v\}$
by *auto*
from ‹$l \in \{Pos\ v,\ Neg\ v\}$›
have $var\ l = v$
by *auto*
have $*$: $vars\ (elements\ (M'\ @\ M'')) = vars\ (elements\ M') \cup vars\ (elements\ M'')$
using *varsAppendClauses*[*of elements* M' *elements* M'']
by *simp*
from ‹$var\ l = v$› ‹$M = M'\ @\ [(l,\ d)]\ @\ M''$›
have $**$: $vars\ (elements\ M) = vars\ (elements\ M') \cup \{v\} \cup vars$

(*elements* M'')
using *varsAppendClauses*[*of elements* M' *elements* ([(*l*, *d*)] @ M'')]
using *varsAppendClauses*[*of elements* [(*l*, *d*)] *elements* M'']
by *simp*
from * ** ‹*vars* (*elements* (M' @ M'')) = V'›
have *vars* (*elements* *M*) = *insert v* V'
by (*auto simp del*: *vars-clause-def*)
moreover
from *
‹*var l* = *v*›
‹$v \notin V'$›
‹*vars* (*elements* (M' @ M'')) = V'›
have *var l* ∉ *vars* (*elements* M') *var l* ∉ *vars* (*elements* M'')
by *auto*
from ‹*var l* ∉ *vars* (*elements* M')›
have ¬ *literalTrue l* (*elements* M') ¬ *literalFalse l* (*elements* M')
using *valuationContainsItsLiteralsVariable*[*of l elements* M']
using *valuationContainsItsLiteralsVariable*[*of opposite l elements* M']
by *auto*
from ‹*var l* ∉ *vars* (*elements* M'')›
have ¬ *literalTrue l* (*elements* M'') ¬ *literalFalse l* (*elements* M'')
using *valuationContainsItsLiteralsVariable*[*of l elements* M'']
using *valuationContainsItsLiteralsVariable*[*of opposite l elements* M'']
by *auto*
have *uniq* (*elements* *M*)
using ‹*M* = M' @ [(*l*, *d*)] @ M''› ‹*uniq* (*elements* (M' @ M''))›
‹¬ *literalTrue l* (*elements* M'')› ‹¬ *literalFalse l* (*elements* M'')›
‹¬ *literalTrue l* (*elements* M')› ‹¬ *literalFalse l* (*elements* M')›
by (*auto iff*: *uniqAppendIff*)
moreover
have *consistent* (*elements* *M*)
proof−
{
assume ¬ *consistent* (*elements* *M*)
then obtain l' **where** *literalTrue* l' (*elements* *M*) *literalFalse* l' (*elements* *M*)
by (*auto simp add*: *inconsistentCharacterization*)
have *False*
proof (*cases* $l' = l$)
case *True*
with ‹*literalFalse* l' (*elements* *M*)› ‹*M* = M' @ [(*l*, *d*)] @

M''›
have *literalFalse* l' (*elements* (M' @ M''))
using *oppositeIsDifferentFromLiteral*[*of* l]
by (*auto split*: *if-split-asm*)
with ‹¬ *literalFalse* l (*elements* M')› ‹¬ *literalFalse* l (*elements* M'')› ‹$l' = l$›
show *?thesis*
by *auto*
next
case *False*
with ‹*literalTrue* l' (*elements* M)› ‹$M = M'$ @ [(l, d)] @ M''›
have *literalTrue* l' (*elements* (M' @ M''))
by (*auto split*: *if-split-asm*)
with ‹*consistent* (*elements* (M' @ M''))›
have ¬ *literalFalse* l' (*elements* (M' @ M''))
by (*auto simp add*: *inconsistentCharacterization*)
with ‹*literalFalse* l' (*elements* M)› ‹$M = M'$ @ [(l, d)] @ M''›
have *opposite* $l' = l$
by (*auto split*: *if-split-asm*)
with ‹*var* $l = v$›
have *var* $l' = v$
by *auto*
with ‹*literalTrue* l' (*elements* (M' @ M''))› ‹*vars* (*elements* (M' @ M'')) = V'›
have $v \in V'$
using *valuationContainsItsLiteralsVariable*[*of* l' *elements* (M' @ M'')]
by *simp*
with ‹$v \notin V'$›
show *?thesis*
by *simp*
qed
}
thus *?thesis*
by *auto*
qed
ultimately
show $M \in$ *?lhs*
by *auto*
qed
qed
moreover
let *?f* = λ ((M', M''), l, d). M' @ [(l, d)] @ M''
let *?Mset* = {(M', M''). M' @ M'' ∈ *?trails* V'}
let *?lSet* = {*Pos* v, *Neg* v}
let *?dSet* = {*True*, *False*}
have *?trails'* V' = *?f* ‘ (*?Mset* × *?lSet* × *?dSet*) (**is** *?lhs* = *?rhs*)

```
    proof
      show ?lhs ⊆ ?rhs
      proof
        fix M :: LiteralTrail
        assume M ∈ ?lhs
        then obtain M' M'' l d
          where P: M = M' @ [(l, d)] @ M'' M' @ M'' ∈ (?trails V')
l ∈ {Pos v, Neg v} d ∈ {True, False}
          by auto
        show M ∈ ?rhs
        proof
          from P
          show M = ?f ((M', M''), l, d)
            by simp
        next
          from P
          show ((M', M''), l, d) ∈ ?Mset × ?lSet × ?dSet
            by auto
        qed
      qed
    next
      show ?rhs ⊆ ?lhs
      proof
        fix M::LiteralTrail
        assume M ∈ ?rhs
        then obtain p l d where P: M = ?f (p, l, d) p ∈ ?Mset l ∈
?lSet d ∈ ?dSet
          by auto
        from ‹p ∈ ?Mset›
        obtain M' M'' where M' @ M'' ∈ ?trails V'
          by auto
        thus M ∈ ?lhs
          using P
          by auto
      qed
    qed
    moreover
    have ?Mset = {(M', M''). ∃ l. l ∈ ?trails V' ∧ l = M' @ M''}
      by auto
    hence finite ?Mset
      using insert(3)
      using finiteListDecomposeSet[of ?trails V']
      by simp
    ultimately
    show ?thesis
      by auto
  qed
qed
```

lemma *finiteUniqAndConsistentTrailsWithGivenVariableSuperset*:
fixes *V* :: *Variable set*
assumes *finite V*
shows *finite* {(*M*::*LiteralTrail*). *vars* (*elements M*) ⊆ *V* ∧ *uniq* (*elements M*) ∧ *consistent* (*elements M*)} (**is** *finite* (*?trails V*))
proof−
have {*M*. *vars* (*elements M*) ⊆ *V* ∧ *uniq* (*elements M*) ∧ *consistent* (*elements M*)} =
(⋃ *v* ∈ *Pow V*.{*M*. *vars* (*elements M*) = *v* ∧ *uniq* (*elements M*) ∧ *consistent* (*elements M*)})
by *auto*
moreover
have *finite* (⋃ *v* ∈ *Pow V*.{*M*. *vars* (*elements M*) = *v* ∧ *uniq* (*elements M*) ∧ *consistent* (*elements M*)})
proof (*rule finite-UN-I*)
from ‹*finite V*›
show *finite* (*Pow V*)
by *simp*
next
fix *v*
assume *v* ∈ *Pow V*
with ‹*finite V*›
have *finite v*
by (*auto simp add*: *finite-subset*)
thus *finite* {*M*. *vars* (*elements M*) = *v* ∧ *uniq* (*elements M*) ∧ *consistent* (*elements M*)}
using *finiteUniqAndConsistentTrailsWithGivenVariableSet*[*of v*]
by *simp*
qed
ultimately
show *?thesis*
by *simp*
qed

Since the restricted ordering is acyclic and its domain is finite, it has to be well-founded.

lemma *wfLexLessRestricted*:
assumes *finite Vbl*
shows *wf* (*lexLessRestricted Vbl*)
proof (*rule finite-acyclic-wf*)
show *finite* (*lexLessRestricted Vbl*)
proof−
let *?X* = {(*M1*, *M2*).
consistent (*elements M1*) ∧ *uniq* (*elements M1*) ∧ *vars* (*elements M1*) ⊆ *Vbl* ∧
consistent (*elements M2*) ∧ *uniq* (*elements M2*) ∧ *vars* (*elements M2*) ⊆ *Vbl*}
let *?Y* = {*M*. *vars* (*elements M*) ⊆ *Vbl* ∧ *uniq* (*elements M*) ∧ *consistent* (*elements M*)}

have *?X* = *?Y* × *?Y*
by *auto*
moreover
have *finite ?Y*
using *finiteUniqAndConsistentTrailsWithGivenVariableSuperset*[*of Vbl*]
‹*finite Vbl*›
by *auto*
ultimately
have *finite ?X*
by *simp*
moreover
have *lexLessRestricted Vbl* ⊆ *?X*
unfolding *lexLessRestricted-def*
by *auto*
ultimately
show *?thesis*
by (*simp add*: *finite-subset*)
qed
next
show *acyclic* (*lexLessRestricted Vbl*)
proof−
{
assume ¬ *?thesis*
then obtain *x* **where** (*x*, *x*) ∈ (*lexLessRestricted Vbl*)^+
unfolding *acyclic-def*
by *auto*
have *lexLessRestricted Vbl* ⊆ *lexLess*
unfolding *lexLessRestricted-def*
by *auto*
have (*lexLessRestricted Vbl*)^+ ⊆ *lexLess*^+
proof
fix *a*
assume *a* ∈ (*lexLessRestricted Vbl*)^+
with ‹*lexLessRestricted Vbl* ⊆ *lexLess*›
show *a* ∈ *lexLess*^+
using *trancl-mono*[*of a lexLessRestricted Vbl lexLess*]
by *blast*
qed
with ‹(*x*, *x*) ∈ (*lexLessRestricted Vbl*)^+›
have (*x*, *x*) ∈ *lexLess*^+
by *auto*
moreover
have *trans lexLess*
using *translexLess*
.
hence *lexLess*^+ = *lexLess*
by (*rule trancl-id*)
ultimately

```
      have (x, x) ∈ lexLess
        by auto
      with irreflexiveLexLess[of x]
      have False
        by simp
    }
    thus ?thesis
      by auto
  qed
qed
```

lexLessRestricted is also transitive.

```
lemma transLexLessRestricted:
  shows trans (lexLessRestricted Vbl)
proof−
  {
    fix x::LiteralTrail and y::LiteralTrail and z::LiteralTrail
    assume (x, y) ∈ lexLessRestricted Vbl (y, z) ∈ lexLessRestricted
Vbl
    hence (x, z) ∈ lexLessRestricted Vbl
      unfolding lexLessRestricted-def
      using translexLess
      unfolding trans-def
      by auto
  }
  thus ?thesis
    unfolding trans-def
    by blast
qed
```

4.4.2 Conflict clause ordering

The ordering of conflict clauses is the multiset ordering induced by the ordering of elements in the trail. Since, resolution operator is defined so that it removes all occurrences of clashing literal, it is also neccessary to remove duplicate literals before comparison.

```
definition
multLess M = inv-image  (mult (precedesOrder (elements M))) (λ x.
mset (remdups (oppositeLiteralList x)))
```

The following lemma will help prove that application of the *Explain* DPLL transition rule decreases the conflict clause in the *multLess* ordering.

```
lemma multLessResolve:
  assumes
  opposite l el C and
  isReason reason l (elements M)
```

```
  shows
  (resolve C reason (opposite l), C) ∈ multLess M
proof−
  let ?X = mset (remdups (oppositeLiteralList C))
  let ?Y = mset (remdups (oppositeLiteralList (resolve C reason (opposite
l))))
  let ?ord = precedesOrder (elements M)
  have (?Y, ?X) ∈ (mult1 ?ord)
  proof−
    let ?Z = mset (remdups (oppositeLiteralList (removeAll (opposite
l) C)))
    let ?W = mset (remdups (oppositeLiteralList (removeAll l (list-diff
reason C))))
    let ?a = l
    from ‹(opposite l) el C›
    have ?X = ?Z + {#?a#}
      using removeAll-multiset[of remdups (oppositeLiteralList C) l]
      using oppositeLiteralListRemove[of opposite l C]
      using literalElListIffOppositeLiteralElOppositeLiteralList[of l op-
positeLiteralList C]
      by auto
    moreover
    have ?Y = ?Z + ?W
    proof−
    have list-diff (oppositeLiteralList (removeAll l reason)) (oppositeLiteralList
(removeAll (opposite l) C)) =
        oppositeLiteralList (removeAll l (list-diff reason C))
      proof−
        from ‹isReason reason l (elements M)›
        have opposite l ∉ set (removeAll l reason)
          unfolding isReason-def
          by auto

        hence list-diff (removeAll l reason) (removeAll (opposite l) C)
= list-diff (removeAll l reason) C
          using listDiffRemoveAllNonMember[of opposite l removeAll l
reason C]
          by simp
        thus ?thesis
          unfolding oppositeLiteralList-def
          using listDiffMap[of opposite removeAll l reason removeAll
(opposite l) C]
          by auto
      qed
      thus ?thesis
        unfolding resolve-def
        using remdupsAppendMultiSet[of oppositeLiteralList (removeAll
(opposite l) C) oppositeLiteralList (removeAll l reason)]
        unfolding oppositeLiteralList-def
```

```
        by auto
    qed
    moreover
    have ∀ b. b ∈# ?W ⟶ (b, ?a) ∈ ?ord
    proof−
      {
        fix b
        assume b ∈# ?W
        hence opposite b ∈ set (removeAll l reason)
        proof−
          from ‹b ∈# ?W›
           have b el remdups (oppositeLiteralList (removeAll l (list-diff
reason C)))
            by simp
          hence opposite b el removeAll l (list-diff reason C)
            using literalElListIffOppositeLiteralElOppositeLiteralList[of
opposite b removeAll l (list-diff reason C)]
            by auto
          hence opposite b el list-diff (removeAll l reason) C
            by simp
          thus ?thesis
            using listDiffIff[of opposite b removeAll l reason C]
            by simp
        qed
        with ‹isReason reason l (elements M)›
        have precedes b l (elements M) b ≠ l
          unfolding isReason-def
          unfolding precedes-def
          by auto
        hence (b, ?a) ∈ ?ord
          unfolding precedesOrder-def
          by simp
      }
      thus ?thesis
        by auto
    qed
    ultimately
    have ∃ a M0 K. ?X = M0 + {#a#} ∧ ?Y = M0 + K ∧ (∀ b. b
∈# K ⟶ (b, a) ∈ ?ord)
      by blast
    thus ?thesis
      unfolding mult1-def
      by auto
  qed
  hence (?Y, ?X) ∈ (mult1 ?ord)⁺
    by simp
  thus ?thesis
    unfolding multLess-def
    unfolding mult-def
```

unfolding *inv-image-def*
by *auto*
qed

lemma *multLessListDiff*:
assumes
(*a*, *b*) ∈ *multLess M*
shows
(*list-diff a x*, *b*) ∈ *multLess M*
proof−
let *?pOrd* = *precedesOrder* (*elements M*)
let *?f* = λ *l*. *remdups* (*map opposite l*)
have *trans ?pOrd*
using *transPrecedesOrder*[*of elements M*]
by *simp*

have (*mset* (*?f a*), *mset* (*?f b*)) ∈ *mult ?pOrd*
using *assms*
unfolding *multLess-def*
unfolding *oppositeLiteralList-def*
by *simp*
moreover
have *multiset-le* (*mset* (*list-diff* (*?f a*) (*?f x*)))
(*mset* (*?f a*))
?pOrd
using ‹*trans ?pOrd*›
using *multisetLeListDiff*[*of ?pOrd ?f a ?f x*]
by *simp*
ultimately
have (*mset* (*list-diff* (*?f a*) (*?f x*)), *mset* (*?f b*)) ∈ *mult ?pOrd*
unfolding *multiset-le-def*
unfolding *mult-def*
by *auto*

thus *?thesis*
unfolding *multLess-def*
unfolding *oppositeLiteralList-def*
by (*simp add*: *listDiffMap remdupsListDiff*)
qed

lemma *multLessRemdups*:
assumes
(*a*, *b*) ∈ *multLess M*
shows
(*remdups a*, *remdups b*) ∈ *multLess M* ∧
(*remdups a*, *b*) ∈ *multLess M* ∧
(*a*, *remdups b*) ∈ *multLess M*
proof−
{

fix *l*
have *remdups* (*map opposite l*) = *remdups* (*map opposite* (*remdups l*))
by (*induct l*) *auto*
}
thus *?thesis*
using *assms*
unfolding *multLess-def*
unfolding *oppositeLiteralList-def*
by *simp*
qed

Now we show that *multLess* is well-founded.

lemma *wfMultLess*:
shows *wf* (*multLess M*)
proof−
have *wf* (*precedesOrder* (*elements M*))
by (*simp add*: *wellFoundedPrecedesOrder*)
hence *wf* (*mult* (*precedesOrder* (*elements M*)))
by (*simp add*: *wf-mult*)
thus *?thesis*
unfolding *multLess-def*
using *wf-inv-image*[*of* (*mult* (*precedesOrder* (*elements M*)))]
by *auto*
qed

4.4.3 ConflictFlag ordering

A trivial ordering on Booleans. It will be used for the *Conflict* transition rule.

definition
boolLess = {(*True*, *False*)}

We show that it is well-founded

lemma *transBoolLess*:
shows *trans boolLess*
proof−
{
fix *x*::*bool* **and** *y*::*bool* **and** *z*::*bool*
assume (*x*, *y*) ∈ *boolLess*
hence *x* = *True y* = *False*
unfolding *boolLess-def*
by *auto*
assume (*y*, *z*) ∈ *boolLess*
hence *y* = *True z* = *False*
unfolding *boolLess-def*
by *auto*
from ‹*y* = *False*› ‹*y* = *True*›

have *False*
  by *simp*
 hence $(x, z) \in$ *boolLess*
  by *simp*
}
thus *?thesis*
 unfolding *trans-def*
 by *blast*
qed

lemma *wfBoolLess*:
 shows *wf boolLess*
proof (*rule finite-acyclic-wf*)
 show *finite boolLess*
  unfolding *boolLess-def*
  by *simp*
next
 have *boolLess*^+ = *boolLess*
  using *transBoolLess*
  by *simp*
 thus *acyclic boolLess*
  unfolding *boolLess-def*
  unfolding *acyclic-def*
  by *auto*
qed

4.4.4 Formulae ordering

A partial ordering of formulae, based on a membersip of a single fixed clause. This ordering will be used for the *Learn* transtion rule.

definition *learnLess* (*C*::*Clause*) == {((*F1*::*Formula*), (*F2*::*Formula*)). *C el F1* ∧ ¬ *C el F2*}

We show that it is well founded

lemma *wfLearnLess*:
 fixes *C*::*Clause*
 shows *wf* (*learnLess C*)
unfolding *wf-eq-minimal*
proof−
 show ∀ *Q F*. *F* ∈ *Q* ⟶ (∃ *Fmin*∈*Q*. ∀ *F'*. (*F'*, *Fmin*) ∈ *learnLess C* ⟶ *F'* ∉ *Q*)
 proof−
  {
   fix *F*::*Formula* **and** *Q*::*Formula set*
   assume *F* ∈ *Q*
   have ∃ *Fmin*∈*Q*. ∀ *F'*. (*F'*, *Fmin*) ∈ *learnLess C* ⟶ *F'* ∉ *Q*
   proof (*cases* ∃ *Fc* ∈ *Q*. *C el Fc*)

```
      case True
      then obtain Fc where Fc ∈ Q C el Fc
        by auto
      have ∀ F'. (F', Fc) ∈ learnLess C ⟶ F' ∉ Q
      proof
        fix F'
        show (F', Fc) ∈ learnLess C ⟶ F' ∉ Q
        proof
          assume (F', Fc) ∈ learnLess C
          hence ¬ C el Fc
            unfolding learnLess-def
            by auto
          with ‹C el Fc› have False
            by simp
          thus F' ∉ Q
            by simp
        qed
      qed
      with ‹Fc ∈ Q›
      show ?thesis
        by auto
    next
      case False
      have ∀ F'. (F', F) ∈ learnLess C ⟶ F' ∉ Q
      proof
        fix F'
        show (F', F) ∈ learnLess C ⟶ F' ∉ Q
        proof
          assume (F', F) ∈ learnLess C
          hence C el F'
            unfolding learnLess-def
            by simp
          with False
          show F' ∉ Q
            by auto
        qed
      qed
      with ‹F ∈ Q›
      show ?thesis
        by auto
    qed
  }
  thus ?thesis
    by auto
 qed
qed
```

4.4.5 Properties of well-founded relations.

```
lemma wellFoundedEmbed:
  fixes rel :: ('a × 'a) set and rel' :: ('a × 'a) set
  assumes ∀ x y. (x, y) ∈ rel ⟶ (x, y) ∈ rel' and wf rel'
  shows wf rel
unfolding wf-eq-minimal
proof−
  show ∀ Q x. x ∈ Q ⟶ (∃ zmin∈Q. ∀ z. (z, zmin) ∈ rel ⟶ z ∉ Q)
  proof−
    {
      fix x::'a and Q::'a set
      assume x ∈ Q
      have ∃ zmin∈Q. ∀ z. (z, zmin) ∈ rel ⟶ z ∉ Q
      proof−
        from ‹wf rel'› ‹x ∈ Q›
        obtain zmin::'a
          where zmin ∈ Q and ∀ z. (z, zmin) ∈ rel' ⟶ z ∉ Q
          unfolding wf-eq-minimal
          by auto
        {
          fix z::'a
          assume (z, zmin) ∈ rel
          have z ∉ Q
          proof−
           from ‹∀ x y. (x, y) ∈ rel ⟶ (x, y) ∈ rel'› ‹(z, zmin) ∈ rel›
            have (z, zmin) ∈ rel'
              by simp
            with ‹∀ z. (z, zmin) ∈ rel' ⟶ z ∉ Q›
            show ?thesis
              by simp
          qed
        }
        with ‹zmin ∈ Q›
        show ?thesis
          by auto
      qed
    }
    thus ?thesis
      by auto
  qed
qed

end
```

5 BasicDPLL

```
theory BasicDPLL
imports SatSolverVerification
```

begin

This theory formalizes the transition rule system BasicDPLL which is based on the classical DPLL procedure, but does not use the PureLiteral rule.

5.1 Specification

The state of the procedure is uniquely determined by its trail.

record *State* =
getM :: *LiteralTrail*

Procedure checks the satisfiability of the formula F0 which does not change during the solving process. An external parameter is the set *decisionVars* which are the variables that branching is performed on. Usually this set contains all variables of the formula F0, but that does not always have to be the case.

Now we define the transition rules of the system

definition
appliedDecide:: *State* ⇒ *State* ⇒ *Variable set* ⇒ *bool*
where
appliedDecide stateA stateB decisionVars ==
∃ *l*.
(*var l*) ∈ *decisionVars* ∧
¬ *l el* (*elements* (*getM stateA*)) ∧
¬ *opposite l el* (*elements* (*getM stateA*)) ∧

getM stateB = *getM stateA* @ [(*l*, *True*)]

definition
applicableDecide :: *State* ⇒ *Variable set* ⇒ *bool*
where
applicableDecide state decisionVars == ∃ *state'*. *appliedDecide state state' decisionVars*

definition
appliedUnitPropagate :: *State* ⇒ *State* ⇒ *Formula* ⇒ *bool*
where
appliedUnitPropagate stateA stateB F0 ==
∃ (*uc*::*Clause*) (*ul*::*Literal*).
uc el F0 ∧
isUnitClause uc ul (*elements* (*getM stateA*)) ∧

getM stateB = *getM stateA* @ [(*ul*, *False*)]

definition
applicableUnitPropagate :: *State* ⇒ *Formula* ⇒ *bool*

where
applicableUnitPropagate state F0 == ∃ *state′*. *appliedUnitPropagate state state′ F0*

definition
appliedBacktrack :: *State* ⇒ *State* ⇒ *Formula* ⇒ *bool*
where
appliedBacktrack stateA stateB F0 ==
formulaFalse F0 (*elements* (*getM stateA*)) ∧
decisions (*getM stateA*) ≠ [] ∧

getM stateB = *prefixBeforeLastDecision* (*getM stateA*) @ [(*opposite* (*lastDecision* (*getM stateA*)), *False*)]

definition
applicableBacktrack :: *State* ⇒ *Formula* ⇒ *bool*
where
applicableBacktrack state F0 == ∃ *state′*. *appliedBacktrack state state′ F0*

Solving starts with the empty trail.

definition
isInitialState :: *State* ⇒ *Formula* ⇒ *bool*
where
isInitialState state F0 ==
getM state = []

Transitions are preformed only by using one of the three given rules.

definition
transition stateA stateB F0 decisionVars ==
appliedDecide stateA stateB decisionVars ∨
appliedUnitPropagate stateA stateB F0 ∨
appliedBacktrack stateA stateB F0

Transition relation is obtained by applying transition rules iteratively. It is defined using a reflexive-transitive closure.

definition
transitionRelation F0 decisionVars == ({(*stateA*, *stateB*). *transition stateA stateB F0 decisionVars*})^*

Final state is one in which no rules apply

definition
isFinalState :: *State* ⇒ *Formula* ⇒ *Variable set* ⇒ *bool*
where
isFinalState state F0 decisionVars == ¬ (∃ *state′*. *transition state state′ F0 decisionVars*)

The following several lemmas give conditions for applicability of different rules.

lemma *applicableDecideCharacterization*:
  fixes *stateA*::*State*
  shows *applicableDecide stateA decisionVars* =
  (∃ *l*.
        (*var l*) ∈ *decisionVars* ∧
        ¬ *l el* (*elements* (*getM stateA*)) ∧
        ¬ *opposite l el* (*elements* (*getM stateA*)))
   (**is** *?lhs* = *?rhs*)
proof
  assume *?rhs*
  then obtain *l* **where**
    ∗: (*var l*) ∈ *decisionVars* ¬ *l el* (*elements* (*getM stateA*)) ¬ *opposite l el* (*elements* (*getM stateA*))
    unfolding *applicableDecide-def*
    by *auto*
  let *?stateB* = *stateA*⦇ *getM* := (*getM stateA*) @ [(*l*, *True*)] ⦈
  from ∗ **have** *appliedDecide stateA ?stateB decisionVars*
    unfolding *appliedDecide-def*
    by *auto*
  thus *?lhs*
    unfolding *applicableDecide-def*
    by *auto*
next
  assume *?lhs*
  then obtain *stateB l*
    where (*var l*) ∈ *decisionVars* ¬ *l el* (*elements* (*getM stateA*))
    ¬ *opposite l el* (*elements* (*getM stateA*))
    unfolding *applicableDecide-def*
    unfolding *appliedDecide-def*
    by *auto*
  thus *?rhs*
    by *auto*
qed

lemma *applicableUnitPropagateCharacterization*:
  fixes *stateA*::*State* **and** *F0*::*Formula*
  shows *applicableUnitPropagate stateA F0* =
  (∃ (*uc*::*Clause*) (*ul*::*Literal*).
       uc el F0 ∧
       isUnitClause uc ul (*elements* (*getM stateA*)))
   (**is** *?lhs* = *?rhs*)
proof
  assume *?rhs*
  then obtain *ul uc*
    where ∗: *uc el F0 isUnitClause uc ul* (*elements* (*getM stateA*))
    unfolding *applicableUnitPropagate-def*
    by *auto*

let *?stateB* = *stateA*⦇ *getM* := *getM stateA* @ [(*ul*, *False*)] ⦈
from ∗ **have** *appliedUnitPropagate stateA ?stateB F0*
unfolding *appliedUnitPropagate-def*
by *auto*
thus *?lhs*
unfolding *applicableUnitPropagate-def*
by *auto*
next
assume *?lhs*
then obtain *stateB uc ul*
where *uc el F0 isUnitClause uc ul* (*elements* (*getM stateA*))
unfolding *applicableUnitPropagate-def*
unfolding *appliedUnitPropagate-def*
by *auto*
thus *?rhs*
by *auto*
qed

lemma *applicableBacktrackCharacterization*:
fixes *stateA*::*State*
shows *applicableBacktrack stateA F0* =
(*formulaFalse F0* (*elements* (*getM stateA*)) ∧
decisions (*getM stateA*) ≠ []) (**is** *?lhs* = *?rhs*)
proof
assume *?rhs*
hence ∗: *formulaFalse F0* (*elements* (*getM stateA*)) *decisions* (*getM stateA*) ≠ []
by *auto*
let *?stateB* = *stateA*⦇ *getM* := *prefixBeforeLastDecision* (*getM stateA*) @ [(*opposite* (*lastDecision* (*getM stateA*)), *False*)]⦈
from ∗ **have** *appliedBacktrack stateA ?stateB F0*
unfolding *appliedBacktrack-def*
by *auto*
thus *?lhs*
unfolding *applicableBacktrack-def*
by *auto*
next
assume *?lhs*
then obtain *stateB*
where *appliedBacktrack stateA stateB F0*
unfolding *applicableBacktrack-def*
by *auto*
hence
formulaFalse F0 (*elements* (*getM stateA*))
decisions (*getM stateA*) ≠ []
getM stateB = *prefixBeforeLastDecision* (*getM stateA*) @ [(*opposite* (*lastDecision* (*getM stateA*)), *False*)]
unfolding *appliedBacktrack-def*
by *auto*

thus *?rhs*
by *auto*
qed

Final states are the ones where no rule is applicable.

lemma *finalStateNonApplicable*:
fixes *state*::*State*
shows *isFinalState state F0 decisionVars* =
(¬ *applicableDecide state decisionVars* ∧
¬ *applicableUnitPropagate state F0* ∧
¬ *applicableBacktrack state F0*)
unfolding *isFinalState-def*
unfolding *transition-def*
unfolding *applicableDecide-def*
unfolding *applicableUnitPropagate-def*
unfolding *applicableBacktrack-def*
by *auto*

5.2 Invariants

Invariants that are relevant for the rest of correctness proof.

definition
invariantsHoldInState :: *State* ⇒ *Formula* ⇒ *Variable set* ⇒ *bool*
where
invariantsHoldInState state F0 decisionVars ==
InvariantImpliedLiterals F0 (*getM state*) ∧
InvariantVarsM (*getM state*) *F0 decisionVars* ∧
InvariantConsistent (*getM state*) ∧
InvariantUniq (*getM state*)

Invariants hold in initial states.

lemma *invariantsHoldInInitialState*:
fixes *state* :: *State* **and** *F0* :: *Formula*
assumes *isInitialState state F0*
shows *invariantsHoldInState state F0 decisionVars*
using *assms*
by (*auto simp add*:
isInitialState-def
invariantsHoldInState-def
InvariantImpliedLiterals-def
InvariantVarsM-def
InvariantConsistent-def
InvariantUniq-def
)

Valid transitions preserve invariants.

lemma *transitionsPreserveInvariants*:

fixes *stateA*::*State* **and** *stateB*::*State*
assumes *transition stateA stateB F0 decisionVars* **and**
invariantsHoldInState stateA F0 decisionVars
shows *invariantsHoldInState stateB F0 decisionVars*
proof−
from ‹*invariantsHoldInState stateA F0 decisionVars*›
have
InvariantImpliedLiterals F0 (*getM stateA*) **and**
InvariantVarsM (*getM stateA*) *F0 decisionVars* **and**
InvariantConsistent (*getM stateA*) **and**
InvariantUniq (*getM stateA*)
unfolding *invariantsHoldInState-def*
by *auto*
{
assume *appliedDecide stateA stateB decisionVars*
then obtain *l*::*Literal* **where**
(*var l*) ∈ *decisionVars*
¬ *literalTrue l* (*elements* (*getM stateA*))
¬ *literalFalse l* (*elements* (*getM stateA*))
getM stateB = *getM stateA* @ [(*l*, *True*)]
unfolding *appliedDecide-def*
by *auto*

from ‹¬ *literalTrue l* (*elements* (*getM stateA*))› ‹¬ *literalFalse l* (*elements* (*getM stateA*))›
have ∗: *var l* ∉ *vars* (*elements* (*getM stateA*))
using *variableDefinedImpliesLiteralDefined*[*of l elements* (*getM stateA*)]
by *simp*

have *InvariantImpliedLiterals F0* (*getM stateB*)
using
‹*getM stateB* = *getM stateA* @ [(*l*, *True*)]›
‹*InvariantImpliedLiterals F0* (*getM stateA*)›
‹*InvariantUniq* (*getM stateA*)›
‹*var l* ∉ *vars* (*elements* (*getM stateA*))›
InvariantImpliedLiteralsAfterDecide[*of F0 getM stateA l getM stateB*]
by *simp*
moreover
have *InvariantVarsM* (*getM stateB*) *F0 decisionVars*
using ‹*getM stateB* = *getM stateA* @ [(*l*, *True*)]›
‹*InvariantVarsM* (*getM stateA*) *F0 decisionVars*›
‹*var l* ∈ *decisionVars*›
InvariantVarsMAfterDecide[*of getM stateA F0 decisionVars l getM stateB*]
by *simp*
moreover
have *InvariantConsistent* (*getM stateB*)

using ‹*getM stateB* = *getM stateA* @ [(*l*, *True*)]›
‹*InvariantConsistent* (*getM stateA*)›
‹*var l* ∉ *vars* (*elements* (*getM stateA*))›
InvariantConsistentAfterDecide[*of getM stateA l getM stateB*]
by *simp*
moreover
have *InvariantUniq* (*getM stateB*)
using ‹*getM stateB* = *getM stateA* @ [(*l*, *True*)]›
‹*InvariantUniq* (*getM stateA*)›
‹*var l* ∉ *vars* (*elements* (*getM stateA*))›
InvariantUniqAfterDecide[*of getM stateA l getM stateB*]
by *simp*
ultimately
have *?thesis*
unfolding *invariantsHoldInState-def*
by *auto*
}
moreover
{
assume *appliedUnitPropagate stateA stateB F0*
then obtain *uc*::*Clause* **and** *ul*::*Literal* **where**
uc el F0
isUnitClause uc ul (*elements* (*getM stateA*))
getM stateB = *getM stateA* @ [(*ul*, *False*)]
unfolding *appliedUnitPropagate-def*
by *auto*

from ‹*isUnitClause uc ul* (*elements* (*getM stateA*))›
have *ul el uc*
unfolding *isUnitClause-def*
by *simp*

from ‹*uc el F0*›
have *formulaEntailsClause F0 uc*
by (*simp add*: *formulaEntailsItsClauses*)

have *InvariantImpliedLiterals F0* (*getM stateB*)
using
‹*InvariantImpliedLiterals F0* (*getM stateA*)›
‹*formulaEntailsClause F0 uc*›
‹*isUnitClause uc ul* (*elements* (*getM stateA*))›
‹*getM stateB* = *getM stateA* @ [(*ul*, *False*)]›
InvariantImpliedLiteralsAfterUnitPropagate[*of F0 getM stateA uc ul getM stateB*]
by *simp*
moreover
from ‹*ul el uc*› ‹*uc el F0*›
have *ul el F0*
by (*auto simp add*: *literalElFormulaCharacterization*)

```
    hence var ul ∈ vars F0 ∪ decisionVars
      using formulaContainsItsLiteralsVariable [of ul F0]
      by auto

    have InvariantVarsM (getM stateB) F0 decisionVars
      using ‹InvariantVarsM (getM stateA) F0 decisionVars›
        ‹var ul ∈ vars F0 ∪ decisionVars›
        ‹getM stateB = getM stateA @ [(ul, False)]›
        InvariantVarsMAfterUnitPropagate[of getM stateA F0 decision-
Vars ul getM stateB]
      by simp
    moreover
    have InvariantConsistent (getM stateB)
      using ‹InvariantConsistent (getM stateA)›
        ‹isUnitClause uc ul (elements (getM stateA))›
        ‹getM stateB = getM stateA @ [(ul, False)]›
          InvariantConsistentAfterUnitPropagate [of getM stateA uc ul
getM stateB]
      by simp
    moreover
    have InvariantUniq (getM stateB)
      using ‹InvariantUniq (getM stateA)›
        ‹isUnitClause uc ul (elements (getM stateA))›
        ‹getM stateB = getM stateA @ [(ul, False)]›
          InvariantUniqAfterUnitPropagate [of getM stateA uc ul getM
stateB]
      by simp
    ultimately
    have ?thesis
      unfolding invariantsHoldInState-def
      by auto
  }
  moreover
  {
    assume appliedBacktrack stateA stateB F0
    hence formulaFalse F0 (elements (getM stateA))
      formulaFalse F0 (elements (getM stateA))
      decisions (getM stateA) ≠ []
     getM stateB = prefixBeforeLastDecision (getM stateA) @ [(opposite
(lastDecision (getM stateA)), False)]
      unfolding appliedBacktrack-def
      by auto

    have InvariantImpliedLiterals F0 (getM stateB)
      using ‹InvariantImpliedLiterals F0 (getM stateA)›
        ‹formulaFalse F0 (elements (getM stateA))›
        ‹decisions (getM stateA) ≠ []›
          ‹getM stateB = prefixBeforeLastDecision (getM stateA) @
[(opposite (lastDecision (getM stateA)), False)]›
```

‹*InvariantUniq* (*getM stateA*)›
‹*InvariantConsistent* (*getM stateA*)›
InvariantImpliedLiteralsAfterBacktrack[*of F0 getM stateA getM stateB*]
by *simp*
moreover
have *InvariantVarsM* (*getM stateB*) *F0 decisionVars*
using ‹*InvariantVarsM* (*getM stateA*) *F0 decisionVars*›
‹*decisions* (*getM stateA*) ≠ []›
‹*getM stateB* = *prefixBeforeLastDecision* (*getM stateA*) @ [(*opposite* (*lastDecision* (*getM stateA*)), *False*)]›
InvariantVarsMAfterBacktrack[*of getM stateA F0 decisionVars getM stateB*]
by *simp*
moreover
have *InvariantConsistent* (*getM stateB*)
using ‹*InvariantConsistent* (*getM stateA*)›
‹*InvariantUniq* (*getM stateA*)›
‹*decisions* (*getM stateA*) ≠ []›
‹*getM stateB* = *prefixBeforeLastDecision* (*getM stateA*) @ [(*opposite* (*lastDecision* (*getM stateA*)), *False*)]›
InvariantConsistentAfterBacktrack[*of getM stateA getM stateB*]
by *simp*
moreover
have *InvariantUniq* (*getM stateB*)
using ‹*InvariantConsistent* (*getM stateA*)›
‹*InvariantUniq* (*getM stateA*)›
‹*decisions* (*getM stateA*) ≠ []›
‹*getM stateB* = *prefixBeforeLastDecision* (*getM stateA*) @ [(*opposite* (*lastDecision* (*getM stateA*)), *False*)]›
InvariantUniqAfterBacktrack[*of getM stateA getM stateB*]
by *simp*
ultimately
have *?thesis*
unfolding *invariantsHoldInState-def*
by *auto*
}
ultimately
show *?thesis*
using ‹*transition stateA stateB F0 decisionVars*›
unfolding *transition-def*
by *auto*
qed

The consequence is that invariants hold in all valid runs.

lemma *invariantsHoldInValidRuns*:
fixes *F0* :: *Formula* **and** *decisionVars* :: *Variable set*
assumes *invariantsHoldInState stateA F0 decisionVars* **and**
(*stateA*, *stateB*) ∈ *transitionRelation F0 decisionVars*

shows *invariantsHoldInState stateB F0 decisionVars*
using *assms*
using *transitionsPreserveInvariants*
using *rtrancl-induct*[*of stateA stateB*
{(*stateA*, *stateB*). *transition stateA stateB F0 decisionVars*} λ *x*.
invariantsHoldInState x F0 decisionVars]
unfolding *transitionRelation-def*
by *auto*

lemma *invariantsHoldInValidRunsFromInitialState*:
fixes *F0* :: *Formula* **and** *decisionVars* :: *Variable set*
assumes *isInitialState state0 F0*
and (*state0*, *state*) ∈ *transitionRelation F0 decisionVars*
shows *invariantsHoldInState state F0 decisionVars*
proof−
from ‹*isInitialState state0 F0*›
have *invariantsHoldInState state0 F0 decisionVars*
by (*simp add*:*invariantsHoldInInitialState*)
with *assms*
show *?thesis*
using *invariantsHoldInValidRuns* [*of state0 F0 decisionVars state*]
by *simp*
qed

In the following text we will show that there are two kinds of states:

1. *UNSAT* states where *formulaFalse F0* (*elements* (*getM state*)) and *decisions* (*getM state*) = [].
2. *SAT* states where ¬ *formulaFalse F0* (*elements* (*getM state*)) and *decisionVars* ⊆ *vars* (*elements* (*getM state*)).

The soundness theorems claim that if *UNSAT* state is reached the formula is unsatisfiable and if *SAT* state is reached, the formula is satisfiable.

Completeness theorems claim that every final state is either *UNSAT* or *SAT*. A consequence of this and soundness theorems, is that if formula is unsatisfiable the solver will finish in an *UNSAT* state, and if the formula is satisfiable the solver will finish in a *SAT* state.

5.3 Soundness

theorem *soundnessForUNSAT*:
fixes *F0* :: *Formula* **and** *decisionVars* :: *Variable set* **and** *state0* :: *State* **and** *state* :: *State*
assumes
isInitialState state0 F0 **and**

(*state0*, *state*) ∈ *transitionRelation F0 decisionVars*

formulaFalse F0 (*elements* (*getM state*))
decisions (*getM state*) = []
shows ¬ *satisfiable F0*

proof−
from ‹*isInitialState state0 F0*› ‹(*state0*, *state*) ∈ *transitionRelation F0 decisionVars*›
have *invariantsHoldInState state F0 decisionVars*
using *invariantsHoldInValidRunsFromInitialState*
by *simp*
hence *InvariantImpliedLiterals F0* (*getM state*)
unfolding *invariantsHoldInState-def*
by *auto*
with ‹*formulaFalse F0* (*elements* (*getM state*))›
‹*decisions* (*getM state*) = []›
show *?thesis*
using *unsatReport*[*of F0 getM state F0*]
unfolding *InvariantEquivalent-def equivalentFormulae-def*
by *simp*
qed

theorem *soundnessForSAT*:
fixes *F0* :: *Formula* **and** *decisionVars* :: *Variable set* **and** *state0* :: *State* **and** *state* :: *State*
assumes
vars F0 ⊆ *decisionVars* **and**

isInitialState state0 F0 **and**
(*state0*, *state*) ∈ *transitionRelation F0 decisionVars*

¬ *formulaFalse F0* (*elements* (*getM state*))
vars (*elements* (*getM state*)) ⊇ *decisionVars*

shows
model (*elements* (*getM state*)) *F0*

proof−
from ‹*isInitialState state0 F0*› ‹(*state0*, *state*) ∈ *transitionRelation F0 decisionVars*›
have *invariantsHoldInState state F0 decisionVars*
using *invariantsHoldInValidRunsFromInitialState*
by *simp*
hence
InvariantConsistent (*getM state*)
unfolding *invariantsHoldInState-def*
by *auto*

with *assms*
show *?thesis*
using *satReport[of F0 decisionVars F0 getM state]*
unfolding *InvariantEquivalent-def equivalentFormulae-def*
InvariantVarsF-def
by *auto*
qed

5.4 Termination

We now define a termination ordering on the set of states based on the *lexLessRestricted* trail ordering. This ordering will be central in termination proof.

definition *terminationLess* (*F0*::*Formula*) *decisionVars* == {((*stateA*::*State*), (*stateB*::*State*)).
(*getM stateA*, *getM stateB*) ∈ *lexLessRestricted* (*vars F0* ∪ *decisionVars*)}

We want to show that every valid transition decreases a state with respect to the constructed termination ordering. Therefore, we show that *Decide*, *UnitPropagate* and *Backtrack* rule decrease the trail with respect to the restricted trail ordering. Invariants ensure that trails are indeed *uniq*, *consistent* and with finite variable sets.

lemma *trailIsDecreasedByDeciedUnitPropagateAndBacktrack*:
fixes *stateA*::*State* **and** *stateB*::*State*
assumes *invariantsHoldInState stateA F0 decisionVars* **and**
appliedDecide stateA stateB decisionVars ∨ *appliedUnitPropagate stateA stateB F0* ∨ *appliedBacktrack stateA stateB F0*
shows (*getM stateB*, *getM stateA*) ∈ *lexLessRestricted* (*vars F0* ∪ *decisionVars*)
proof−
from ‹*appliedDecide stateA stateB decisionVars* ∨ *appliedUnitPropagate stateA stateB F0* ∨ *appliedBacktrack stateA stateB F0*›
‹*invariantsHoldInState stateA F0 decisionVars*›
have *invariantsHoldInState stateB F0 decisionVars*
using *transitionsPreserveInvariants*
unfolding *transition-def*
by *auto*
from ‹*invariantsHoldInState stateA F0 decisionVars*›
have ∗: *uniq* (*elements* (*getM stateA*)) *consistent* (*elements* (*getM stateA*)) *vars* (*elements* (*getM stateA*)) ⊆ *vars F0* ∪ *decisionVars*
unfolding *invariantsHoldInState-def*
unfolding *InvariantVarsM-def*
unfolding *InvariantConsistent-def*
unfolding *InvariantUniq-def*
by *auto*
from ‹*invariantsHoldInState stateB F0 decisionVars*›

have ∗∗: *uniq* (*elements* (*getM stateB*)) *consistent* (*elements* (*getM stateB*)) *vars* (*elements* (*getM stateB*)) ⊆ *vars F0* ∪ *decisionVars*
unfolding *invariantsHoldInState-def*
unfolding *InvariantVarsM-def*
unfolding *InvariantConsistent-def*
unfolding *InvariantUniq-def*
by *auto*
{
assume *appliedDecide stateA stateB decisionVars*
hence (*getM stateB*, *getM stateA*) ∈ *lexLess*
unfolding *appliedDecide-def*
by (*auto simp add*:*lexLessAppend*)
with ∗ ∗∗
have ((*getM stateB*), (*getM stateA*)) ∈ *lexLessRestricted* (*vars F0* ∪ *decisionVars*)
unfolding *lexLessRestricted-def*
by *auto*
}
moreover
{
assume *appliedUnitPropagate stateA stateB F0*
hence (*getM stateB*, *getM stateA*) ∈ *lexLess*
unfolding *appliedUnitPropagate-def*
by (*auto simp add*:*lexLessAppend*)
with ∗ ∗∗
have (*getM stateB*, *getM stateA*) ∈ *lexLessRestricted* (*vars F0* ∪ *decisionVars*)
unfolding *lexLessRestricted-def*
by *auto*
}
moreover
{
assume *appliedBacktrack stateA stateB F0*
hence
formulaFalse F0 (*elements* (*getM stateA*))
decisions (*getM stateA*) ≠ []
getM stateB = *prefixBeforeLastDecision* (*getM stateA*) @ [(*opposite* (*lastDecision* (*getM stateA*)), *False*)]
unfolding *appliedBacktrack-def*
by *auto*
hence (*getM stateB*, *getM stateA*) ∈ *lexLess*
using ‹*decisions* (*getM stateA*) ≠ []›
‹*getM stateB* = *prefixBeforeLastDecision* (*getM stateA*) @ [(*opposite* (*lastDecision* (*getM stateA*)), *False*)]›
by (*simp add*:*lexLessBacktrack*)
with ∗ ∗∗
have (*getM stateB*, *getM stateA*) ∈ *lexLessRestricted* (*vars F0* ∪ *decisionVars*)
unfolding *lexLessRestricted-def*

```
    by auto
  }
  ultimately
  show ?thesis
    using assms
    by auto
qed
```

Now we can show that every rule application decreases a state with respect to the constructed termination ordering.

```
lemma stateIsDecreasedByValidTransitions:
  fixes stateA::State and stateB::State
  assumes invariantsHoldInState stateA F0 decisionVars and transition stateA stateB F0 decisionVars
  shows (stateB, stateA) ∈ terminationLess F0 decisionVars
proof−
  from ‹transition stateA stateB F0 decisionVars›
  have appliedDecide stateA stateB decisionVars ∨ appliedUnitPropagate stateA stateB F0 ∨ appliedBacktrack stateA stateB F0
    unfolding transition-def
    by simp
  with ‹invariantsHoldInState stateA F0 decisionVars›
  have (getM stateB, getM stateA) ∈ lexLessRestricted (vars F0 ∪ decisionVars)
    using trailIsDecreasedByDeciedUnitPropagateAndBacktrack
    by simp
  thus ?thesis
    unfolding terminationLess-def
    by simp
qed
```

The minimal states with respect to the termination ordering are final i.e., no further transition rules are applicable.

```
definition
isMinimalState stateMin F0 decisionVars == (∀ state::State. (state, stateMin) ∉ terminationLess F0 decisionVars)

lemma minimalStatesAreFinal:
  fixes stateA::State
  assumes invariantsHoldInState state F0 decisionVars and isMinimalState state F0 decisionVars
  shows isFinalState state F0 decisionVars
proof−
  {
    assume ¬ ?thesis
    then obtain state'::State
      where transition state state' F0 decisionVars
      unfolding isFinalState-def
      by auto
```

with ‹*invariantsHoldInState state F0 decisionVars*›
have (*state′*, *state*) ∈ *terminationLess F0 decisionVars*
using *stateIsDecreasedByValidTransitions*[*of state F0 decisionVars state′*]
unfolding *transition-def*
by *auto*
with ‹*isMinimalState state F0 decisionVars*›
have *False*
unfolding *isMinimalState-def*
by *auto*
}
thus *?thesis*
by *auto*
qed

The following key lemma shows that the termination ordering is well founded.

lemma *wfTerminationLess*:
fixes *decisionVars* :: *Variable set* **and** *F0* :: *Formula*
assumes *finite decisionVars*
shows *wf* (*terminationLess F0 decisionVars*)
unfolding *wf-eq-minimal*
proof−
show ∀ *Q state*. *state* ∈ *Q* ⟶ (∃ *stateMin*∈*Q*. ∀ *state′*. (*state′*, *stateMin*) ∈ *terminationLess F0 decisionVars* ⟶ *state′* ∉ *Q*)
proof−
{
fix *Q* :: *State set* **and** *state* :: *State*
assume *state* ∈ *Q*
let *?Q1* = {*M*::*LiteralTrail*. ∃ *state*. *state* ∈ *Q* ∧ (*getM state*) = *M*}
from ‹*state* ∈ *Q*›
have *getM state* ∈ *?Q1*
by *auto*
from ‹*finite decisionVars*›
have *finite* (*vars F0* ∪ *decisionVars*)
using *finiteVarsFormula*[*of F0*]
by *simp*
hence *wf* (*lexLessRestricted* (*vars F0* ∪ *decisionVars*))
using *wfLexLessRestricted*[*of vars F0* ∪ *decisionVars*]
by *simp*
with ‹*getM state* ∈ *?Q1*›
obtain *Mmin* **where** *Mmin* ∈ *?Q1* ∀ *M′*. (*M′*, *Mmin*) ∈ *lexLessRestricted* (*vars F0* ∪ *decisionVars*) ⟶ *M′* ∉ *?Q1*
unfolding *wf-eq-minimal*
apply (*erule-tac x*=*?Q1* **in** *allE*)
apply (*erule-tac x*=*getM state* **in** *allE*)
by *auto*
from ‹*Mmin* ∈ *?Q1*› **obtain** *stateMin*

where *stateMin* ∈ *Q* (*getM stateMin*) = *Mmin*
by *auto*
have ∀ *state'*. (*state'*, *stateMin*) ∈ *terminationLess F0 decisionVars* ⟶ *state'* ∉ *Q*
proof
fix *state'*
show (*state'*, *stateMin*) ∈ *terminationLess F0 decisionVars* ⟶ *state'* ∉ *Q*
proof
assume (*state'*, *stateMin*) ∈ *terminationLess F0 decisionVars*
hence (*getM state'*, *getM stateMin*) ∈ *lexLessRestricted* (*vars F0* ∪ *decisionVars*)
unfolding *terminationLess-def*
by *auto*
from ‹∀ *M'*. (*M'*, *Mmin*) ∈ *lexLessRestricted* (*vars F0* ∪ *decisionVars*) ⟶ *M'* ∉ *?Q1*›
‹(*getM state'*, *getM stateMin*) ∈ *lexLessRestricted* (*vars F0* ∪ *decisionVars*)› ‹*getM stateMin* = *Mmin*›
have *getM state'* ∉ *?Q1*
by *simp*
with ‹*getM stateMin* = *Mmin*›
show *state'* ∉ *Q*
by *auto*
qed
qed
with ‹*stateMin* ∈ *Q*›
have ∃ *stateMin* ∈ *Q*. (∀ *state'*. (*state'*, *stateMin*) ∈ *terminationLess F0 decisionVars* ⟶ *state'* ∉ *Q*)
by *auto*
}
thus *?thesis*
by *auto*
qed
qed

Using the termination ordering we show that the transition relation is well founded on states reachable from initial state.

theorem *wfTransitionRelation*:
fixes *decisionVars* :: *Variable set* **and** *F0* :: *Formula* **and** *state0* :: *State*
assumes *finite decisionVars* **and** *isInitialState state0 F0*
shows *wf* {(*stateB*, *stateA*).
(*state0*, *stateA*) ∈ *transitionRelation F0 decisionVars* ∧ (*transition stateA stateB F0 decisionVars*)}

proof−
let *?rel* = {(*stateB*, *stateA*).
(*state0*, *stateA*) ∈ *transitionRelation F0 decisionVars* ∧ (*transition stateA stateB F0 decisionVars*)}

let *?rel′= terminationLess F0 decisionVars*

have ∀ *x y*. (*x*, *y*) ∈ *?rel* ⟶ (*x*, *y*) ∈ *?rel′*
proof−
{
fix *stateA*::*State* **and** *stateB*::*State*
assume (*stateB*, *stateA*) ∈ *?rel*
hence (*stateB*, *stateA*) ∈ *?rel′*
using ‹*isInitialState state0 F0*›
using *invariantsHoldInValidRunsFromInitialState*[*of state0 F0 stateA decisionVars*]
using *stateIsDecreasedByValidTransitions*[*of stateA F0 decisionVars stateB*]
by *simp*
}
thus *?thesis*
by *simp*
qed
moreover
have *wf ?rel′*
using ‹*finite decisionVars*›
by (*rule wfTerminationLess*)
ultimately
show *?thesis*
using *wellFoundedEmbed*[*of ?rel ?rel′*]
by *simp*
qed

We will now give two corollaries of the previous theorem. First is a weak termination result that shows that there is a terminating run from every intial state to the final one.

corollary
fixes *decisionVars* :: *Variable set* **and** *F0* :: *Formula* **and** *state0* :: *State*
assumes *finite decisionVars* **and** *isInitialState state0 F0*
shows ∃ *state*. (*state0*, *state*) ∈ *transitionRelation F0 decisionVars* ∧ *isFinalState state F0 decisionVars*
proof−
{
assume ¬ *?thesis*
let *?Q* = {*state*. (*state0*, *state*) ∈ *transitionRelation F0 decisionVars*}
let *?rel* = {(*stateB*, *stateA*). (*state0*, *stateA*) ∈ *transitionRelation F0 decisionVars* ∧
transition stateA stateB F0 decisionVars}
have *state0* ∈ *?Q*
unfolding *transitionRelation-def*
by *simp*
hence ∃ *state*. *state* ∈ *?Q*

by *auto*

from *assms*
have *wf ?rel*
using *wfTransitionRelation*[*of decisionVars state0 F0*]
by *auto*
hence ∀ *Q*. (∃ *x*. *x* ∈ *Q*) ⟶ (∃ *stateMin* ∈ *Q*. ∀ *state*. (*state*, *stateMin*) ∈ *?rel* ⟶ *state* ∉ *Q*)
unfolding *wf-eq-minimal*
by *simp*
hence (∃ *x*. *x* ∈ *?Q*) ⟶ (∃ *stateMin* ∈ *?Q*. ∀ *state*. (*state*, *stateMin*) ∈ *?rel* ⟶ *state* ∉ *?Q*)
by *rule*
with ‹∃ *state*. *state* ∈ *?Q*›
have ∃ *stateMin* ∈ *?Q*. ∀ *state*. (*state*, *stateMin*) ∈ *?rel* ⟶ *state* ∉ *?Q*
by *simp*
then obtain *stateMin*
where *stateMin* ∈ *?Q* **and** ∀ *state*. (*state*, *stateMin*) ∈ *?rel* ⟶ *state* ∉ *?Q*
by *auto*

from ‹*stateMin* ∈ *?Q*›
have (*state0*, *stateMin*) ∈ *transitionRelation F0 decisionVars*
by *simp*
with ‹¬ *?thesis*›
have ¬ *isFinalState stateMin F0 decisionVars*
by *simp*
then obtain *state'*::*State*
where *transition stateMin state' F0 decisionVars*
unfolding *isFinalState-def*
by *auto*
have (*state'*, *stateMin*) ∈ *?rel*
using ‹(*state0*, *stateMin*) ∈ *transitionRelation F0 decisionVars*›
‹*transition stateMin state' F0 decisionVars*›
by *simp*
with ‹∀ *state*. (*state*, *stateMin*) ∈ *?rel* ⟶ *state* ∉ *?Q*›
have *state'* ∉ *?Q*
by *force*
moreover
from ‹(*state0*, *stateMin*) ∈ *transitionRelation F0 decisionVars*› ‹*transition stateMin state' F0 decisionVars*›
have *state'* ∈ *?Q*
unfolding *transitionRelation-def*
using *rtrancl-into-rtrancl*[*of state0 stateMin* {(*stateA*, *stateB*). *transition stateA stateB F0 decisionVars*} *state'*]
by *simp*
ultimately
have *False*

```
    by simp
  }
  thus ?thesis
    by auto
qed
```

Now we prove the final strong termination result which states that there cannot be infinite chains of transitions. If there is an infinite transition chain that starts from an initial state, its elements would for a set that would contain initial state and for every element of that set there would be another element of that set that is directly reachable from it. We show that no such set exists.

```
corollary noInfiniteTransitionChains:
  fixes F0::Formula and decisionVars::Variable set
  assumes finite decisionVars
  shows ¬ (∃ Q::(State set). ∃ state0 ∈ Q. isInitialState state0 F0 ∧

                              (∀ state ∈ Q. (∃ state' ∈ Q. transition state
state' F0 decisionVars))
           )
proof−
  {
  assume ¬ ?thesis
  then obtain Q::State set and state0::State
    where isInitialState state0 F0 state0 ∈ Q
          ∀ state ∈ Q. (∃ state' ∈ Q. transition state state' F0 deci-
sionVars)
    by auto
  let ?rel = {(stateB, stateA). (state0, stateA) ∈ transitionRelation
F0 decisionVars ∧
                         transition stateA stateB F0 decisionVars}
  from ‹finite decisionVars› ‹isInitialState state0 F0›
  have wf ?rel
    using wfTransitionRelation
    by simp
  hence wfmin: ∀ Q x. x ∈ Q ⟶
         (∃ z∈Q. ∀ y. (y, z) ∈ ?rel ⟶ y ∉ Q)
    unfolding wf-eq-minimal
    by simp
  let ?Q = {state ∈ Q. (state0, state) ∈ transitionRelation F0 deci-
sionVars}
  from ‹state0 ∈ Q›
  have state0 ∈ ?Q
    unfolding transitionRelation-def
    by simp
  with wfmin
  obtain stateMin::State
```

where *stateMin* ∈ *?Q* **and** ∀ *y*. (*y*, *stateMin*) ∈ *?rel* ⟶ *y* ∉ *?Q*
apply (*erule-tac x=?Q* **in** *allE*)
by *auto*

from ‹*stateMin* ∈ *?Q*›
have *stateMin* ∈ *Q* (*state0*, *stateMin*) ∈ *transitionRelation F0 decisionVars*
by *auto*
with ‹∀ *state* ∈ *Q*. (∃ *state'* ∈ *Q*. *transition state state' F0 decisionVars*)›
obtain *state'*::*State*
where *state'* ∈ *Q* *transition stateMin state' F0 decisionVars*
by *auto*

with ‹(*state0*, *stateMin*) ∈ *transitionRelation F0 decisionVars*›
have (*state'*, *stateMin*) ∈ *?rel*
by *simp*
with ‹∀ *y*. (*y*, *stateMin*) ∈ *?rel* ⟶ *y* ∉ *?Q*›
have *state'* ∉ *?Q*
by *force*

from ‹*state'* ∈ *Q*› ‹(*state0*, *stateMin*) ∈ *transitionRelation F0 decisionVars*›
‹*transition stateMin state' F0 decisionVars*›
have *state'* ∈ *?Q*
unfolding *transitionRelation-def*
using *rtrancl-into-rtrancl*[*of state0 stateMin* {(*stateA*, *stateB*). *transition stateA stateB F0 decisionVars*} *state'*]
by *simp*
with ‹*state'* ∉ *?Q*›
have *False*
by *simp*
}
thus *?thesis*
by *force*
qed

5.5 Completeness

In this section we will first show that each final state is either *SAT* or *UNSAT* state.

lemma *finalNonConflictState*:
fixes *state*::*State* **and** *FO* :: *Formula*
assumes
¬ *applicableDecide state decisionVars*
shows *vars* (*elements* (*getM state*)) ⊇ *decisionVars*
proof
fix *x* :: *Variable*
let *?l* = *Pos x*

assume *x* ∈ *decisionVars*
hence *var ?l* = *x* **and** *var ?l* ∈ *decisionVars* **and** *var* (*opposite ?l*) ∈ *decisionVars*
by *auto*
with ‹¬ *applicableDecide state decisionVars*›
have *literalTrue ?l* (*elements* (*getM state*)) ∨ *literalFalse ?l* (*elements* (*getM state*))
unfolding *applicableDecideCharacterization*
by *force*
with ‹*var ?l* = *x*›
show *x* ∈ *vars* (*elements* (*getM state*))
using *valuationContainsItsLiteralsVariable*[*of ?l elements* (*getM state*)]
using *valuationContainsItsLiteralsVariable*[*of opposite ?l elements* (*getM state*)]
by *auto*
qed

lemma *finalConflictingState*:
fixes *state* :: *State*
assumes
¬ *applicableBacktrack state F0* **and**
formulaFalse F0 (*elements* (*getM state*))
shows
decisions (*getM state*) = []
using *assms*
using *applicableBacktrackCharacterization*
by *auto*

lemma *finalStateCharacterizationLemma*:
fixes *state* :: *State*
assumes
¬ *applicableDecide state decisionVars* **and**
¬ *applicableBacktrack state F0*
shows
(¬ *formulaFalse F0* (*elements* (*getM state*)) ∧ *vars* (*elements* (*getM state*)) ⊇ *decisionVars*) ∨
(*formulaFalse F0* (*elements* (*getM state*)) ∧ *decisions* (*getM state*) = [])
proof (*cases formulaFalse F0* (*elements* (*getM state*)))
case *True*
hence *decisions* (*getM state*) = []
using *assms*
using *finalConflictingState*
by *auto*
with *True*
show *?thesis*
by *simp*

next
  case *False*
  hence *vars* (*elements* (*getM state*)) ⊇ *decisionVars*
    using *assms*
    using *finalNonConflictState*
    by *auto*
  with *False*
  show *?thesis*
    by *simp*
qed

theorem *finalStateCharacterization*:
  fixes *F0* :: *Formula* **and** *decisionVars* :: *Variable set* **and** *state0* :: *State* **and** *state* :: *State*
  assumes
  isInitialState state0 F0 **and**
  (*state0*, *state*) ∈ *transitionRelation F0 decisionVars* **and**
  isFinalState state F0 decisionVars
  shows
  (¬ *formulaFalse F0* (*elements* (*getM state*)) ∧ *vars* (*elements* (*getM state*)) ⊇ *decisionVars*) ∨
    (*formulaFalse F0* (*elements* (*getM state*)) ∧ *decisions* (*getM state*) = [])

proof−
  from ‹*isFinalState state F0 decisionVars*›
  have ∗∗:
    ¬ *applicableBacktrack state F0*
    ¬ *applicableDecide state decisionVars*
    unfolding *finalStateNonApplicable*
    by *auto*

  thus *?thesis*
    using *finalStateCharacterizationLemma*[*of state decisionVars*]
    by *simp*
qed

Completeness theorems are easy consequences of this characterization and soundness.

theorem *completenessForSAT*:
  fixes *F0* :: *Formula* **and** *decisionVars* :: *Variable set* **and** *state0* :: *State* **and** *state* :: *State*
  assumes
  satisfiable F0 **and**

  isInitialState state0 F0 **and**
  (*state0*, *state*) ∈ *transitionRelation F0 decisionVars* **and**

isFinalState state F0 decisionVars

shows ¬ *formulaFalse F0* (*elements* (*getM state*)) ∧ *vars* (*elements* (*getM state*)) ⊇ *decisionVars*

proof−
from *assms*
have ∗: (¬ *formulaFalse F0* (*elements* (*getM state*)) ∧ *vars* (*elements* (*getM state*)) ⊇ *decisionVars*) ∨
(*formulaFalse F0* (*elements* (*getM state*)) ∧ *decisions* (*getM state*) = [])
using *finalStateCharacterization*[*of state0 F0 state decisionVars*]
by *auto*
{
assume *formulaFalse F0* (*elements* (*getM state*))
with ∗
have *formulaFalse F0* (*elements* (*getM state*)) *decisions* (*getM state*) = []
by *auto*
with *assms*
have ¬ *satisfiable F0*
using *soundnessForUNSAT*
by *simp*
with ‹*satisfiable F0*›
have *False*
by *simp*
}
with ∗ **show** *?thesis*
by *auto*
qed

theorem *completenessForUNSAT*:
fixes *F0* :: *Formula* **and** *decisionVars* :: *Variable set* **and** *state0* :: *State* **and** *state* :: *State*
assumes
vars F0 ⊆ *decisionVars* **and**
¬ *satisfiable F0* **and**

isInitialState state0 F0 **and**
(*state0*, *state*) ∈ *transitionRelation F0 decisionVars* **and**
isFinalState state F0 decisionVars

shows
formulaFalse F0 (*elements* (*getM state*)) ∧ *decisions* (*getM state*) = []

proof−
from *assms*

have ∗:
(¬ *formulaFalse F0* (*elements* (*getM state*)) ∧ *vars* (*elements* (*getM state*)) ⊇ *decisionVars*) ∨
(*formulaFalse F0* (*elements* (*getM state*)) ∧ *decisions* (*getM state*) = [])
using *finalStateCharacterization*[*of state0 F0 state decisionVars*]
by *auto*
{
assume ¬ *formulaFalse F0* (*elements* (*getM state*))
with ∗
have ¬ *formulaFalse F0* (*elements* (*getM state*)) *vars* (*elements* (*getM state*)) ⊇ *decisionVars*
by *auto*
with *assms*
have *satisfiable F0*
using *soundnessForSAT*[*of F0 decisionVars state0 state*]
unfolding *satisfiable-def*
by *auto*
with ‹¬ *satisfiable F0*›
have *False*
by *simp*
}
with ∗ **show** *?thesis*
by *auto*
qed

theorem *partialCorrectness*:
fixes *F0* :: *Formula* **and** *decisionVars* :: *Variable set* **and** *state0* :: *State* **and** *state* :: *State*
assumes
vars F0 ⊆ *decisionVars* **and**

isInitialState state0 F0 **and**
(*state0*, *state*) ∈ *transitionRelation F0 decisionVars* **and**
isFinalState state F0 decisionVars

shows
satisfiable F0 = (¬ *formulaFalse F0* (*elements* (*getM state*)))

using *assms*
using *completenessForUNSAT*[*of F0 decisionVars state0 state*]
using *completenessForSAT*[*of F0 state0 state decisionVars*]
by *auto*

end

6 Transition system of Nieuwenhuis, Oliveras and Tinelli.

theory *NieuwenhuisOliverasTinelli*
imports *SatSolverVerification*
begin

This theory formalizes the transition rule system given by Nieuwenhuis et al. in [3]

6.1 Specification

record *State* =
getF :: *Formula*
getM :: *LiteralTrail*

definition
appliedDecide:: *State* ⇒ *State* ⇒ *Variable set* ⇒ *bool*
where
appliedDecide stateA stateB decisionVars ==
∃ *l*.
(*var l*) ∈ *decisionVars* ∧
¬ *l el* (*elements* (*getM stateA*)) ∧
¬ *opposite l el* (*elements* (*getM stateA*)) ∧

getF stateB = *getF stateA* ∧
getM stateB = *getM stateA* @ [(*l*, *True*)]

definition
applicableDecide :: *State* ⇒ *Variable set* ⇒ *bool*
where
applicableDecide state decisionVars == ∃ *state′*. *appliedDecide state state′ decisionVars*

definition
appliedUnitPropagate :: *State* ⇒ *State* ⇒ *bool*
where
appliedUnitPropagate stateA stateB ==
∃ (*uc*::*Clause*) (*ul*::*Literal*).
uc el (*getF stateA*) ∧
isUnitClause uc ul (*elements* (*getM stateA*)) ∧

getF stateB = *getF stateA* ∧
getM stateB = *getM stateA* @ [(*ul*, *False*)]

definition
applicableUnitPropagate :: *State* ⇒ *bool*
where

applicableUnitPropagate state == ∃ *state*′. *appliedUnitPropagate state state*′

definition
appliedBackjump :: *State* ⇒ *State* ⇒ *bool*
where
appliedBackjump stateA stateB ==
 ∃ *bc bl level*.
  isUnitClause bc bl (*elements* (*prefixToLevel level* (*getM stateA*))) ∧
  formulaEntailsClause (*getF stateA*) *bc* ∧
  var bl ∈ *vars* (*getF stateA*) ∪ *vars* (*elements* (*getM stateA*)) ∧
  0 ≤ *level* ∧ *level* < (*currentLevel* (*getM stateA*)) ∧

  getF stateB = *getF stateA* ∧
  getM stateB = *prefixToLevel level* (*getM stateA*) @ [(*bl*, *False*)]

definition
applicableBackjump :: *State* ⇒ *bool*
where
applicableBackjump state == ∃ *state*′. *appliedBackjump state state*′

definition
appliedLearn :: *State* ⇒ *State* ⇒ *bool*
where
appliedLearn stateA stateB ==
 ∃ *c*.
  (*formulaEntailsClause* (*getF stateA*) *c*) ∧
  (*vars c*) ⊆ *vars* (*getF stateA*) ∪ *vars* (*elements* (*getM stateA*)) ∧
  getF stateB = *getF stateA* @ [*c*] ∧
  getM stateB = *getM stateA*

definition
applicableLearn :: *State* ⇒ *bool*
where
applicableLearn state == (∃ *state*′. *appliedLearn state state*′)

Solving starts with the initial formula and the empty trail.

definition
isInitialState :: *State* ⇒ *Formula* ⇒ *bool*
where
isInitialState state F0 ==
  getF state = *F0* ∧
  getM state = []

Transitions are preformed only by using given rules.

definition
transition stateA stateB decisionVars ==
  appliedDecide stateA stateB decisionVars ∨
  appliedUnitPropagate stateA stateB ∨
  appliedLearn stateA stateB ∨
  appliedBackjump stateA stateB

Transition relation is obtained by applying transition rules iteratively. It is defined using a reflexive-transitive closure.

definition
transitionRelation decisionVars == ({(*stateA*, *stateB*). *transition stateA stateB decisionVars*})^*

Final state is one in which no rules apply

definition
isFinalState :: *State* ⇒ *Variable set* ⇒ *bool*
where
isFinalState state decisionVars == ¬ (∃ *state′*. *transition state state′ decisionVars*)

The following several lemmas establish conditions for applicability of different rules.

lemma *applicableDecideCharacterization*:
 fixes *stateA*::*State*
 shows *applicableDecide stateA decisionVars* =
 (∃ *l*.
   (*var l*) ∈ *decisionVars* ∧
   ¬ *l el* (*elements* (*getM stateA*)) ∧
   ¬ *opposite l el* (*elements* (*getM stateA*)))
  (**is** *?lhs* = *?rhs*)
proof
 assume *?rhs*
 then obtain *l* **where**
  ∗: (*var l*) ∈ *decisionVars* ¬ *l el* (*elements* (*getM stateA*)) ¬ *opposite l el* (*elements* (*getM stateA*))
  unfolding *applicableDecide-def*
  by *auto*
 let *?stateB* = *stateA*(| *getM* := (*getM stateA*) @ [(*l*, *True*)] |)
 from ∗ **have** *appliedDecide stateA ?stateB decisionVars*
  unfolding *appliedDecide-def*
  by *auto*
 thus *?lhs*
  unfolding *applicableDecide-def*
  by *auto*
next
 assume *?lhs*
 then obtain *stateB l*
  where (*var l*) ∈ *decisionVars* ¬ *l el* (*elements* (*getM stateA*))

¬ *opposite l el* (*elements* (*getM stateA*))
unfolding *applicableDecide-def*
unfolding *appliedDecide-def*
by *auto*
thus *?rhs*
by *auto*
qed

lemma *applicableUnitPropagateCharacterization*:
fixes *stateA*::*State* **and** *F0*::*Formula*
shows *applicableUnitPropagate stateA* =
(∃ (*uc*::*Clause*) (*ul*::*Literal*).
uc el (*getF stateA*) ∧
isUnitClause uc ul (*elements* (*getM stateA*)))
(**is** *?lhs* = *?rhs*)
proof
assume *?rhs*
then obtain *ul uc*
where ∗: *uc el* (*getF stateA*) *isUnitClause uc ul* (*elements* (*getM stateA*))
unfolding *applicableUnitPropagate-def*
by *auto*
let *?stateB* = *stateA*⦇ *getM* := *getM stateA* @ [(*ul*, *False*)] ⦈
from ∗ **have** *appliedUnitPropagate stateA ?stateB*
unfolding *appliedUnitPropagate-def*
by *auto*
thus *?lhs*
unfolding *applicableUnitPropagate-def*
by *auto*
next
assume *?lhs*
then obtain *stateB uc ul*
where *uc el* (*getF stateA*) *isUnitClause uc ul* (*elements* (*getM stateA*))
unfolding *applicableUnitPropagate-def*
unfolding *appliedUnitPropagate-def*
by *auto*
thus *?rhs*
by *auto*
qed

lemma *applicableBackjumpCharacterization*:
fixes *stateA*::*State*
shows *applicableBackjump stateA* =
(∃ *bc bl level*.
isUnitClause bc bl (*elements* (*prefixToLevel level* (*getM stateA*))) ∧
formulaEntailsClause (*getF stateA*) *bc* ∧
var bl ∈ *vars* (*getF stateA*) ∪ *vars* (*elements* (*getM stateA*)) ∧

0 ≤ *level* ∧ *level* < (*currentLevel* (*getM stateA*))) (**is** *?lhs* = *?rhs*)
proof
  assume *?rhs*
  then obtain *bc bl level*
    where ∗: *isUnitClause bc bl* (*elements* (*prefixToLevel level* (*getM stateA*)))
      formulaEntailsClause (*getF stateA*) *bc*
      var bl ∈ *vars* (*getF stateA*) ∪ *vars* (*elements* (*getM stateA*))
      0 ≤ *level level* < (*currentLevel* (*getM stateA*))
    unfolding *applicableBackjump-def*
    by *auto*
  let *?stateB* = *stateA*⦇ *getM* := *prefixToLevel level* (*getM stateA*) @ [(*bl*, *False*)]⦈
  from ∗ **have** *appliedBackjump stateA ?stateB*
    unfolding *appliedBackjump-def*
    by *auto*
  thus *?lhs*
    unfolding *applicableBackjump-def*
    by *auto*
next
  assume *?lhs*
  then obtain *stateB*
    where *appliedBackjump stateA stateB*
    unfolding *applicableBackjump-def*
    by *auto*
  then obtain *bc bl level*
    where *isUnitClause bc bl* (*elements* (*prefixToLevel level* (*getM stateA*)))
    formulaEntailsClause (*getF stateA*) *bc*
    var bl ∈ *vars* (*getF stateA*) ∪ *vars* (*elements* (*getM stateA*))
    getF stateB = *getF stateA*
    getM stateB = *prefixToLevel level* (*getM stateA*) @ [(*bl*, *False*)]
    0 ≤ *level level* < (*currentLevel* (*getM stateA*))
    unfolding *appliedBackjump-def*
    by *auto*
  thus *?rhs*
    by *auto*
qed

lemma *applicableLearnCharacterization*:
  fixes *stateA*::*State*
  shows *applicableLearn stateA* =
    (∃ *c*. *formulaEntailsClause* (*getF stateA*) *c* ∧
        vars c ⊆ *vars* (*getF stateA*) ∪ *vars* (*elements* (*getM stateA*)))
(**is** *?lhs* = *?rhs*)
proof
  assume *?rhs*
  then obtain *c* **where**
  ∗: *formulaEntailsClause* (*getF stateA*) *c*

vars c ⊆ *vars* (*getF stateA*) ∪ *vars* (*elements* (*getM stateA*))
unfolding *applicableLearn-def*
by *auto*
let *?stateB* = *stateA*⦇ *getF* := *getF stateA* @ [*c*]⦈
from ∗ **have** *appliedLearn stateA ?stateB*
unfolding *appliedLearn-def*
by *auto*
thus *?lhs*
unfolding *applicableLearn-def*
by *auto*
next
assume *?lhs*
then obtain *c stateB*
where
formulaEntailsClause (*getF stateA*) *c*
vars c ⊆ *vars* (*getF stateA*) ∪ *vars* (*elements* (*getM stateA*))
unfolding *applicableLearn-def*
unfolding *appliedLearn-def*
by *auto*
thus *?rhs*
by *auto*
qed

Final states are the ones where no rule is applicable.

lemma *finalStateNonApplicable*:
fixes *state*::*State*
shows *isFinalState state decisionVars* =
(¬ *applicableDecide state decisionVars* ∧
¬ *applicableUnitPropagate state* ∧
¬ *applicableBackjump state* ∧
¬ *applicableLearn state*)
unfolding *isFinalState-def*
unfolding *transition-def*
unfolding *applicableDecide-def*
unfolding *applicableUnitPropagate-def*
unfolding *applicableBackjump-def*
unfolding *applicableLearn-def*
by *auto*

6.2 Invariants

Invariants that are relevant for the rest of correctness proof.

definition
invariantsHoldInState :: *State* ⇒ *Formula* ⇒ *Variable set* ⇒ *bool*
where
invariantsHoldInState state F0 decisionVars ==
InvariantImpliedLiterals (*getF state*) (*getM state*) ∧
InvariantVarsM (*getM state*) *F0 decisionVars* ∧
InvariantVarsF (*getF state*) *F0 decisionVars* ∧

InvariantConsistent (*getM state*) ∧
InvariantUniq (*getM state*) ∧
InvariantEquivalent F0 (*getF state*)

Invariants hold in initial states.

lemma *invariantsHoldInInitialState*:
fixes *state* :: *State* **and** *F0* :: *Formula*
assumes *isInitialState state F0*
shows *invariantsHoldInState state F0 decisionVars*
using *assms*
by (*auto simp add*:
isInitialState-def
invariantsHoldInState-def
InvariantImpliedLiterals-def
InvariantVarsM-def
InvariantVarsF-def
InvariantConsistent-def
InvariantUniq-def
InvariantEquivalent-def equivalentFormulae-def
)

Valid transitions preserve invariants.

lemma *transitionsPreserveInvariants*:
fixes *stateA*::*State* **and** *stateB*::*State*
assumes *transition stateA stateB decisionVars* **and**
invariantsHoldInState stateA F0 decisionVars
shows *invariantsHoldInState stateB F0 decisionVars*
proof−
from ‹*invariantsHoldInState stateA F0 decisionVars*›
have
InvariantImpliedLiterals (*getF stateA*) (*getM stateA*) **and**
InvariantVarsM (*getM stateA*) *F0 decisionVars* **and**
InvariantVarsF (*getF stateA*) *F0 decisionVars* **and**
InvariantConsistent (*getM stateA*) **and**
InvariantUniq (*getM stateA*) **and**
InvariantEquivalent F0 (*getF stateA*)
unfolding *invariantsHoldInState-def*
by *auto*
{
assume *appliedDecide stateA stateB decisionVars*
then obtain *l*::*Literal* **where**
(*var l*) ∈ *decisionVars*
¬ *literalTrue l* (*elements* (*getM stateA*))
¬ *literalFalse l* (*elements* (*getM stateA*))
getM stateB = *getM stateA* @ [(*l*, *True*)]
getF stateB = *getF stateA*
unfolding *appliedDecide-def*
by *auto*

from ‹¬ *literalTrue l* (*elements* (*getM stateA*))› ‹¬ *literalFalse l* (*elements* (*getM stateA*))›
have ∗: *var l* ∉ *vars* (*elements* (*getM stateA*))
using *variableDefinedImpliesLiteralDefined*[*of l elements* (*getM stateA*)]
by *simp*

have *InvariantImpliedLiterals* (*getF stateB*) (*getM stateB*)
using ‹*getF stateB* = *getF stateA*›
‹*getM stateB* = *getM stateA* @ [(*l*, *True*)]›
‹*InvariantImpliedLiterals* (*getF stateA*) (*getM stateA*)›
‹*InvariantUniq* (*getM stateA*)›
‹*var l* ∉ *vars* (*elements* (*getM stateA*))›
InvariantImpliedLiteralsAfterDecide[*of getF stateA getM stateA l getM stateB*]
by *simp*
moreover
have *InvariantVarsM* (*getM stateB*) *F0 decisionVars*
using ‹*getM stateB* = *getM stateA* @ [(*l*, *True*)]›
‹*InvariantVarsM* (*getM stateA*) *F0 decisionVars*›
‹*var l* ∈ *decisionVars*›
InvariantVarsMAfterDecide[*of getM stateA F0 decisionVars l getM stateB*]
by *simp*
moreover
have *InvariantVarsF* (*getF stateB*) *F0 decisionVars*
using ‹*getF stateB* = *getF stateA*›
‹*InvariantVarsF* (*getF stateA*) *F0 decisionVars*›
by *simp*
moreover
have *InvariantConsistent* (*getM stateB*)
using ‹*getM stateB* = *getM stateA* @ [(*l*, *True*)]›
‹*InvariantConsistent* (*getM stateA*)›
‹*var l* ∉ *vars* (*elements* (*getM stateA*))›
InvariantConsistentAfterDecide[*of getM stateA l getM stateB*]
by *simp*
moreover
have *InvariantUniq* (*getM stateB*)
using ‹*getM stateB* = *getM stateA* @ [(*l*, *True*)]›
‹*InvariantUniq* (*getM stateA*)›
‹*var l* ∉ *vars* (*elements* (*getM stateA*))›
InvariantUniqAfterDecide[*of getM stateA l getM stateB*]
by *simp*
moreover
have *InvariantEquivalent F0* (*getF stateB*)
using ‹*getF stateB* = *getF stateA*›
‹*InvariantEquivalent F0* (*getF stateA*)›
by *simp*

```
    ultimately
    have ?thesis
      unfolding invariantsHoldInState-def
      by auto
  }
  moreover
  {
    assume appliedUnitPropagate stateA stateB
    then obtain uc::Clause and ul::Literal where
      uc el (getF stateA)
      isUnitClause uc ul (elements (getM stateA))
      getF stateB = getF stateA
      getM stateB = getM stateA @ [(ul, False)]
      unfolding appliedUnitPropagate-def
      by auto

    from ‹isUnitClause uc ul (elements (getM stateA))›
    have ul el uc
      unfolding isUnitClause-def
      by simp

    from ‹uc el (getF stateA)›
    have formulaEntailsClause (getF stateA) uc
      by (simp add: formulaEntailsItsClauses)


    have InvariantImpliedLiterals (getF stateB) (getM stateB)
      using ‹getF stateB = getF stateA›
        ‹InvariantImpliedLiterals (getF stateA) (getM stateA)›
        ‹formulaEntailsClause (getF stateA) uc›
        ‹isUnitClause uc ul (elements (getM stateA))›
        ‹getM stateB = getM stateA @ [(ul, False)]›
       InvariantImpliedLiteralsAfterUnitPropagate[of getF stateA getM
stateA uc ul getM stateB]
      by simp
    moreover
    from ‹ul el uc› ‹uc el (getF stateA)›
    have ul el (getF stateA)
      by (auto simp add: literalElFormulaCharacterization)
    with ‹InvariantVarsF (getF stateA) F0 decisionVars›
    have var ul ∈ vars F0 ∪ decisionVars
      using formulaContainsItsLiteralsVariable [of ul getF stateA]
      unfolding InvariantVarsF-def
      by auto

    have InvariantVarsM (getM stateB) F0 decisionVars
      using ‹InvariantVarsM (getM stateA) F0 decisionVars›
        ‹var ul ∈ vars F0 ∪ decisionVars›
        ‹getM stateB = getM stateA @ [(ul, False)]›
```

```
      InvariantVarsMAfterUnitPropagate[of getM stateA F0 decision-
Vars ul getM stateB]
      by simp
    moreover
    have InvariantVarsF (getF stateB) F0 decisionVars
      using ‹getF stateB = getF stateA›
      ‹InvariantVarsF (getF stateA) F0 decisionVars›
      by simp
    moreover
    have InvariantConsistent (getM stateB)
      using ‹InvariantConsistent (getM stateA)›
        ‹isUnitClause uc ul (elements (getM stateA))›
        ‹getM stateB = getM stateA @ [(ul, False)]›
        InvariantConsistentAfterUnitPropagate [of getM stateA uc ul
getM stateB]
      by simp
    moreover
    have InvariantUniq (getM stateB)
      using ‹InvariantUniq (getM stateA)›
        ‹isUnitClause uc ul (elements (getM stateA))›
        ‹getM stateB = getM stateA @ [(ul, False)]›
        InvariantUniqAfterUnitPropagate [of getM stateA uc ul getM
stateB]
      by simp
    moreover
    have InvariantEquivalent F0 (getF stateB)
      using ‹getF stateB = getF stateA›
      ‹InvariantEquivalent F0 (getF stateA)›
      by simp
    ultimately
    have ?thesis
      unfolding invariantsHoldInState-def
      by auto
  }
  moreover
  {
    assume appliedLearn stateA stateB
    then obtain c::Clause where
      formulaEntailsClause (getF stateA) c
      vars c ⊆ vars (getF stateA) ∪ vars (elements (getM stateA))
      getF stateB = getF stateA @ [c]
      getM stateB = getM stateA
      unfolding appliedLearn-def
      by auto

    have InvariantImpliedLiterals (getF stateB) (getM stateB)
      using
        ‹InvariantImpliedLiterals (getF stateA) (getM stateA)›
        ‹getF stateB = getF stateA @ [c]›
```

```
        ‹getM stateB = getM stateA›
        InvariantImpliedLiteralsAfterLearn[of getF stateA getM stateA
getF stateB]
      by simp
    moreover
    have InvariantVarsM (getM stateB) F0 decisionVars
      using
        ‹InvariantVarsM (getM stateA) F0 decisionVars›
        ‹getM stateB = getM stateA›
      by simp
    moreover
    from ‹vars c ⊆ vars (getF stateA) ∪ vars (elements (getM stateA))›
      ‹InvariantVarsM (getM stateA) F0 decisionVars›
        ‹InvariantVarsF (getF stateA) F0 decisionVars›
    have vars c ⊆ vars F0 ∪ decisionVars
      unfolding InvariantVarsM-def
      unfolding InvariantVarsF-def
      by auto
    hence InvariantVarsF (getF stateB) F0 decisionVars
      using ‹InvariantVarsF (getF stateA) F0 decisionVars›
        ‹getF stateB = getF stateA @ [c]›
      using varsAppendFormulae [of getF stateA [c]]
      unfolding InvariantVarsF-def
      by simp
    moreover
    have InvariantConsistent (getM stateB)
      using ‹InvariantConsistent (getM stateA)›
        ‹getM stateB = getM stateA›
      by simp
    moreover
    have InvariantUniq (getM stateB)
      using ‹InvariantUniq (getM stateA)›
        ‹getM stateB = getM stateA›
      by simp
    moreover
    have InvariantEquivalent F0 (getF stateB)
      using
        ‹InvariantEquivalent F0 (getF stateA)›
        ‹formulaEntailsClause (getF stateA) c›
        ‹getF stateB = getF stateA @ [c]›
        InvariantEquivalentAfterLearn[of F0 getF stateA c getF stateB]
      by simp
    ultimately
    have ?thesis
      unfolding invariantsHoldInState-def
      by simp
  }
  moreover
  {
```

assume *appliedBackjump stateA stateB*
then obtain *bc*::*Clause* **and** *bl*::*Literal* **and** *level*::*nat*
where
isUnitClause bc bl (*elements* (*prefixToLevel level* (*getM stateA*)))
formulaEntailsClause (*getF stateA*) *bc*
var bl ∈ *vars* (*getF stateA*) ∪ *vars* (*elements* (*getM stateA*))
getF stateB = *getF stateA*
getM stateB = *prefixToLevel level* (*getM stateA*) @ [(*bl*, *False*)]
unfolding *appliedBackjump-def*
by *auto*

have *isPrefix* (*prefixToLevel level* (*getM stateA*)) (*getM stateA*)
by (*simp add*:*isPrefixPrefixToLevel*)

have *InvariantImpliedLiterals* (*getF stateB*) (*getM stateB*)
using ‹*InvariantImpliedLiterals* (*getF stateA*) (*getM stateA*)›
‹*isPrefix* (*prefixToLevel level* (*getM stateA*)) (*getM stateA*)›
‹*isUnitClause bc bl* (*elements* (*prefixToLevel level* (*getM stateA*)))›
‹*formulaEntailsClause* (*getF stateA*) *bc*›
‹*getF stateB* = *getF stateA*›
‹*getM stateB* = *prefixToLevel level* (*getM stateA*) @ [(*bl*, *False*)]›
InvariantImpliedLiteralsAfterBackjump[*of getF stateA getM stateA prefixToLevel level* (*getM stateA*) *bc bl getM stateB*]
by *simp*
moreover

from ‹*InvariantVarsF* (*getF stateA*) *F0 decisionVars*›
‹*InvariantVarsM* (*getM stateA*) *F0 decisionVars*›
‹*var bl* ∈ *vars* (*getF stateA*) ∪ *vars* (*elements* (*getM stateA*))›
have *var bl* ∈ *vars F0* ∪ *decisionVars*
unfolding *InvariantVarsM-def*
unfolding *InvariantVarsF-def*
by *auto*

have *InvariantVarsM* (*getM stateB*) *F0 decisionVars*
using ‹*InvariantVarsM* (*getM stateA*) *F0 decisionVars*›
‹*isPrefix* (*prefixToLevel level* (*getM stateA*)) (*getM stateA*)›
‹*getM stateB* = *prefixToLevel level* (*getM stateA*) @ [(*bl*, *False*)]›
‹*var bl* ∈ *vars F0* ∪ *decisionVars*›
InvariantVarsMAfterBackjump[*of getM stateA F0 decisionVars prefixToLevel level* (*getM stateA*) *bl getM stateB*]
by *simp*
moreover
have *InvariantVarsF* (*getF stateB*) *F0 decisionVars*
using ‹*getF stateB* = *getF stateA*›
‹*InvariantVarsF* (*getF stateA*) *F0 decisionVars*›
by *simp*
moreover
have *InvariantConsistent* (*getM stateB*)

using ‹*InvariantConsistent* (*getM stateA*)›
‹*isPrefix* (*prefixToLevel level* (*getM stateA*)) (*getM stateA*)›
‹*isUnitClause bc bl* (*elements* (*prefixToLevel level* (*getM stateA*)))›
‹*getM stateB* = *prefixToLevel level* (*getM stateA*) @ [(*bl*, *False*)]›
InvariantConsistentAfterBackjump[*of getM stateA prefixToLevel level* (*getM stateA*) *bc bl getM stateB*]
by *simp*
moreover
have *InvariantUniq* (*getM stateB*)
using ‹*InvariantUniq* (*getM stateA*)›
‹*isPrefix* (*prefixToLevel level* (*getM stateA*)) (*getM stateA*)›
‹*isUnitClause bc bl* (*elements* (*prefixToLevel level* (*getM stateA*)))›
‹*getM stateB* = *prefixToLevel level* (*getM stateA*) @ [(*bl*, *False*)]›
InvariantUniqAfterBackjump[*of getM stateA prefixToLevel level* (*getM stateA*) *bc bl getM stateB*]
by *simp*
moreover
have *InvariantEquivalent F0* (*getF stateB*)
using
‹*InvariantEquivalent F0* (*getF stateA*)›
‹*getF stateB* = *getF stateA*›
by *simp*
ultimately
have *?thesis*
unfolding *invariantsHoldInState-def*
by *auto*
}
ultimately
show *?thesis*
using ‹*transition stateA stateB decisionVars*›
unfolding *transition-def*
by *auto*
qed

The consequence is that invariants hold in all valid runs.

lemma *invariantsHoldInValidRuns*:
fixes *F0* :: *Formula* **and** *decisionVars* :: *Variable set*
assumes *invariantsHoldInState stateA F0 decisionVars* **and**
(*stateA*, *stateB*) ∈ *transitionRelation decisionVars*
shows *invariantsHoldInState stateB F0 decisionVars*
using *assms*
using *transitionsPreserveInvariants*
using *rtrancl-induct*[*of stateA stateB*
{(*stateA*, *stateB*). *transition stateA stateB decisionVars*} λ *x*. *invariantsHoldInState x F0 decisionVars*]
unfolding *transitionRelation-def*
by *auto*

lemma *invariantsHoldInValidRunsFromInitialState*:

fixes *F0* :: *Formula* **and** *decisionVars* :: *Variable set*
assumes *isInitialState state0 F0*
and (*state0*, *state*) ∈ *transitionRelation decisionVars*
shows *invariantsHoldInState state F0 decisionVars*
proof –
from ‹*isInitialState state0 F0*›
have *invariantsHoldInState state0 F0 decisionVars*
by (*simp add*:*invariantsHoldInInitialState*)
with *assms*
show *?thesis*
using *invariantsHoldInValidRuns* [*of state0 F0 decisionVars state*]
by *simp*
qed

In the following text we will show that there are two kinds of states:

1. *UNSAT* states where *formulaFalse F0* (*elements* (*getM state*)) and *decisions* (*getM state*) = [].
2. *SAT* states where ¬ *formulaFalse F0* (*elements* (*getM state*)) and *decisionVars* ⊆ *vars* (*elements* (*getM state*))

The soundness theorems claim that if *UNSAT* state is reached the formula is unsatisfiable and if *SAT* state is reached, the formula is satisfiable.

Completeness theorems claim that every final state is either *UNSAT* or *SAT*. A consequence of this and soundness theorems, is that if formula is unsatisfiable the solver will finish in an *UNSAT* state, and if the formula is satisfiable the solver will finish in a *SAT* state.

6.3 Soundness

theorem *soundnessForUNSAT*:
fixes *F0* :: *Formula* **and** *decisionVars* :: *Variable set* **and** *state0* :: *State* **and** *state* :: *State*
assumes
isInitialState state0 F0 **and**
(*state0*, *state*) ∈ *transitionRelation decisionVars*

formulaFalse (*getF state*) (*elements* (*getM state*))
decisions (*getM state*) = []

shows ¬ *satisfiable F0*

proof –
from ‹*isInitialState state0 F0*› ‹(*state0*, *state*) ∈ *transitionRelation decisionVars*›

have *invariantsHoldInState state F0 decisionVars*
using *invariantsHoldInValidRunsFromInitialState*
by *simp*
hence *InvariantImpliedLiterals* (*getF state*) (*getM state*) *InvariantEquivalent F0* (*getF state*)
unfolding *invariantsHoldInState-def*
by *auto*
with ‹*formulaFalse* (*getF state*) (*elements* (*getM state*))›
‹*decisions* (*getM state*) = []›
show *?thesis*
using *unsatReport*[*of getF state getM state F0*]
by *simp*
qed

theorem *soundnessForSAT*:
fixes *F0* :: *Formula* **and** *decisionVars* :: *Variable set* **and** *state0* :: *State* **and** *state* :: *State*
assumes
vars F0 ⊆ *decisionVars* **and**

isInitialState state0 F0 **and**
(*state0*, *state*) ∈ *transitionRelation decisionVars*

¬ *formulaFalse* (*getF state*) (*elements* (*getM state*))
vars (*elements* (*getM state*)) ⊇ *decisionVars*
shows
model (*elements* (*getM state*)) *F0*

proof−
from ‹*isInitialState state0 F0*› ‹(*state0*, *state*) ∈ *transitionRelation decisionVars*›
have *invariantsHoldInState state F0 decisionVars*
using *invariantsHoldInValidRunsFromInitialState*
by *simp*
hence
InvariantConsistent (*getM state*)
InvariantEquivalent F0 (*getF state*)
InvariantVarsF (*getF state*) *F0 decisionVars*
unfolding *invariantsHoldInState-def*
by *auto*
with *assms*
show *?thesis*
using *satReport*[*of F0 decisionVars getF state getM state*]
by *simp*
qed

6.4 Termination

This system is terminating, but only under assumption that there is no infinite derivation consisting only of applications of rule *Learn*. We will formalize this condition by requiring that there there exists an ordering *learnL* on the formulae that is well-founded such that the state is decreased with each application of the *Learn* rule. If such ordering exists, the termination ordering is built as a lexicographic combination of *lexLessRestricted* trail ordering and the *learnL* ordering.

definition *lexLessState F0 decisionVars* == {((*stateA*::*State*), (*stateB*::*State*)).

(*getM stateA*, *getM stateB*) ∈ *lexLessRestricted* (*vars F0* ∪ *decisionVars*)}
definition *learnLessState learnL* == {((*stateA*::*State*), (*stateB*::*State*)).

getM stateA = *getM stateB* ∧ (*getF stateA*, *getF stateB*) ∈ *learnL*}
definition *terminationLess F0 decisionVars learnL* ==
{((*stateA*::*State*), (*stateB*::*State*)).
(*stateA*,*stateB*) ∈ *lexLessState F0 decisionVars* ∨
(*stateA*,*stateB*) ∈ *learnLessState learnL*}

We want to show that every valid transition decreases a state with respect to the constructed termination ordering. Therefore, we show that *Decide*, *UnitPropagate* and *Backjump* rule decrease the trail with respect to the restricted trail ordering *lexLessRestricted*. Invariants ensure that trails are indeed uniq, consistent and with finite variable sets. By assumption, *Learn* rule will decrease the formula component of the state with respect to the *learnL* ordering.

lemma *trailIsDecreasedByDeciedUnitPropagateAndBackjump*:
fixes *stateA*::*State* **and** *stateB*::*State*
assumes *invariantsHoldInState stateA F0 decisionVars* **and**
appliedDecide stateA stateB decisionVars ∨ *appliedUnitPropagate stateA stateB* ∨ *appliedBackjump stateA stateB*
shows (*getM stateB*, *getM stateA*) ∈ *lexLessRestricted* (*vars F0* ∪ *decisionVars*)
proof−
from ‹*appliedDecide stateA stateB decisionVars* ∨ *appliedUnitPropagate stateA stateB* ∨ *appliedBackjump stateA stateB*›
‹*invariantsHoldInState stateA F0 decisionVars*›
have *invariantsHoldInState stateB F0 decisionVars*
using *transitionsPreserveInvariants*
unfolding *transition-def*
by *auto*
from ‹*invariantsHoldInState stateA F0 decisionVars*›

have ∗: *uniq* (*elements* (*getM stateA*)) *consistent* (*elements* (*getM stateA*)) *vars* (*elements* (*getM stateA*)) ⊆ *vars F0* ∪ *decisionVars*
unfolding *invariantsHoldInState-def*
unfolding *InvariantVarsM-def*
unfolding *InvariantConsistent-def*
unfolding *InvariantUniq-def*
by *auto*
from ‹*invariantsHoldInState stateB F0 decisionVars*›
have ∗∗: *uniq* (*elements* (*getM stateB*)) *consistent* (*elements* (*getM stateB*)) *vars* (*elements* (*getM stateB*)) ⊆ *vars F0* ∪ *decisionVars*
unfolding *invariantsHoldInState-def*
unfolding *InvariantVarsM-def*
unfolding *InvariantConsistent-def*
unfolding *InvariantUniq-def*
by *auto*
{
assume *appliedDecide stateA stateB decisionVars*
hence (*getM stateB*, *getM stateA*) ∈ *lexLess*
unfolding *appliedDecide-def*
by (*auto simp add*:*lexLessAppend*)
with ∗ ∗∗
have ((*getM stateB*), (*getM stateA*)) ∈ *lexLessRestricted* (*vars F0* ∪ *decisionVars*)
unfolding *lexLessRestricted-def*
by *auto*
}
moreover
{
assume *appliedUnitPropagate stateA stateB*
hence (*getM stateB*, *getM stateA*) ∈ *lexLess*
unfolding *appliedUnitPropagate-def*
by (*auto simp add*:*lexLessAppend*)
with ∗ ∗∗
have (*getM stateB*, *getM stateA*) ∈ *lexLessRestricted* (*vars F0* ∪ *decisionVars*)
unfolding *lexLessRestricted-def*
by *auto*
}
moreover
{
assume *appliedBackjump stateA stateB*
then obtain *bc*::*Clause* **and** *bl*::*Literal* **and** *level*::*nat*
where
isUnitClause bc bl (*elements* (*prefixToLevel level* (*getM stateA*)))
formulaEntailsClause (*getF stateA*) *bc*
var bl ∈ *vars* (*getF stateA*) ∪ *vars* (*elements* (*getM stateA*))
0 ≤ *level level* < *currentLevel* (*getM stateA*)
getF stateB = *getF stateA*
getM stateB = *prefixToLevel level* (*getM stateA*) @ [(*bl*, *False*)]

unfolding *appliedBackjump-def*
by *auto*

with ‹*getM stateB* = *prefixToLevel level* (*getM stateA*) @ [(*bl*, *False*)]›
have (*getM stateB*, *getM stateA*) ∈ *lexLess*
by (*simp add*:*lexLessBackjump*)
with ∗ ∗∗
have (*getM stateB*, *getM stateA*) ∈ *lexLessRestricted* (*vars F0* ∪ *decisionVars*)
unfolding *lexLessRestricted-def*
by *auto*
}
ultimately
show *?thesis*
using *assms*
by *auto*
qed

Now we can show that, under the assumption for *Learn* rule, every rule application decreases a state with respect to the constructed termination ordering.

theorem *stateIsDecreasedByValidTransitions*:
fixes *stateA*::*State* **and** *stateB*::*State*
assumes *invariantsHoldInState stateA F0 decisionVars* **and** *transition stateA stateB decisionVars*
appliedLearn stateA stateB ⟶ (*getF stateB*, *getF stateA*) ∈ *learnL*
shows (*stateB*, *stateA*) ∈ *terminationLess F0 decisionVars learnL*
proof−
{
assume *appliedDecide stateA stateB decisionVars* ∨ *appliedUnitPropagate stateA stateB* ∨ *appliedBackjump stateA stateB*
with ‹*invariantsHoldInState stateA F0 decisionVars*›
have (*getM stateB*, *getM stateA*) ∈ *lexLessRestricted* (*vars F0* ∪ *decisionVars*)
using *trailIsDecreasedByDeciedUnitPropagateAndBackjump*
by *simp*
hence (*stateB*, *stateA*) ∈ *lexLessState F0 decisionVars*
unfolding *lexLessState-def*
by *simp*
hence (*stateB*, *stateA*) ∈ *terminationLess F0 decisionVars learnL*
unfolding *terminationLess-def*
by *simp*
}
moreover
{
assume *appliedLearn stateA stateB*
with ‹*appliedLearn stateA stateB* ⟶ (*getF stateB*, *getF stateA*) ∈ *learnL*›

have (*getF stateB*, *getF stateA*) ∈ *learnL*
by *simp*
moreover
from ‹*appliedLearn stateA stateB*›
have (*getM stateB*) = (*getM stateA*)
unfolding *appliedLearn-def*
by *auto*
ultimately
have (*stateB*, *stateA*) ∈ *learnLessState learnL*
unfolding *learnLessState-def*
by *simp*
hence (*stateB*, *stateA*) ∈ *terminationLess F0 decisionVars learnL*
unfolding *terminationLess-def*
by *simp*
}
ultimately
show *?thesis*
using ‹*transition stateA stateB decisionVars*›
unfolding *transition-def*
by *auto*
qed

The minimal states with respect to the termination ordering are final i.e., no further transition rules are applicable.

definition
isMinimalState stateMin F0 decisionVars learnL == (∀ *state*::*State*. (*state*, *stateMin*) ∉ *terminationLess F0 decisionVars learnL*)

lemma *minimalStatesAreFinal*:
fixes *stateA*::*State*
assumes ∗: ∀ (*stateA*::*State*) (*stateB*::*State*). *appliedLearn stateA stateB* ⟶ (*getF stateB*, *getF stateA*) ∈ *learnL* **and**
invariantsHoldInState state F0 decisionVars **and** *isMinimalState state F0 decisionVars learnL*
shows *isFinalState state decisionVars*
proof−
{
assume ¬ *?thesis*
then obtain *state′*::*State*
where *transition state state′ decisionVars*
unfolding *isFinalState-def*
by *auto*
with ‹*invariantsHoldInState state F0 decisionVars*› ∗
have (*state′*, *state*) ∈ *terminationLess F0 decisionVars learnL*
using *stateIsDecreasedByValidTransitions*[*of state F0 decisionVars state′ learnL*]
unfolding *transition-def*
by *auto*
with ‹*isMinimalState state F0 decisionVars learnL*›

have *False*
unfolding *isMinimalState-def*
by *auto*
}
thus *?thesis*
by *auto*
qed

We now prove that termination ordering is well founded. We start with two auxiliary lemmas.

lemma *wfLexLessState*:
fixes *decisionVars* :: *Variable set* **and** *F0* :: *Formula*
assumes *finite decisionVars*
shows *wf* (*lexLessState F0 decisionVars*)
unfolding *wf-eq-minimal*
proof−
show $\forall$ *Q state. state* $\in$ *Q* $\longrightarrow$ ($\exists$ *stateMin*$\in$*Q*. $\forall$ *state'*. (*state'*, *stateMin*) $\in$ *lexLessState F0 decisionVars* $\longrightarrow$ *state'* $\notin$ *Q*)
proof−
{
fix *Q* :: *State set* **and** *state* :: *State*
assume *state* $\in$ *Q*
let *?Q1* = {*M*::*LiteralTrail*. $\exists$ *state. state* $\in$ *Q* $\wedge$ (*getM state*) = *M*}
from ‹*state* $\in$ *Q*›
have *getM state* $\in$ *?Q1*
by *auto*
from ‹*finite decisionVars*›
have *finite* (*vars F0* $\cup$ *decisionVars*)
using *finiteVarsFormula*[*of F0*]
by *simp*
hence *wf* (*lexLessRestricted* (*vars F0* $\cup$ *decisionVars*))
using *wfLexLessRestricted*[*of vars F0* $\cup$ *decisionVars*]
by *simp*
with ‹*getM state* $\in$ *?Q1*›
obtain *Mmin* **where** *Mmin* $\in$ *?Q1* $\forall$ *M'*. (*M'*, *Mmin*) $\in$ *lexLess-Restricted* (*vars F0* $\cup$ *decisionVars*) $\longrightarrow$ *M'* $\notin$ *?Q1*
unfolding *wf-eq-minimal*
apply (*erule-tac x=?Q1* **in** *allE*)
apply (*erule-tac x=getM state* **in** *allE*)
by *auto*
from ‹*Mmin* $\in$ *?Q1*› **obtain** *stateMin*
where *stateMin* $\in$ *Q* (*getM stateMin*) = *Mmin*
by *auto*
have $\forall$ *state'*. (*state'*, *stateMin*) $\in$ *lexLessState F0 decisionVars* $\longrightarrow$ *state'* $\notin$ *Q*
proof
fix *state'*
show (*state'*, *stateMin*) $\in$ *lexLessState F0 decisionVars* $\longrightarrow$

state' ∉ *Q*
proof
assume (*state'*, *stateMin*) ∈ *lexLessState F0 decisionVars*
hence (*getM state'*, *getM stateMin*) ∈ *lexLessRestricted* (*vars F0* ∪ *decisionVars*)
unfolding *lexLessState-def*
by *auto*
from ‹∀ *M'*. (*M'*, *Mmin*) ∈ *lexLessRestricted* (*vars F0* ∪ *decisionVars*) ⟶ *M'* ∉ *?Q1*›
‹(*getM state'*, *getM stateMin*) ∈ *lexLessRestricted* (*vars F0* ∪ *decisionVars*)› ‹*getM stateMin* = *Mmin*›
have *getM state'* ∉ *?Q1*
by *simp*
with ‹*getM stateMin* = *Mmin*›
show *state'* ∉ *Q*
by *auto*
qed
qed
with ‹*stateMin* ∈ *Q*›
have ∃ *stateMin* ∈ *Q*. (∀ *state'*. (*state'*, *stateMin*) ∈ *lexLessState F0 decisionVars* ⟶ *state'* ∉ *Q*)
by *auto*
}
thus *?thesis*
by *auto*
qed
qed

lemma *wfLearnLessState*:
assumes *wf learnL*
shows *wf* (*learnLessState learnL*)
unfolding *wf-eq-minimal*
proof−
show ∀ *Q state*. *state* ∈ *Q* ⟶ (∃ *stateMin*∈*Q*. ∀ *state'*. (*state'*, *stateMin*) ∈ *learnLessState learnL* ⟶ *state'* ∉ *Q*)
proof−
{
fix *Q* :: *State set* **and** *state* :: *State*
assume *state* ∈ *Q*
let *?M* = (*getM state*)
let *?Q1* = {*f*::*Formula*. ∃ *state*. *state* ∈ *Q* ∧ (*getM state*) = *?M* ∧ (*getF state*) = *f*}
from ‹*state* ∈ *Q*›
have *getF state* ∈ *?Q1*
by *auto*
with ‹*wf learnL*›
obtain *FMin* **where** *FMin* ∈ *?Q1* ∀ *F'*. (*F'*, *FMin*) ∈ *learnL* ⟶ *F'* ∉ *?Q1*
unfolding *wf-eq-minimal*

apply (*erule-tac x=?Q1* **in** *allE*)
apply (*erule-tac x=getF state* **in** *allE*)
by *auto*
from ‹*FMin* ∈ *?Q1*› **obtain** *stateMin*
where *stateMin* ∈ *Q* (*getM stateMin*) = *?M getF stateMin* = *FMin*
by *auto*
have ∀ *state'*. (*state'*, *stateMin*) ∈ *learnLessState learnL* ⟶ *state'* ∉ *Q*
proof
fix *state'*
show (*state'*, *stateMin*) ∈ *learnLessState learnL* ⟶ *state'* ∉ *Q*
proof
assume (*state'*, *stateMin*) ∈ *learnLessState learnL*
with ‹*getM stateMin* = *?M*›
have *getM state'* = *getM stateMin* (*getF state'*, *getF stateMin*) ∈ *learnL*
unfolding *learnLessState-def*
by *auto*
from ‹∀ *F'*. (*F'*, *FMin*) ∈ *learnL* ⟶ *F'* ∉ *?Q1*›
‹(*getF state'*, *getF stateMin*) ∈ *learnL*› ‹*getF stateMin* = *FMin*›
have *getF state'* ∉ *?Q1*
by *simp*
with ‹*getM state'* = *getM stateMin*› ‹*getM stateMin* = *?M*›
show *state'* ∉ *Q*
by *auto*
qed
qed
with ‹*stateMin* ∈ *Q*›
have ∃ *stateMin* ∈ *Q*. (∀ *state'*. (*state'*, *stateMin*) ∈ *learnLessState learnL* ⟶ *state'* ∉ *Q*)
by *auto*
}
thus *?thesis*
by *auto*
qed
qed

Now we can prove the following key lemma which shows that the termination ordering is well founded.

lemma *wfTerminationLess*:
fixes *F0* :: *Formula* **and** *decisionVars* :: *Variable set*
assumes *finite decisionVars wf learnL*
shows *wf* (*terminationLess F0 decisionVars learnL*)
unfolding *wf-eq-minimal*
proof−
show ∀ *Q state*. *state* ∈ *Q* ⟶ (∃ *stateMin* ∈ *Q*. ∀ *state'*. (*state'*, *stateMin*) ∈ *terminationLess F0 decisionVars learnL* ⟶ *state'* ∉ *Q*)

proof −
{
fix *Q*::*State set*
fix *state*::*State*
assume *state* ∈ *Q*
have *wf* (*lexLessState F0 decisionVars*)
using *wfLexLessState*[*of decisionVars F0*]
using ‹*finite decisionVars*›
by *simp*
with ‹*state* ∈ *Q*› **obtain** *state0*
where *state0* ∈ *Q* ∀ *state'*. (*state'*, *state0*) ∈ *lexLessState F0 decisionVars* ⟶ *state'* ∉ *Q*
unfolding *wf-eq-minimal*
by *auto*
let *?Q0* = {*state*. *state* ∈ *Q* ∧ (*getM state*) = (*getM state0*)}
from ‹*state0* ∈ *Q*›
have *state0* ∈ *?Q0*
by *simp*
from ‹*wf learnL*›
have *wf* (*learnLessState learnL*)
using *wfLearnLessState*
by *simp*
with ‹*state0* ∈ *?Q0*› **obtain** *state1*
where *state1* ∈ *?Q0* ∀ *state'*. (*state'*, *state1*) ∈ *learnLessState learnL* ⟶ *state'* ∉ *?Q0*
unfolding *wf-eq-minimal*
apply (*erule-tac x=?Q0* **in** *allE*)
apply (*erule-tac x=state0* **in** *allE*)
by *auto*
from ‹*state1* ∈ *?Q0*›
have *state1* ∈ *Q* *getM state1* = *getM state0*
by *auto*
let *?stateMin* = *state1*
have ∀ *state'*. (*state'*, *?stateMin*) ∈ *terminationLess F0 decisionVars learnL* ⟶ *state'* ∉ *Q*
proof
fix *state'*
show (*state'*, *?stateMin*) ∈ *terminationLess F0 decisionVars learnL* ⟶ *state'* ∉ *Q*
proof
assume (*state'*, *?stateMin*) ∈ *terminationLess F0 decisionVars learnL*
hence
(*state'*, *?stateMin*) ∈ *lexLessState F0 decisionVars* ∨
(*state'*, *?stateMin*) ∈ *learnLessState learnL*
unfolding *terminationLess-def*
by *auto*
moreover
{

assume (*state′*, *?stateMin*) ∈ *lexLessState F0 decisionVars*
with ‹*getM state1* = *getM state0*›
have (*state′*, *state0*) ∈ *lexLessState F0 decisionVars*
unfolding *lexLessState-def*
by *simp*
with ‹∀ *state′*. (*state′*, *state0*) ∈ *lexLessState F0 decisionVars* ⟶ *state′* ∉ *Q*›
have *state′* ∉ *Q*
by *simp*
}
moreover
{
assume (*state′*, *?stateMin*) ∈ *learnLessState learnL*
with ‹∀ *state′*. (*state′*, *state1*) ∈ *learnLessState learnL* ⟶ *state′* ∉ *?Q0*›
have *state′* ∉ *?Q0*
by *simp*
from ‹(*state′*, *state1*) ∈ *learnLessState learnL*› ‹*getM state1* = *getM state0*›
have *getM state′* = *getM state0*
unfolding *learnLessState-def*
by *auto*
with ‹*state′* ∉ *?Q0*›
have *state′* ∉ *Q*
by *simp*
}
ultimately
show *state′* ∉ *Q*
by *auto*
qed
qed
with ‹*?stateMin* ∈ *Q*› **have** (∃ *stateMin* ∈ *Q*. ∀ *state′*. (*state′*, *stateMin*) ∈ *terminationLess F0 decisionVars learnL* ⟶ *state′* ∉ *Q*)
by *auto*
}
thus *?thesis*
by *simp*
qed
qed

Using the termination ordering we show that the transition relation is well founded on states reachable from initial state. The assumption for the *Learn* rule is neccessary.

theorem *wfTransitionRelation*:
fixes *decisionVars* :: *Variable set* **and** *F0* :: *Formula*
assumes *finite decisionVars* **and** *isInitialState state0 F0* **and**
∗: ∃ *learnL*::(*Formula* × *Formula*) *set*.
wf learnL ∧
(∀ *stateA stateB*. *appliedLearn stateA stateB* ⟶ (*getF stateB*,

getF stateA) ∈ *learnL*)
  shows *wf* {(*stateB*, *stateA*).
                  (*state0*, *stateA*) ∈ *transitionRelation decisionVars* ∧
(*transition stateA stateB decisionVars*)}

proof−
  from ∗ **obtain** *learnL*::(*Formula* × *Formula*) *set*
    where
    wf learnL **and**
    ∗∗: ∀ *stateA stateB*. *appliedLearn stateA stateB* ⟶ (*getF stateB*,
getF stateA) ∈ *learnL*
    by *auto*
  let *?rel* = {(*stateB*, *stateA*).
                  (*state0*, *stateA*) ∈ *transitionRelation decisionVars* ∧
(*transition stateA stateB decisionVars*)}
  let *?rel′*= *terminationLess F0 decisionVars learnL*

  have ∀ *x y*. (*x*, *y*) ∈ *?rel* ⟶ (*x*, *y*) ∈ *?rel′*
  proof−
    {
      fix *stateA*::*State* **and** *stateB*::*State*
      assume (*stateB*, *stateA*) ∈ *?rel*
      hence (*stateB*, *stateA*) ∈ *?rel′*
        using ‹*isInitialState state0 F0*›
        using *invariantsHoldInValidRunsFromInitialState*[*of state0 F0*
stateA decisionVars]
        using *stateIsDecreasedByValidTransitions*[*of stateA F0 deci-*
sionVars stateB] ∗∗
        by *simp*
    }
    thus *?thesis*
      by *simp*
  qed
  moreover
  have *wf ?rel′*
    using ‹*finite decisionVars*› ‹*wf learnL*›
    by (*rule wfTerminationLess*)
  ultimately
  show *?thesis*
    using *wellFoundedEmbed*[*of ?rel ?rel′*]
    by *simp*
qed

We will now give two corollaries of the previous theorem. First is a weak termination result that shows that there is a terminating run from every intial state to the final one.

corollary
  fixes *decisionVars* :: *Variable set* **and** *F0* :: *Formula* **and** *state0* ::
State

assumes *finite decisionVars* **and** *isInitialState state0 F0* **and**
∗: ∃ *learnL*::(*Formula* × *Formula*) *set*.
wf learnL ∧
(∀ *stateA stateB*. *appliedLearn stateA stateB* ⟶ (*getF stateB*, *getF stateA*) ∈ *learnL*)
shows ∃ *state*. (*state0*, *state*) ∈ *transitionRelation decisionVars* ∧ *isFinalState state decisionVars*
proof−
{
assume ¬ *?thesis*
let *?Q* = {*state*. (*state0*, *state*) ∈ *transitionRelation decisionVars*}
let *?rel* = {(*stateB*, *stateA*). (*state0*, *stateA*) ∈ *transitionRelation decisionVars* ∧
transition stateA stateB decisionVars}
have *state0* ∈ *?Q*
unfolding *transitionRelation-def*
by *simp*
hence ∃ *state*. *state* ∈ *?Q*
by *auto*

from *assms*
have *wf ?rel*
using *wfTransitionRelation*[*of decisionVars state0 F0*]
by *auto*
hence ∀ *Q*. (∃ *x*. *x* ∈ *Q*) ⟶ (∃ *stateMin* ∈ *Q*. ∀ *state*. (*state*, *stateMin*) ∈ *?rel* ⟶ *state* ∉ *Q*)
unfolding *wf-eq-minimal*
by *simp*
hence (∃ *x*. *x* ∈ *?Q*) ⟶ (∃ *stateMin* ∈ *?Q*. ∀ *state*. (*state*, *stateMin*) ∈ *?rel* ⟶ *state* ∉ *?Q*)
by *rule*
with ‹∃ *state*. *state* ∈ *?Q*›
have ∃ *stateMin* ∈ *?Q*. ∀ *state*. (*state*, *stateMin*) ∈ *?rel* ⟶ *state* ∉ *?Q*
by *simp*
then obtain *stateMin*
where *stateMin* ∈ *?Q* **and** ∀ *state*. (*state*, *stateMin*) ∈ *?rel* ⟶ *state* ∉ *?Q*
by *auto*

from ‹*stateMin* ∈ *?Q*›
have (*state0*, *stateMin*) ∈ *transitionRelation decisionVars*
by *simp*
with ‹¬ *?thesis*›
have ¬ *isFinalState stateMin decisionVars*
by *simp*
then obtain *state′*::*State*
where *transition stateMin state′ decisionVars*
unfolding *isFinalState-def*

by *auto*
have (*state'*, *stateMin*) ∈ *?rel*
using ‹(*state0*, *stateMin*) ∈ *transitionRelation decisionVars*›
‹*transition stateMin state' decisionVars*›
by *simp*
with ‹∀ *state*. (*state*, *stateMin*) ∈ *?rel* ⟶ *state* ∉ *?Q*›
have *state'* ∉ *?Q*
by *force*
moreover
from ‹(*state0*, *stateMin*) ∈ *transitionRelation decisionVars*› ‹*transition stateMin state' decisionVars*›
have *state'* ∈ *?Q*
unfolding *transitionRelation-def*
using *rtrancl-into-rtrancl*[*of state0 stateMin* {(*stateA*, *stateB*). *transition stateA stateB decisionVars*} *state'*]
by *simp*
ultimately
have *False*
by *simp*
}
thus *?thesis*
by *auto*
qed

Now we prove the final strong termination result which states that there cannot be infinite chains of transitions. If there is an infinite transition chain that starts from an initial state, its elements would for a set that would contain initial state and for every element of that set there would be another element of that set that is directly reachable from it. We show that no such set exists.

corollary *noInfiniteTransitionChains*:
fixes *F0*::*Formula* **and** *decisionVars*::*Variable set*
assumes *finite decisionVars* **and**
∗: ∃ *learnL*::(*Formula* × *Formula*) *set*.
wf learnL ∧
(∀ *stateA stateB*. *appliedLearn stateA stateB* ⟶ (*getF stateB*, *getF stateA*) ∈ *learnL*)
shows ¬ (∃ *Q*::(*State set*). ∃ *state0* ∈ *Q*. *isInitialState state0 F0* ∧

(∀ *state* ∈ *Q*. (∃ *state'* ∈ *Q*. *transition state state' decisionVars*))
)
proof−
{
assume ¬ *?thesis*
then obtain *Q*::*State set* **and** *state0*::*State*
where *isInitialState state0 F0 state0* ∈ *Q*

∀ *state* ∈ *Q*. (∃ *state′* ∈ *Q*. *transition state state′ decisionVars*)
by *auto*
let *?rel* = {(*stateB*, *stateA*). (*state0*, *stateA*) ∈ *transitionRelation decisionVars* ∧
transition stateA stateB decisionVars}
from ‹*finite decisionVars*› ‹*isInitialState state0 F0*› ∗
have *wf ?rel*
using *wfTransitionRelation*
by *simp*
hence *wfmin*: ∀ *Q x*. *x* ∈ *Q* ⟶
(∃ *z*∈*Q*. ∀ *y*. (*y*, *z*) ∈ *?rel* ⟶ *y* ∉ *Q*)
unfolding *wf-eq-minimal*
by *simp*
let *?Q* = {*state* ∈ *Q*. (*state0*, *state*) ∈ *transitionRelation decisionVars*}
from ‹*state0* ∈ *Q*›
have *state0* ∈ *?Q*
unfolding *transitionRelation-def*
by *simp*
with *wfmin*
obtain *stateMin*::*State*
where *stateMin* ∈ *?Q* **and** ∀ *y*. (*y*, *stateMin*) ∈ *?rel* ⟶ *y* ∉ *?Q*
apply (*erule-tac x*=*?Q* **in** *allE*)
by *auto*

from ‹*stateMin* ∈ *?Q*›
have *stateMin* ∈ *Q* (*state0*, *stateMin*) ∈ *transitionRelation decisionVars*
by *auto*
with ‹∀ *state* ∈ *Q*. (∃ *state′* ∈ *Q*. *transition state state′ decisionVars*)›
obtain *state′*::*State*
where *state′* ∈ *Q transition stateMin state′ decisionVars*
by *auto*

with ‹(*state0*, *stateMin*) ∈ *transitionRelation decisionVars*›
have (*state′*, *stateMin*) ∈ *?rel*
by *simp*
with ‹∀ *y*. (*y*, *stateMin*) ∈ *?rel* ⟶ *y* ∉ *?Q*›
have *state′* ∉ *?Q*
by *force*

from ‹*state′* ∈ *Q*› ‹(*state0*, *stateMin*) ∈ *transitionRelation decisionVars*›
‹*transition stateMin state′ decisionVars*›
have *state′* ∈ *?Q*
unfolding *transitionRelation-def*
using *rtrancl-into-rtrancl*[*of state0 stateMin* {(*stateA*, *stateB*). *transition stateA stateB decisionVars*} *state′*]

by *simp*
with ‹*state'* ∉ *?Q*›
have *False*
by *simp*
}
thus *?thesis*
by *force*
qed

6.5 Completeness

In this section we will first show that each final state is either *SAT* or *UNSAT* state.

lemma *finalNonConflictState*:
fixes *state*::*State* **and** *FO* :: *Formula*
assumes
¬ *applicableDecide state decisionVars*
shows *vars* (*elements* (*getM state*)) ⊇ *decisionVars*
proof
fix *x* :: *Variable*
let *?l* = *Pos x*
assume *x* ∈ *decisionVars*
hence *var ?l* = *x* **and** *var ?l* ∈ *decisionVars* **and** *var* (*opposite ?l*) ∈ *decisionVars*
by *auto*
with ‹¬ *applicableDecide state decisionVars*›
have *literalTrue ?l* (*elements* (*getM state*)) ∨ *literalFalse ?l* (*elements* (*getM state*))
unfolding *applicableDecideCharacterization*
by *force*
with ‹*var ?l* = *x*›
show *x* ∈ *vars* (*elements* (*getM state*))
using *valuationContainsItsLiteralsVariable*[*of ?l elements* (*getM state*)]
using *valuationContainsItsLiteralsVariable*[*of opposite ?l elements* (*getM state*)]
by *auto*
qed

lemma *finalConflictingState*:
fixes *state* :: *State*
assumes
InvariantUniq (*getM state*) **and**
InvariantConsistent (*getM state*) **and**
InvariantImpliedLiterals (*getF state*) (*getM state*)
¬ *applicableBackjump state* **and**
formulaFalse (*getF state*) (*elements* (*getM state*))
shows

```
  decisions (getM state) = []
proof−
  from ‹InvariantUniq (getM state)›
  have uniq (elements (getM state))
    unfolding InvariantUniq-def
    .
  from ‹InvariantConsistent (getM state)›
  have consistent (elements (getM state))
    unfolding InvariantConsistent-def
    .

  let ?c = oppositeLiteralList (decisions (getM state))
  {
    assume ¬ ?thesis
    hence ?c ≠ []
      using oppositeLiteralListNonempty[of decisions (getM state)]
      by simp
    moreover
    have clauseFalse ?c (elements (getM state))
    proof−
      {
        fix l::Literal
        assume l el ?c
        hence opposite l el decisions (getM state)
          using literalElListIffOppositeLiteralElOppositeLiteralList [of l
?c]
          by simp
        hence literalFalse l (elements (getM state))
          using markedElementsAreElements[of opposite l getM state]
          by simp
      }
      thus ?thesis
          using clauseFalseIffAllLiteralsAreFalse[of ?c elements (getM
state)]
        by simp
    qed
    moreover
    let ?l = getLastAssertedLiteral (oppositeLiteralList ?c) (elements
(getM state))
     have isLastAssertedLiteral ?l (oppositeLiteralList ?c) (elements
(getM state))
      using ‹InvariantUniq (getM state)›
    using getLastAssertedLiteralCharacterization[of ?c elements (getM
state)]
        ‹?c ≠ []› ‹clauseFalse ?c (elements (getM state))›
      unfolding InvariantUniq-def
      by simp
    moreover
    have ∀ l. l el ?c ⟶ (opposite l) el (decisions (getM state))
```

proof −
{
fix *l*::*Literal*
assume *l el ?c*
hence (*opposite l*) *el* (*oppositeLiteralList ?c*)
using *literalElListIffOppositeLiteralElOppositeLiteralList*[*of l ?c*]
by *simp*
}
thus *?thesis*
by *simp*
qed
ultimately
have ∃ *level*. (*isBackjumpLevel level* (*opposite ?l*) *?c* (*getM state*))
using ‹*uniq* (*elements* (*getM state*))›
using *allDecisionsThenExistsBackjumpLevel*[*of getM state ?c opposite ?l*]
by *simp*
then obtain *level*::*nat*
where *isBackjumpLevel level* (*opposite ?l*) *?c* (*getM state*)
by *auto*
with ‹*consistent* (*elements* (*getM state*))› ‹*uniq* (*elements* (*getM state*))› ‹*clauseFalse ?c* (*elements* (*getM state*))›
have *isUnitClause ?c* (*opposite ?l*) (*elements* (*prefixToLevel level* (*getM state*)))
using *isBackjumpLevelEnsuresIsUnitInPrefix*[*of getM state ?c level opposite ?l*]
by *simp*
moreover
have *formulaEntailsClause* (*getF state*) *?c*
proof −
from ‹*clauseFalse ?c* (*elements* (*getM state*))› ‹*consistent* (*elements* (*getM state*))›
have ¬ *clauseTautology ?c*
using *tautologyNotFalse*[*of ?c elements* (*getM state*)]
by *auto*

from ‹*formulaFalse* (*getF state*) (*elements* (*getM state*))› ‹*InvariantImpliedLiterals* (*getF state*) (*getM state*)›
have ¬ *satisfiable* ((*getF state*) @ *val2form* (*decisions* (*getM state*)))
using *InvariantImpliedLiteralsAndFormulaFalseThenFormulaAndDecisionsAreNotSatisfiable*
by *simp*
hence ¬ *satisfiable* ((*getF state*) @ *val2form* (*oppositeLiteralList ?c*))
by *simp*
with ‹¬ *clauseTautology ?c*›
show *?thesis*

using *unsatisfiableFormulaWithSingleLiteralClauses*
by *simp*
qed
moreover
have *var ?l* ∈ *vars* (*getF state*) ∪ *vars* (*elements* (*getM state*))
proof−
from ‹*isLastAssertedLiteral ?l* (*oppositeLiteralList ?c*) (*elements* (*getM state*))›
have *?l el* (*oppositeLiteralList ?c*)
unfolding *isLastAssertedLiteral-def*
by *simp*
hence *literalTrue ?l* (*elements* (*getM state*))
by (*simp add*: *markedElementsAreElements*)
hence *var ?l* ∈ *vars* (*elements* (*getM state*))
using *valuationContainsItsLiteralsVariable*[*of ?l elements* (*getM state*)]
by *simp*
thus *?thesis*
by *simp*
qed
moreover
have *0* ≤ *level level* < (*currentLevel* (*getM state*))
proof−
from ‹*isBackjumpLevel level* (*opposite ?l*) *?c* (*getM state*)›
have *0* ≤ *level level* < (*elementLevel ?l* (*getM state*))
unfolding *isBackjumpLevel-def*
by *auto*
thus *0* ≤ *level level* < (*currentLevel* (*getM state*))
using *elementLevelLeqCurrentLevel*[*of ?l getM state*]
by *auto*
qed
ultimately
have *applicableBackjump state*
unfolding *applicableBackjumpCharacterization*
by *force*
with ‹¬ *applicableBackjump state*›
have *False*
by *simp*
}
thus *?thesis*
by *auto*
qed

lemma *finalStateCharacterizationLemma*:
fixes *state* :: *State*
assumes
InvariantUniq (*getM state*) **and**
InvariantConsistent (*getM state*) **and**
InvariantImpliedLiterals (*getF state*) (*getM state*)

¬ *applicableDecide state decisionVars* **and**
¬ *applicableBackjump state*
shows
(¬ *formulaFalse* (*getF state*) (*elements* (*getM state*)) ∧ *vars* (*elements* (*getM state*)) ⊇ *decisionVars*) ∨
(*formulaFalse* (*getF state*) (*elements* (*getM state*)) ∧ *decisions* (*getM state*) = [])
proof (*cases formulaFalse* (*getF state*) (*elements* (*getM state*)))
case *True*
hence *decisions* (*getM state*) = []
using *assms*
using *finalConflictingState*
by *auto*
with *True*
show *?thesis*
by *simp*
next
case *False*
hence *vars* (*elements* (*getM state*)) ⊇ *decisionVars*
using *assms*
using *finalNonConflictState*
by *auto*
with *False*
show *?thesis*
by *simp*
qed

theorem *finalStateCharacterization*:
fixes *F0* :: *Formula* **and** *decisionVars* :: *Variable set* **and** *state0* :: *State* **and** *state* :: *State*
assumes
isInitialState state0 F0 **and**
(*state0*, *state*) ∈ *transitionRelation decisionVars* **and**
isFinalState state decisionVars
shows
(¬ *formulaFalse* (*getF state*) (*elements* (*getM state*)) ∧ *vars* (*elements* (*getM state*)) ⊇ *decisionVars*) ∨
(*formulaFalse* (*getF state*) (*elements* (*getM state*)) ∧ *decisions* (*getM state*) = [])

proof−
from ‹*isInitialState state0 F0*› ‹(*state0*, *state*) ∈ *transitionRelation decisionVars*›
have *invariantsHoldInState state F0 decisionVars*
using *invariantsHoldInValidRunsFromInitialState*
by *simp*
hence
∗: *InvariantUniq* (*getM state*)

InvariantConsistent (*getM state*)
InvariantImpliedLiterals (*getF state*) (*getM state*)
unfolding *invariantsHoldInState-def*
by *auto*

from ‹*isFinalState state decisionVars*›
have ∗∗:
¬ *applicableBackjump state*
¬ *applicableDecide state decisionVars*
unfolding *finalStateNonApplicable*
by *auto*

from ∗ ∗∗
show *?thesis*
using *finalStateCharacterizationLemma*[*of state decisionVars*]
by *simp*
qed

Completeness theorems are easy consequences of this characterization and soundness.

theorem *completenessForSAT*:
fixes *F0* :: *Formula* **and** *decisionVars* :: *Variable set* **and** *state0* :: *State* **and** *state* :: *State*
assumes
satisfiable F0 **and**

isInitialState state0 F0 **and**
(*state0*, *state*) ∈ *transitionRelation decisionVars* **and**
isFinalState state decisionVars
shows ¬ *formulaFalse* (*getF state*) (*elements* (*getM state*)) ∧ *vars* (*elements* (*getM state*)) ⊇ *decisionVars*

proof−
from *assms*
have ∗: (¬ *formulaFalse* (*getF state*) (*elements* (*getM state*)) ∧ *vars* (*elements* (*getM state*)) ⊇ *decisionVars*) ∨
(*formulaFalse* (*getF state*) (*elements* (*getM state*)) ∧ *decisions* (*getM state*) = [])
using *finalStateCharacterization*[*of state0 F0 state decisionVars*]
by *auto*
{
assume *formulaFalse* (*getF state*) (*elements* (*getM state*))
with ∗
have *formulaFalse* (*getF state*) (*elements* (*getM state*)) *decisions* (*getM state*) = []
by *auto*
with *assms*
have ¬ *satisfiable F0*
using *soundnessForUNSAT*

by *simp*
with ‹*satisfiable F0*›
have *False*
by *simp*
}
with ∗ **show** *?thesis*
by *auto*
qed

theorem *completenessForUNSAT*:
fixes *F0* :: *Formula* **and** *decisionVars* :: *Variable set* **and** *state0* :: *State* **and** *state* :: *State*
assumes
vars F0 ⊆ *decisionVars* **and**

¬ *satisfiable F0* **and**

isInitialState state0 F0 **and**
(*state0*, *state*) ∈ *transitionRelation decisionVars* **and**
isFinalState state decisionVars
shows
formulaFalse (*getF state*) (*elements* (*getM state*)) ∧ *decisions* (*getM state*) = []

proof−
from *assms*
have ∗:
(¬ *formulaFalse* (*getF state*) (*elements* (*getM state*)) ∧ *vars* (*elements* (*getM state*)) ⊇ *decisionVars*) ∨
(*formulaFalse* (*getF state*) (*elements* (*getM state*)) ∧ *decisions* (*getM state*) = [])
using *finalStateCharacterization*[*of state0 F0 state decisionVars*]
by *auto*
{
assume ¬ *formulaFalse* (*getF state*) (*elements* (*getM state*))
with ∗
have ¬ *formulaFalse* (*getF state*) (*elements* (*getM state*)) *vars* (*elements* (*getM state*)) ⊇ *decisionVars*
by *auto*
with *assms*
have *satisfiable F0*
using *soundnessForSAT*[*of F0 decisionVars state0 state*]
unfolding *satisfiable-def*
by *auto*
with ‹¬ *satisfiable F0*›
have *False*
by *simp*
}

with $*$ **show** *?thesis*
  by *auto*
qed

theorem *partialCorrectness*:
  fixes *F0* :: *Formula* **and** *decisionVars* :: *Variable set* **and** *state0* :: *State* **and** *state* :: *State*
  assumes
  vars F0 $\subseteq$ *decisionVars* **and**

  isInitialState state0 F0 **and**
  (*state0*, *state*) $\in$ *transitionRelation decisionVars* **and**
  isFinalState state decisionVars
  shows
  satisfiable F0 = ($\neg$ *formulaFalse* (*getF state*) (*elements* (*getM state*)))

using *assms*
using *completenessForUNSAT*[*of F0 decisionVars state0 state*]
using *completenessForSAT*[*of F0 state0 state decisionVars*]
by *auto*

end

7 Transition system of Krstić and Goel.

theory *KrsticGoel*
imports *SatSolverVerification*
begin

This theory formalizes the transition rule system given by Krstić and Goel in [1]. Some rules of the system are generalized a bit, so that the system can model some more general solvers (e.g., SMT solvers).

7.1 Specification

record *State* =
getF :: *Formula*
getM :: *LiteralTrail*
getConflictFlag :: *bool*
getC :: *Clause*

definition
appliedDecide:: *State* $\Rightarrow$ *State* $\Rightarrow$ *Variable set* $\Rightarrow$ *bool*
where
appliedDecide stateA stateB decisionVars ==
  $\exists$ *l*.

$(var\ l) \in decisionVars\ \wedge$
$\neg\ l\ el\ (elements\ (getM\ stateA))\ \wedge$
$\neg\ opposite\ l\ el\ (elements\ (getM\ stateA))\ \wedge$

$getF\ stateB = getF\ stateA\ \wedge$
$getM\ stateB = getM\ stateA\ @\ [(l,\ True)]\ \wedge$
$getConflictFlag\ stateB = getConflictFlag\ stateA\ \wedge$
$getC\ stateB = getC\ stateA$

definition
applicableDecide :: *State* ⇒ *Variable set* ⇒ *bool*
where
applicableDecide state decisionVars == ∃ *state′*. *appliedDecide state state′ decisionVars*

Notice that the given UnitPropagate description is weaker than in original [1] paper. Namely, propagation can be done over a clause that is not a member of the formula, but is entailed by it. The condition imposed on the variable of the unit literal is necessary to ensure the termination.

definition
appliedUnitPropagate :: *State* ⇒ *State* ⇒ *Formula* ⇒ *Variable set* ⇒ *bool*
where
appliedUnitPropagate stateA stateB F0 decisionVars ==
∃ (*uc*::*Clause*) (*ul*::*Literal*).
$formulaEntailsClause\ (getF\ stateA)\ uc\ \wedge$
$(var\ ul) \in decisionVars \cup vars\ F0\ \wedge$
$isUnitClause\ uc\ ul\ (elements\ (getM\ stateA))\ \wedge$

$getF\ stateB = getF\ stateA\ \wedge$
$getM\ stateB = getM\ stateA\ @\ [(ul,\ False)]\ \wedge$
$getConflictFlag\ stateB = getConflictFlag\ stateA\ \wedge$
$getC\ stateB = getC\ stateA$

definition
applicableUnitPropagate :: *State* ⇒ *Formula* ⇒ *Variable set* ⇒ *bool*
where
applicableUnitPropagate state F0 decisionVars == ∃ *state′*. *appliedUnitPropagate state state′ F0 decisionVars*

Notice, also, that $Conflict$ can be performed for a clause that is not a member of the formula.

definition
appliedConflict :: *State* ⇒ *State* ⇒ *bool*
where
appliedConflict stateA stateB ==
∃ *clause*.

$getConflictFlag\ stateA = False\ \wedge$
$formulaEntailsClause\ (getF\ stateA)\ clause\ \wedge$
$clauseFalse\ clause\ (elements\ (getM\ stateA))\ \wedge$

$getF\ stateB = getF\ stateA\ \wedge$
$getM\ stateB = getM\ stateA\ \wedge$
$getConflictFlag\ stateB = True\ \wedge$
$getC\ stateB = clause$

definition
$applicableConflict :: State \Rightarrow bool$
where
$applicableConflict\ state == \exists\ state'.\ appliedConflict\ state\ state'$

Notice, also, that the explanation can be done over a reason clause that is not a member of the formula, but is only entailed by it.

definition
$appliedExplain :: State \Rightarrow State \Rightarrow bool$
where
$appliedExplain\ stateA\ stateB ==$
$\exists\ l\ reason.$
$getConflictFlag\ stateA = True\ \wedge$
$l\ el\ getC\ stateA\ \wedge$
$formulaEntailsClause\ (getF\ stateA)\ reason\ \wedge$
$isReason\ reason\ (opposite\ l)\ (elements\ (getM\ stateA))\ \wedge$

$getF\ stateB = getF\ stateA\ \wedge$
$getM\ stateB = getM\ stateA\ \wedge$
$getConflictFlag\ stateB = True\ \wedge$
$getC\ stateB = resolve\ (getC\ stateA)\ reason\ l$

definition
$applicableExplain :: State \Rightarrow bool$
where
$applicableExplain\ state == \exists\ state'.\ appliedExplain\ state\ state'$

definition
$appliedLearn :: State \Rightarrow State \Rightarrow bool$
where
$appliedLearn\ stateA\ stateB ==$
$getConflictFlag\ stateA = True\ \wedge$
$\neg\ getC\ stateA\ el\ getF\ stateA\ \wedge$

$getF\ stateB = getF\ stateA\ @\ [getC\ stateA]\ \wedge$
$getM\ stateB = getM\ stateA\ \wedge$
$getConflictFlag\ stateB = True\ \wedge$
$getC\ stateB = getC\ stateA$

definition
applicableLearn :: *State* ⇒ *bool*
where
applicableLearn state == ∃ *state′*. *appliedLearn state state′*

Since unit propagation can be done over non-member clauses, it is not required that the conflict clause is learned before the *Backjump* is applied.

definition
appliedBackjump :: *State* ⇒ *State* ⇒ *bool*
where
appliedBackjump stateA stateB ==
 ∃ *l level*.
  getConflictFlag stateA = *True* ∧
  isBackjumpLevel level l (*getC stateA*) (*getM stateA*) ∧

  getF stateB = *getF stateA* ∧
  getM stateB = *prefixToLevel level* (*getM stateA*) @ [(*l*, *False*)] ∧
  getConflictFlag stateB = *False* ∧
  getC stateB = []

definition
applicableBackjump :: *State* ⇒ *bool*
where
applicableBackjump state == ∃ *state′*. *appliedBackjump state state′*

Solving starts with the initial formula, the empty trail and in non conflicting state.

definition
isInitialState :: *State* ⇒ *Formula* ⇒ *bool*
where
isInitialState state F0 ==
  getF state = *F0* ∧
  getM state = [] ∧
  getConflictFlag state = *False* ∧
  getC state = []

Transitions are preformed only by using given rules.

definition
transition :: *State* ⇒ *State* ⇒ *Formula* ⇒ *Variable set* ⇒ *bool*
where
transition stateA stateB F0 decisionVars==
  appliedDecide stateA stateB decisionVars ∨
  appliedUnitPropagate stateA stateB F0 decisionVars ∨
  appliedConflict stateA stateB ∨
  appliedExplain stateA stateB ∨
  appliedLearn stateA stateB ∨

appliedBackjump *stateA stateB*

Transition relation is obtained by applying transition rules iteratively. It is defined using a reflexive-transitive closure.

definition
transitionRelation F0 decisionVars == ({(*stateA*, *stateB*). *transition stateA stateB F0 decisionVars*})^*

Final state is one in which no rules apply

definition
isFinalState :: *State* ⇒ *Formula* ⇒ *Variable set* ⇒ *bool*
where
isFinalState state F0 decisionVars == ¬ (∃ *state'*. *transition state state' F0 decisionVars*)

The following several lemmas establish conditions for applicability of different rules.

lemma *applicableDecideCharacterization*:
  fixes *stateA*::*State*
  shows *applicableDecide stateA decisionVars* =
  (∃ *l*.
      (*var l*) ∈ *decisionVars* ∧
      ¬ *l el* (*elements* (*getM stateA*)) ∧
      ¬ *opposite l el* (*elements* (*getM stateA*)))
   (**is** *?lhs* = *?rhs*)
proof
  assume *?rhs*
  then obtain *l* **where**
   ∗: (*var l*) ∈ *decisionVars* ¬ *l el* (*elements* (*getM stateA*)) ¬ *opposite l el* (*elements* (*getM stateA*))
    unfolding *applicableDecide-def*
    by *auto*
  let *?stateB* = *stateA*⦇ *getM* := (*getM stateA*) @ [(*l*, *True*)] ⦈
  from ∗ **have** *appliedDecide stateA ?stateB decisionVars*
    unfolding *appliedDecide-def*
    by *auto*
  thus *?lhs*
    unfolding *applicableDecide-def*
    by *auto*
next
  assume *?lhs*
  then obtain *stateB l*
    where (*var l*) ∈ *decisionVars* ¬ *l el* (*elements* (*getM stateA*))
    ¬ *opposite l el* (*elements* (*getM stateA*))
    unfolding *applicableDecide-def*
    unfolding *appliedDecide-def*
    by *auto*
  thus *?rhs*
    by *auto*

```
qed

lemma applicableUnitPropagateCharacterization:
  fixes stateA::State and F0::Formula
  shows applicableUnitPropagate stateA F0 decisionVars =
  (∃ (uc::Clause) (ul::Literal).
        formulaEntailsClause (getF stateA) uc ∧
        (var ul) ∈ decisionVars ∪ vars F0 ∧
        isUnitClause uc ul (elements (getM stateA)))
   (is ?lhs = ?rhs)
proof
  assume ?rhs
  then obtain ul uc
    where *:
    formulaEntailsClause (getF stateA) uc
    (var ul) ∈ decisionVars ∪ vars F0
    isUnitClause uc ul (elements (getM stateA))
    unfolding applicableUnitPropagate-def
    by auto
  let ?stateB = stateA(| getM := getM stateA @ [(ul, False)] |)
  from * have appliedUnitPropagate stateA ?stateB F0 decisionVars
    unfolding appliedUnitPropagate-def
    by auto
  thus ?lhs
    unfolding applicableUnitPropagate-def
    by auto
next
  assume ?lhs
  then obtain stateB uc ul
    where
     formulaEntailsClause (getF stateA) uc
    (var ul) ∈ decisionVars ∪ vars F0
    isUnitClause uc ul (elements (getM stateA))
    unfolding applicableUnitPropagate-def
    unfolding appliedUnitPropagate-def
    by auto
  thus ?rhs
    by auto
qed


lemma applicableBackjumpCharacterization:
  fixes stateA::State
  shows applicableBackjump stateA =
     (∃ l level.
         getConflictFlag stateA = True ∧
         isBackjumpLevel level l (getC stateA) (getM stateA)
     ) (is ?lhs = ?rhs)
proof
```

assume *?rhs*
then obtain *l level*
where ∗:
getConflictFlag stateA = *True*
isBackjumpLevel level l (*getC stateA*) (*getM stateA*)
unfolding *applicableBackjump-def*
by *auto*
let *?stateB* = *stateA*⦇ *getM* := *prefixToLevel level* (*getM stateA*) @ [(*l*, *False*)],
getConflictFlag := *False*,
getC := [] ⦈
from ∗ **have** *appliedBackjump stateA ?stateB*
unfolding *appliedBackjump-def*
by *auto*
thus *?lhs*
unfolding *applicableBackjump-def*
by *auto*
next
assume *?lhs*
then obtain *stateB l level*
where *getConflictFlag stateA* = *True*
isBackjumpLevel level l (*getC stateA*) (*getM stateA*)
unfolding *applicableBackjump-def*
unfolding *appliedBackjump-def*
by *auto*
thus *?rhs*
by *auto*
qed

lemma *applicableExplainCharacterization*:
fixes *stateA*::*State*
shows *applicableExplain stateA* =
(∃ *l reason*.
getConflictFlag stateA = *True* ∧
l el getC stateA ∧
formulaEntailsClause (*getF stateA*) *reason* ∧
isReason reason (*opposite l*) (*elements* (*getM stateA*))
)
(**is** *?lhs* = *?rhs*)
proof
assume *?rhs*
then obtain *l reason*
where ∗:
getConflictFlag stateA = *True*
l el (*getC stateA*) *formulaEntailsClause* (*getF stateA*) *reason*
isReason reason (*opposite l*) (*elements* (*getM stateA*))
unfolding *applicableExplain-def*
by *auto*
let *?stateB* = *stateA*⦇ *getC* := *resolve* (*getC stateA*) *reason l* ⦈

```
  from * have appliedExplain stateA ?stateB
    unfolding appliedExplain-def
    by auto
  thus ?lhs
    unfolding applicableExplain-def
    by auto
next
  assume ?lhs
  then obtain stateB l reason
    where
    getConflictFlag stateA = True
    l el getC stateA formulaEntailsClause (getF stateA) reason
    isReason reason (opposite l) (elements (getM stateA))
    unfolding applicableExplain-def
    unfolding appliedExplain-def
    by auto
  thus ?rhs
    by auto
qed

lemma applicableConflictCharacterization:
  fixes stateA::State
  shows applicableConflict stateA =
    (∃ clause.
       getConflictFlag stateA = False ∧
       formulaEntailsClause (getF stateA) clause ∧
       clauseFalse clause (elements (getM stateA))) (is ?lhs = ?rhs)
proof
  assume ?rhs
  then obtain clause
    where *:
    getConflictFlag stateA = False formulaEntailsClause (getF stateA)
clause clauseFalse clause (elements (getM stateA))
    unfolding applicableConflict-def
    by auto
  let ?stateB = stateA(| getC := clause,
                         getConflictFlag := True |)
  from * have appliedConflict stateA ?stateB
    unfolding appliedConflict-def
    by auto
  thus ?lhs
    unfolding applicableConflict-def
    by auto
next
  assume ?lhs
  then obtain stateB clause
    where
    getConflictFlag stateA = False
    formulaEntailsClause (getF stateA) clause
```

```
    clauseFalse clause (elements (getM stateA))
    unfolding applicableConflict-def
    unfolding appliedConflict-def
    by auto
  thus ?rhs
    by auto
qed

lemma applicableLearnCharacterization:
  fixes stateA::State
  shows applicableLearn stateA =
          (getConflictFlag stateA = True ∧
           ¬ getC stateA el getF stateA) (is ?lhs = ?rhs)
proof
  assume ?rhs
  hence *: getConflictFlag stateA = True ¬ getC stateA el getF stateA
    unfolding applicableLearn-def
    by auto
  let ?stateB = stateA(| getF := getF stateA @ [getC stateA]|)
  from * have appliedLearn stateA ?stateB
    unfolding appliedLearn-def
    by auto
  thus ?lhs
    unfolding applicableLearn-def
    by auto
next
  assume ?lhs
  then obtain stateB
    where
    getConflictFlag stateA = True ¬ (getC stateA) el (getF stateA)
    unfolding applicableLearn-def
    unfolding appliedLearn-def
    by auto
  thus ?rhs
    by auto
qed
```

Final states are the ones where no rule is applicable.

```
lemma finalStateNonApplicable:
  fixes state::State
  shows isFinalState state F0 decisionVars =
          (¬ applicableDecide state decisionVars ∧
           ¬ applicableUnitPropagate state F0 decisionVars ∧
           ¬ applicableBackjump state ∧
           ¬ applicableLearn state ∧
           ¬ applicableConflict state ∧
           ¬ applicableExplain state)
unfolding isFinalState-def
unfolding transition-def
```

unfolding *applicableDecide-def*
unfolding *applicableUnitPropagate-def*
unfolding *applicableBackjump-def*
unfolding *applicableLearn-def*
unfolding *applicableConflict-def*
unfolding *applicableExplain-def*
by *auto*

7.2 Invariants

Invariants that are relevant for the rest of correctness proof.

definition
invariantsHoldInState :: *State* ⇒ *Formula* ⇒ *Variable set* ⇒ *bool*
where
invariantsHoldInState state F0 decisionVars ==
  InvariantVarsM (*getM state*) *F0 decisionVars* ∧
  InvariantVarsF (*getF state*) *F0 decisionVars* ∧
  InvariantConsistent (*getM state*) ∧
  InvariantUniq (*getM state*) ∧
  InvariantReasonClauses (*getF state*) (*getM state*) ∧
  InvariantEquivalent F0 (*getF state*) ∧
  InvariantCFalse (*getConflictFlag state*) (*getM state*) (*getC state*) ∧
  InvariantCEntailed (*getConflictFlag state*) (*getF state*) (*getC state*)

Invariants hold in initial states

lemma *invariantsHoldInInitialState*:
  fixes *state* :: *State* **and** *F0* :: *Formula*
  assumes *isInitialState state F0*
  shows *invariantsHoldInState state F0 decisionVars*
using *assms*
by (*auto simp add*:
  isInitialState-def
  invariantsHoldInState-def
  InvariantVarsM-def
  InvariantVarsF-def
  InvariantConsistent-def
  InvariantUniq-def
  InvariantReasonClauses-def
  InvariantEquivalent-def equivalentFormulae-def
  InvariantCFalse-def
  InvariantCEntailed-def
)

Valid transitions preserve invariants.

lemma *transitionsPreserveInvariants*:
  fixes *stateA*::*State* **and** *stateB*::*State*
  assumes *transition stateA stateB F0 decisionVars* **and**

invariantsHoldInState stateA F0 decisionVars
shows *invariantsHoldInState stateB F0 decisionVars*
proof−
from ‹*invariantsHoldInState stateA F0 decisionVars*›
have
InvariantVarsM (*getM stateA*) *F0 decisionVars* **and**
InvariantVarsF (*getF stateA*) *F0 decisionVars* **and**
InvariantConsistent (*getM stateA*) **and**
InvariantUniq (*getM stateA*) **and**
InvariantReasonClauses (*getF stateA*) (*getM stateA*) **and**
InvariantEquivalent F0 (*getF stateA*) **and**
InvariantCFalse (*getConflictFlag stateA*) (*getM stateA*) (*getC stateA*) **and**
InvariantCEntailed (*getConflictFlag stateA*) (*getF stateA*) (*getC stateA*)
unfolding *invariantsHoldInState-def*
by *auto*
{
assume *appliedDecide stateA stateB decisionVars*
then obtain *l*::*Literal* **where**
(*var l*) ∈ *decisionVars*
¬ *literalTrue l* (*elements* (*getM stateA*))
¬ *literalFalse l* (*elements* (*getM stateA*))
getM stateB = *getM stateA* @ [(*l*, *True*)]
getF stateB = *getF stateA*
getConflictFlag stateB = *getConflictFlag stateA*
getC stateB = *getC stateA*
unfolding *appliedDecide-def*
by *auto*

from ‹¬ *literalTrue l* (*elements* (*getM stateA*))› ‹¬ *literalFalse l* (*elements* (*getM stateA*))›
have ∗: *var l* ∉ *vars* (*elements* (*getM stateA*))
using *variableDefinedImpliesLiteralDefined*[*of l elements* (*getM stateA*)]
by *simp*

have *InvariantVarsM* (*getM stateB*) *F0 decisionVars*
using ‹*getF stateB* = *getF stateA*›
‹*getM stateB* = *getM stateA* @ [(*l*, *True*)]›
‹*InvariantVarsM* (*getM stateA*) *F0 decisionVars*›
‹*var l* ∈ *decisionVars*›
InvariantVarsMAfterDecide [*of getM stateA F0 decisionVars l getM stateB*]
by *simp*
moreover
have *InvariantVarsF* (*getF stateB*) *F0 decisionVars*
using ‹*getF stateB* = *getF stateA*›
‹*InvariantVarsF* (*getF stateA*) *F0 decisionVars*›

by *simp*
moreover
have *InvariantConsistent* (*getM stateB*)
using ‹*getM stateB* = *getM stateA* @ [(*l*, *True*)]›
‹*InvariantConsistent* (*getM stateA*)›
‹*var l* ∉ *vars* (*elements* (*getM stateA*))›
InvariantConsistentAfterDecide[*of getM stateA l getM stateB*]
by *simp*
moreover
have *InvariantUniq* (*getM stateB*)
using ‹*getM stateB* = *getM stateA* @ [(*l*, *True*)]›
‹*InvariantUniq* (*getM stateA*)›
‹*var l* ∉ *vars* (*elements* (*getM stateA*))›
InvariantUniqAfterDecide[*of getM stateA l getM stateB*]
by *simp*
moreover
have *InvariantReasonClauses* (*getF stateB*) (*getM stateB*)
using ‹*getF stateB* = *getF stateA*›
‹*getM stateB* = *getM stateA* @ [(*l*, *True*)]›
‹*InvariantUniq* (*getM stateA*)›
‹*InvariantReasonClauses* (*getF stateA*) (*getM stateA*)›
using *InvariantReasonClausesAfterDecide*[*of getF stateA getM stateA getM stateB l*]
by *simp*
moreover
have *InvariantEquivalent F0* (*getF stateB*)
using ‹*getF stateB* = *getF stateA*›
‹*InvariantEquivalent F0* (*getF stateA*)›
by *simp*
moreover
have *InvariantCFalse* (*getConflictFlag stateB*) (*getM stateB*) (*getC stateB*)
using ‹*getM stateB* = *getM stateA* @ [(*l*, *True*)]›
‹*getConflictFlag stateB* = *getConflictFlag stateA*›
‹*getC stateB* = *getC stateA*›
‹*InvariantCFalse* (*getConflictFlag stateA*) (*getM stateA*) (*getC stateA*)›
InvariantCFalseAfterDecide[*of getConflictFlag stateA getM stateA getC stateA getM stateB l*]
by *simp*
moreover
have *InvariantCEntailed* (*getConflictFlag stateB*) (*getF stateB*) (*getC stateB*)
using ‹*getF stateB* = *getF stateA*›
‹*getConflictFlag stateB* = *getConflictFlag stateA*›
‹*getC stateB* = *getC stateA*›
‹*InvariantCEntailed* (*getConflictFlag stateA*) (*getF stateA*) (*getC stateA*)›
by *simp*

ultimately
have *?thesis*
unfolding *invariantsHoldInState-def*
by *auto*
}
moreover
{
assume *appliedUnitPropagate stateA stateB F0 decisionVars*
then obtain *uc*::*Clause* **and** *ul*::*Literal* **where**
formulaEntailsClause (*getF stateA*) *uc*
(*var ul*) ∈ *decisionVars* ∪ *vars F0*
isUnitClause uc ul (*elements* (*getM stateA*))
getF stateB = *getF stateA*
getM stateB = *getM stateA* @ [(*ul*, *False*)]
getConflictFlag stateB = *getConflictFlag stateA*
getC stateB = *getC stateA*
unfolding *appliedUnitPropagate-def*
by *auto*

from ‹*isUnitClause uc ul* (*elements* (*getM stateA*))›
have *ul el uc*
unfolding *isUnitClause-def*
by *simp*

from ‹*var ul* ∈ *decisionVars* ∪ *vars F0*›
have *InvariantVarsM* (*getM stateB*) *F0 decisionVars*
using ‹*getF stateB* = *getF stateA*›
‹*InvariantVarsM* (*getM stateA*) *F0 decisionVars*›
‹*getM stateB* = *getM stateA* @ [(*ul*, *False*)]›
InvariantVarsMAfterUnitPropagate[*of getM stateA F0 decisionVars ul getM stateB*]
by *auto*
moreover
have *InvariantVarsF* (*getF stateB*) *F0 decisionVars*
using ‹*getF stateB* = *getF stateA*›
‹*InvariantVarsF* (*getF stateA*) *F0 decisionVars*›
by *simp*
moreover
have *InvariantConsistent* (*getM stateB*)
using ‹*InvariantConsistent* (*getM stateA*)›
‹*isUnitClause uc ul* (*elements* (*getM stateA*))›
‹*getM stateB* = *getM stateA* @ [(*ul*, *False*)]›
InvariantConsistentAfterUnitPropagate [*of getM stateA uc ul getM stateB*]
by *simp*
moreover
have *InvariantUniq* (*getM stateB*)
using ‹*InvariantUniq* (*getM stateA*)›
‹*isUnitClause uc ul* (*elements* (*getM stateA*))›

‹getM stateB = getM stateA @ [(ul, False)]›
InvariantUniqAfterUnitPropagate [*of getM stateA uc ul getM stateB*]
by *simp*
moreover
have *InvariantReasonClauses* (*getF stateB*) (*getM stateB*)
using ‹*getF stateB* = *getF stateA*›
‹*InvariantReasonClauses* (*getF stateA*) (*getM stateA*)›
‹*isUnitClause uc ul* (*elements* (*getM stateA*))›
‹*getM stateB* = *getM stateA* @ [(*ul*, *False*)]›
‹*formulaEntailsClause* (*getF stateA*) *uc*›
InvariantReasonClausesAfterUnitPropagate[*of getF stateA getM stateA uc ul getM stateB*]
by *simp*
moreover
have *InvariantEquivalent F0* (*getF stateB*)
using ‹*getF stateB* = *getF stateA*›
‹*InvariantEquivalent F0* (*getF stateA*)›
by *simp*
moreover
have *InvariantCFalse* (*getConflictFlag stateB*) (*getM stateB*) (*getC stateB*)
using ‹*getM stateB* = *getM stateA* @ [(*ul*, *False*)]›
‹*getConflictFlag stateB* = *getConflictFlag stateA*›
‹*getC stateB* = *getC stateA*›
‹*InvariantCFalse* (*getConflictFlag stateA*) (*getM stateA*) (*getC stateA*)›
InvariantCFalseAfterUnitPropagate[*of getConflictFlag stateA getM stateA getC stateA getM stateB ul*]
by *simp*
moreover
have *InvariantCEntailed* (*getConflictFlag stateB*) (*getF stateB*) (*getC stateB*)
using ‹*getF stateB* = *getF stateA*›
‹*getConflictFlag stateB* = *getConflictFlag stateA*›
‹*getC stateB* = *getC stateA*›
‹*InvariantCEntailed* (*getConflictFlag stateA*) (*getF stateA*) (*getC stateA*)›
by *simp*
ultimately
have *?thesis*
unfolding *invariantsHoldInState-def*
by *auto*
}
moreover
{
assume *appliedConflict stateA stateB*
then obtain *clause*::*Clause* **where**
getConflictFlag stateA = *False*

```
    formulaEntailsClause (getF stateA) clause
    clauseFalse clause (elements (getM stateA))
    getF stateB = getF stateA
    getM stateB = getM stateA
    getConflictFlag stateB = True
    getC stateB = clause
  unfolding appliedConflict-def
  by auto

  have InvariantVarsM (getM stateB) F0 decisionVars
    using ‹InvariantVarsM (getM stateA) F0 decisionVars›
      ‹getM stateB = getM stateA›
    by simp
  moreover
  have InvariantVarsF (getF stateB) F0 decisionVars
    using ‹InvariantVarsF (getF stateA) F0 decisionVars›
      ‹getF stateB = getF stateA›
    by simp
  moreover
  have InvariantConsistent (getM stateB)
    using ‹InvariantConsistent (getM stateA)›
      ‹getM stateB = getM stateA›
    by simp
  moreover
  have InvariantUniq (getM stateB)
    using ‹InvariantUniq (getM stateA)›
      ‹getM stateB = getM stateA›
    by simp
  moreover
  have InvariantReasonClauses (getF stateB) (getM stateB)
    using ‹InvariantReasonClauses (getF stateA) (getM stateA)›
      ‹getF stateB = getF stateA›
      ‹getM stateB = getM stateA›
    by simp
  moreover
  have InvariantEquivalent F0 (getF stateB)
    using ‹InvariantEquivalent F0 (getF stateA)›
      ‹getF stateB = getF stateA›
    by simp
  moreover
 have InvariantCFalse (getConflictFlag stateB) (getM stateB) (getC
stateB)
    using
    ‹clauseFalse clause (elements (getM stateA))›
    ‹getM stateB = getM stateA›
    ‹getConflictFlag stateB = True›
    ‹getC stateB = clause›
    unfolding InvariantCFalse-def
    by simp
```

```
    moreover
     have InvariantCEntailed (getConflictFlag stateB) (getF stateB)
(getC stateB)
      unfolding InvariantCEntailed-def
      using
      ‹getConflictFlag stateB = True›
      ‹formulaEntailsClause (getF stateA) clause›
      ‹getF stateB = getF stateA›
      ‹getC stateB = clause›
      by simp
    ultimately
    have ?thesis
      unfolding invariantsHoldInState-def
      by auto
  }
  moreover
  {
    assume appliedExplain stateA stateB
    then obtain l::Literal and reason::Clause where
        getConflictFlag stateA = True
        l el getC stateA
        formulaEntailsClause (getF stateA) reason
        isReason reason (opposite l) (elements (getM stateA))
        getF stateB = getF stateA
        getM stateB = getM stateA
        getConflictFlag stateB = True
        getC stateB = resolve (getC stateA) reason l
      unfolding appliedExplain-def
      by auto

    have InvariantVarsM (getM stateB) F0 decisionVars
      using ‹InvariantVarsM (getM stateA) F0 decisionVars›
        ‹getM stateB = getM stateA›
      by simp
    moreover
    have InvariantVarsF (getF stateB) F0 decisionVars
      using ‹InvariantVarsF (getF stateA) F0 decisionVars›
        ‹getF stateB = getF stateA›
      by simp
    moreover
    have InvariantConsistent (getM stateB)
      using
        ‹getM stateB = getM stateA›
        ‹InvariantConsistent (getM stateA)›
      by simp
    moreover
    have InvariantUniq (getM stateB)
      using
        ‹getM stateB = getM stateA›
```

‹*InvariantUniq* (*getM stateA*)›
by *simp*
moreover
have *InvariantReasonClauses* (*getF stateB*) (*getM stateB*)
using
‹*getF stateB* = *getF stateA*›
‹*getM stateB* = *getM stateA*›
‹*InvariantReasonClauses* (*getF stateA*) (*getM stateA*)›
by *simp*
moreover
have *InvariantEquivalent F0* (*getF stateB*)
using
‹*getF stateB* = *getF stateA*›
‹*InvariantEquivalent F0* (*getF stateA*)›
by *simp*
moreover
have *InvariantCFalse* (*getConflictFlag stateB*) (*getM stateB*) (*getC stateB*)
using
‹*InvariantCFalse* (*getConflictFlag stateA*) (*getM stateA*) (*getC stateA*)›
‹*l el getC stateA*›
‹*isReason reason* (*opposite l*) (*elements* (*getM stateA*))›
‹*getM stateB* = *getM stateA*›
‹*getC stateB* = *resolve* (*getC stateA*) *reason l*›
‹*getConflictFlag stateA* = *True*›
‹*getConflictFlag stateB* = *True*›
InvariantCFalseAfterExplain[*of getConflictFlag stateA getM stateA getC stateA opposite l reason getC stateB*]
by *simp*
moreover
have *InvariantCEntailed* (*getConflictFlag stateB*) (*getF stateB*) (*getC stateB*)
using
‹*InvariantCEntailed* (*getConflictFlag stateA*) (*getF stateA*) (*getC stateA*)›
‹*l el getC stateA*›
‹*isReason reason* (*opposite l*) (*elements* (*getM stateA*))›
‹*getF stateB* = *getF stateA*›
‹*getC stateB* = *resolve* (*getC stateA*) *reason l*›
‹*getConflictFlag stateA* = *True*›
‹*getConflictFlag stateB* = *True*›
‹*formulaEntailsClause* (*getF stateA*) *reason*›
InvariantCEntailedAfterExplain[*of getConflictFlag stateA getF stateA getC stateA reason getC stateB opposite l*]
by *simp*
moreover
ultimately
have *?thesis*

unfolding *invariantsHoldInState-def*
by *auto*
}
moreover
{
assume *appliedLearn stateA stateB*
hence
getConflictFlag stateA = True
¬ *getC stateA el getF stateA*
getF stateB = getF stateA @ [getC stateA]
getM stateB = getM stateA
getConflictFlag stateB = True
getC stateB = getC stateA
unfolding *appliedLearn-def*
by *auto*

from ‹*getConflictFlag stateA = True*› ‹*InvariantCEntailed* (*getConflictFlag stateA*) (*getF stateA*) (*getC stateA*)›
have *formulaEntailsClause* (*getF stateA*) (*getC stateA*)
unfolding *InvariantCEntailed-def*
by *simp*

have *InvariantVarsM* (*getM stateB*) *F0 decisionVars*
using ‹*InvariantVarsM* (*getM stateA*) *F0 decisionVars*›
‹*getM stateB = getM stateA*›
by *simp*
moreover
from ‹*InvariantCFalse* (*getConflictFlag stateA*) (*getM stateA*) (*getC stateA*)› ‹*getConflictFlag stateA = True*›
have *clauseFalse* (*getC stateA*) (*elements* (*getM stateA*))
unfolding *InvariantCFalse-def*
by *simp*
with ‹*InvariantVarsM* (*getM stateA*) *F0 decisionVars*›
have (*vars* (*getC stateA*)) ⊆ *vars F0* ∪ *decisionVars*
unfolding *InvariantVarsM-def*
using *valuationContainsItsFalseClausesVariables*[*of getC stateA elements* (*getM stateA*)]
by *simp*
hence *InvariantVarsF* (*getF stateB*) *F0 decisionVars*
using ‹*getF stateB = getF stateA @* [*getC stateA*]›
‹*InvariantVarsF* (*getF stateA*) *F0 decisionVars*›
InvariantVarsFAfterLearn [*of getF stateA F0 decisionVars getC stateA getF stateB*]
by *simp*
moreover
have *InvariantConsistent* (*getM stateB*)
using ‹*InvariantConsistent* (*getM stateA*)›
‹*getM stateB = getM stateA*›
by *simp*

moreover
have *InvariantUniq* (*getM stateB*)
using ‹*InvariantUniq* (*getM stateA*)›
‹*getM stateB* = *getM stateA*›
by *simp*
moreover
have *InvariantReasonClauses* (*getF stateB*) (*getM stateB*)
using
‹*InvariantReasonClauses* (*getF stateA*) (*getM stateA*)›
‹*formulaEntailsClause* (*getF stateA*) (*getC stateA*)›
‹*getF stateB* = *getF stateA* @ [*getC stateA*]›
‹*getM stateB* = *getM stateA*›
InvariantReasonClausesAfterLearn[*of getF stateA getM stateA getC stateA getF stateB*]
by *simp*
moreover
have *InvariantEquivalent F0* (*getF stateB*)
using
‹*InvariantEquivalent F0* (*getF stateA*)›
‹*formulaEntailsClause* (*getF stateA*) (*getC stateA*)›
‹*getF stateB* = *getF stateA* @ [*getC stateA*]›
InvariantEquivalentAfterLearn[*of F0 getF stateA getC stateA getF stateB*]
by *simp*
moreover
have *InvariantCFalse* (*getConflictFlag stateB*) (*getM stateB*) (*getC stateB*)
using ‹*InvariantCFalse* (*getConflictFlag stateA*) (*getM stateA*) (*getC stateA*)›
‹*getM stateB* = *getM stateA*›
‹*getConflictFlag stateA* = *True*›
‹*getConflictFlag stateB* = *True*›
‹*getM stateB* = *getM stateA*›
‹*getC stateB* = *getC stateA*›
by *simp*
moreover
have *InvariantCEntailed* (*getConflictFlag stateB*) (*getF stateB*) (*getC stateB*)
using
‹*InvariantCEntailed* (*getConflictFlag stateA*) (*getF stateA*) (*getC stateA*)›
‹*formulaEntailsClause* (*getF stateA*) (*getC stateA*)›
‹*getF stateB* = *getF stateA* @ [*getC stateA*]›
‹*getConflictFlag stateA* = *True*›
‹*getConflictFlag stateB* = *True*›
‹*getC stateB* = *getC stateA*›
InvariantCEntailedAfterLearn[*of getConflictFlag stateA getF stateA getC stateA getF stateB*]
by *simp*

ultimately
have *?thesis*
unfolding *invariantsHoldInState-def*
by *auto*
}
moreover
{
assume *appliedBackjump stateA stateB*
then obtain *l*::*Literal* **and** *level*::*nat*
where
getConflictFlag stateA = *True*
isBackjumpLevel level l (*getC stateA*) (*getM stateA*)
getF stateB = *getF stateA*
getM stateB = *prefixToLevel level* (*getM stateA*) @ [(*l*, *False*)]
getConflictFlag stateB = *False*
getC stateB = []
unfolding *appliedBackjump-def*
by *auto*
with ‹*InvariantConsistent* (*getM stateA*)› ‹*InvariantUniq* (*getM stateA*)›
‹*InvariantCFalse* (*getConflictFlag stateA*) (*getM stateA*) (*getC stateA*)›
have *isUnitClause* (*getC stateA*) *l* (*elements* (*prefixToLevel level* (*getM stateA*)))
unfolding *InvariantUniq-def*
unfolding *InvariantConsistent-def*
unfolding *InvariantCFalse-def*
using *isBackjumpLevelEnsuresIsUnitInPrefix*[*of getM stateA getC stateA level l*]
by *simp*

from ‹*getConflictFlag stateA* = *True*› ‹*InvariantCEntailed* (*getConflictFlag stateA*) (*getF stateA*) (*getC stateA*)›
have *formulaEntailsClause* (*getF stateA*) (*getC stateA*)
unfolding *InvariantCEntailed-def*
by *simp*

from ‹*isBackjumpLevel level l* (*getC stateA*) (*getM stateA*)›
have *isLastAssertedLiteral* (*opposite l*) (*oppositeLiteralList* (*getC stateA*)) (*elements* (*getM stateA*))
unfolding *isBackjumpLevel-def*
by *simp*
hence *l el getC stateA*
unfolding *isLastAssertedLiteral-def*
using *literalElListIffOppositeLiteralElOppositeLiteralList*[*of l getC stateA*]
by *simp*

have *isPrefix* (*prefixToLevel level* (*getM stateA*)) (*getM stateA*)

by (*simp add*:*isPrefixPrefixToLevel*)

from ‹*getConflictFlag stateA* = *True*› ‹*InvariantCEntailed* (*getConflictFlag stateA*) (*getF stateA*) (*getC stateA*)›
have *formulaEntailsClause* (*getF stateA*) (*getC stateA*)
unfolding *InvariantCEntailed-def*
by *simp*

from ‹*getConflictFlag stateA* = *True*› ‹*InvariantCFalse* (*getConflictFlag stateA*) (*getM stateA*) (*getC stateA*)›
have *clauseFalse* (*getC stateA*) (*elements* (*getM stateA*))
unfolding *InvariantCFalse-def*
by *simp*
hence *vars* (*getC stateA*) ⊆ *vars* (*elements* (*getM stateA*))
using *valuationContainsItsFalseClausesVariables*[*of getC stateA elements* (*getM stateA*)]
by *simp*
moreover
from ‹*l el getC stateA*›
have *var l* ∈ *vars* (*getC stateA*)
using *clauseContainsItsLiteralsVariable*[*of l getC stateA*]
by *simp*
ultimately
have *var l* ∈ *vars F0* ∪ *decisionVars*
using ‹*InvariantVarsM* (*getM stateA*) *F0 decisionVars*›
unfolding *InvariantVarsM-def*
by *auto*

have *InvariantVarsM* (*getM stateB*) *F0 decisionVars*
using ‹*InvariantVarsM* (*getM stateA*) *F0 decisionVars*›
‹*isUnitClause* (*getC stateA*) *l* (*elements* (*prefixToLevel level* (*getM stateA*)))›
‹*isPrefix* (*prefixToLevel level* (*getM stateA*)) (*getM stateA*)›
‹*var l* ∈ *vars F0* ∪ *decisionVars*›
‹*formulaEntailsClause* (*getF stateA*) (*getC stateA*)›
‹*getF stateB* = *getF stateA*›
‹*getM stateB* = *prefixToLevel level* (*getM stateA*) @ [(*l*, *False*)]›
InvariantVarsMAfterBackjump[*of getM stateA F0 decisionVars prefixToLevel level* (*getM stateA*) *l getM stateB*]
by *simp*
moreover
have *InvariantVarsF* (*getF stateB*) *F0 decisionVars*
using ‹*InvariantVarsF* (*getF stateA*) *F0 decisionVars*›
‹*getF stateB* = *getF stateA*›
by *simp*
moreover
have *InvariantConsistent* (*getM stateB*)
using ‹*InvariantConsistent* (*getM stateA*)›
‹*isUnitClause* (*getC stateA*) *l* (*elements* (*prefixToLevel level*

(*getM stateA*)))›
‹*isPrefix* (*prefixToLevel level* (*getM stateA*)) (*getM stateA*)›
‹*getM stateB* = *prefixToLevel level* (*getM stateA*) @ [(*l*, *False*)]›
InvariantConsistentAfterBackjump[*of getM stateA prefixToLevel level* (*getM stateA*) *getC stateA l getM stateB*]
by *simp*
moreover
have *InvariantUniq* (*getM stateB*)
using ‹*InvariantUniq* (*getM stateA*)›
‹*isUnitClause* (*getC stateA*) *l* (*elements* (*prefixToLevel level* (*getM stateA*)))›
‹*isPrefix* (*prefixToLevel level* (*getM stateA*)) (*getM stateA*)›
‹*getM stateB* = *prefixToLevel level* (*getM stateA*) @ [(*l*, *False*)]›
InvariantUniqAfterBackjump[*of getM stateA prefixToLevel level* (*getM stateA*) *getC stateA l getM stateB*]
by *simp*
moreover
have *InvariantReasonClauses* (*getF stateB*) (*getM stateB*)
using ‹*InvariantUniq* (*getM stateA*)› ‹*InvariantReasonClauses* (*getF stateA*) (*getM stateA*)›
‹*isUnitClause* (*getC stateA*) *l* (*elements* (*prefixToLevel level* (*getM stateA*)))›
‹*isPrefix* (*prefixToLevel level* (*getM stateA*)) (*getM stateA*)›
‹*formulaEntailsClause* (*getF stateA*) (*getC stateA*)›
‹*getF stateB* = *getF stateA*›
‹*getM stateB* = *prefixToLevel level* (*getM stateA*) @ [(*l*, *False*)]›
InvariantReasonClausesAfterBackjump[*of getF stateA getM stateA*
prefixToLevel level (*getM stateA*) *getC stateA l getM stateB*]
by *simp*
moreover
have *InvariantEquivalent F0* (*getF stateB*)
using
‹*InvariantEquivalent F0* (*getF stateA*)›
‹*getF stateB* = *getF stateA*›
by *simp*
moreover
have *InvariantCFalse* (*getConflictFlag stateB*) (*getM stateB*) (*getC stateB*)
using ‹*getConflictFlag stateB* = *False*›
unfolding *InvariantCFalse-def*
by *simp*
moreover
have *InvariantCEntailed* (*getConflictFlag stateB*) (*getF stateB*) (*getC stateB*)
using ‹*getConflictFlag stateB* = *False*›
unfolding *InvariantCEntailed-def*
by *simp*
moreover

ultimately
have *?thesis*
unfolding *invariantsHoldInState-def*
by *auto*
}
ultimately
show *?thesis*
using ‹*transition stateA stateB F0 decisionVars*›
unfolding *transition-def*
by *auto*
qed

The consequence is that invariants hold in all valid runs.

lemma *invariantsHoldInValidRuns*:
fixes *F0* :: *Formula* **and** *decisionVars* :: *Variable set*
assumes *invariantsHoldInState stateA F0 decisionVars* **and**
(*stateA*, *stateB*) ∈ *transitionRelation F0 decisionVars*
shows *invariantsHoldInState stateB F0 decisionVars*
using *assms*
using *transitionsPreserveInvariants*
using *rtrancl-induct*[*of stateA stateB*
{(*stateA*, *stateB*). *transition stateA stateB F0 decisionVars*} λ *x*. *invariantsHoldInState x F0 decisionVars*]
unfolding *transitionRelation-def*
by *auto*

lemma *invariantsHoldInValidRunsFromInitialState*:
fixes *F0* :: *Formula* **and** *decisionVars* :: *Variable set*
assumes *isInitialState state0 F0*
and (*state0*, *state*) ∈ *transitionRelation F0 decisionVars*
shows *invariantsHoldInState state F0 decisionVars*
proof−
from ‹*isInitialState state0 F0*›
have *invariantsHoldInState state0 F0 decisionVars*
by (*simp add*:*invariantsHoldInInitialState*)
with *assms*
show *?thesis*
using *invariantsHoldInValidRuns* [*of state0 F0 decisionVars state*]
by *simp*
qed

In the following text we will show that there are two kinds of states:

1. *UNSAT* states where *getConflictFlag state* = *True* and *getC state* = [].
2. *SAT* states where *getConflictFlag state* = *False*, ¬ *formulaFalse F0* (*elements* (*getM state*)) and *decisionVars* ⊆ *vars* (*elements* (*getM state*)).

The soundness theorems claim that if *UNSAT* state is reached the formula is unsatisfiable and if *SAT* state is reached, the formula is satisfiable.

Completeness theorems claim that every final state is either *UNSAT* or *SAT*. A consequence of this and soundness theorems, is that if formula is unsatisfiable the solver will finish in an *UNSAT* state, and if the formula is satisfiable the solver will finish in a *SAT* state.

7.3 Soundness

theorem *soundnessForUNSAT*:
fixes *F0* :: *Formula* **and** *decisionVars* :: *Variable set* **and** *state0* :: *State* **and** *state* :: *State*
assumes
isInitialState state0 F0 **and**
(*state0*, *state*) ∈ *transitionRelation F0 decisionVars*

getConflictFlag state = *True* **and**
getC state = []
shows ¬ *satisfiable F0*
proof−
from ‹*isInitialState state0 F0*› ‹(*state0*, *state*) ∈ *transitionRelation F0 decisionVars*›
have *invariantsHoldInState state F0 decisionVars*
using *invariantsHoldInValidRunsFromInitialState*
by *simp*
hence
InvariantEquivalent F0 (*getF state*)
InvariantCEntailed (*getConflictFlag state*) (*getF state*) (*getC state*)
unfolding *invariantsHoldInState-def*
by *auto*
with ‹*getConflictFlag state* = *True*› ‹*getC state* = []›
show *?thesis*
by (*simp add*:*unsatReportExtensiveExplain*)
qed

theorem *soundnessForSAT*:
fixes *F0* :: *Formula* **and** *decisionVars* :: *Variable set* **and** *state0* :: *State* **and** *state* :: *State*
assumes
vars F0 ⊆ *decisionVars* **and**

isInitialState state0 F0 **and**
(*state0*, *state*) ∈ *transitionRelation F0 decisionVars* **and**

getConflictFlag state = *False*
¬ *formulaFalse* (*getF state*) (*elements* (*getM state*))

vars (*elements* (*getM state*)) ⊇ *decisionVars*
shows
model (*elements* (*getM state*)) *F0*
proof−
from ‹*isInitialState state0 F0*› ‹(*state0*, *state*) ∈ *transitionRelation F0 decisionVars*›
have *invariantsHoldInState state F0 decisionVars*
using *invariantsHoldInValidRunsFromInitialState*
by *simp*
hence
InvariantConsistent (*getM state*)
InvariantEquivalent F0 (*getF state*)
InvariantVarsF (*getF state*) *F0 decisionVars*
unfolding *invariantsHoldInState-def*
by *auto*
with *assms*
show *?thesis*
using *satReport*[*of F0 decisionVars getF state getM state*]
by *simp*
qed

7.4 Termination

We now define a termination ordering which is a lexicographic combination of *lexLessRestricted* trail ordering, *boolLess* conflict flag ordering, *multLess* conflict clause ordering and *learnLess* formula ordering. This ordering will be central in termination proof.

definition *lexLessState* (*F0*::*Formula*) *decisionVars* == {((*stateA*::*State*), (*stateB*::*State*)).
(*getM stateA*, *getM stateB*) ∈ *lexLessRestricted* (*vars F0* ∪ *decisionVars*)}
definition *boolLessState* == {((*stateA*::*State*), (*stateB*::*State*)).
getM stateA = *getM stateB* ∧
(*getConflictFlag stateA*, *getConflictFlag stateB*) ∈ *boolLess*}
definition *multLessState* == {((*stateA*::*State*), (*stateB*::*State*)).
getM stateA = *getM stateB* ∧
getConflictFlag stateA = *getConflictFlag stateB* ∧
(*getC stateA*, *getC stateB*) ∈ *multLess* (*getM stateA*)}
definition *learnLessState* == {((*stateA*::*State*), (*stateB*::*State*)).
getM stateA = *getM stateB* ∧
getConflictFlag stateA = *getConflictFlag stateB* ∧
getC stateA = *getC stateB* ∧
(*getF stateA*, *getF stateB*) ∈ *learnLess* (*getC stateA*)}

definition *terminationLess F0 decisionVars* == {((*stateA*::*State*), (*stateB*::*State*)).
(*stateA*,*stateB*) ∈ *lexLessState F0 decisionVars* ∨
(*stateA*,*stateB*) ∈ *boolLessState* ∨
(*stateA*,*stateB*) ∈ *multLessState* ∨

(*stateA*,*stateB*) ∈ *learnLessState*}

We want to show that every valid transition decreases a state with respect to the constructed termination ordering.

First we show that *Decide*, *UnitPropagate* and *Backjump* rule decrease the trail with respect to the restricted trail ordering *lexLessRestricted*. Invariants ensure that trails are indeed uniq, consistent and with finite variable sets.

lemma *trailIsDecreasedByDeciedUnitPropagateAndBackjump*:
fixes *stateA*::*State* **and** *stateB*::*State*
assumes *invariantsHoldInState stateA F0 decisionVars* **and**
appliedDecide stateA stateB decisionVars ∨ *appliedUnitPropagate stateA stateB F0 decisionVars* ∨ *appliedBackjump stateA stateB*
shows (*getM stateB*, *getM stateA*) ∈ *lexLessRestricted* (*vars F0* ∪ *decisionVars*)
proof−
from ‹*appliedDecide stateA stateB decisionVars* ∨ *appliedUnitPropagate stateA stateB F0 decisionVars* ∨ *appliedBackjump stateA stateB*›
‹*invariantsHoldInState stateA F0 decisionVars*›
have *invariantsHoldInState stateB F0 decisionVars*
using *transitionsPreserveInvariants*
unfolding *transition-def*
by *auto*
from ‹*invariantsHoldInState stateA F0 decisionVars*›
have ∗: *uniq* (*elements* (*getM stateA*)) *consistent* (*elements* (*getM stateA*)) *vars* (*elements* (*getM stateA*)) ⊆ *vars F0* ∪ *decisionVars*
unfolding *invariantsHoldInState-def*
unfolding *InvariantVarsM-def*
unfolding *InvariantConsistent-def*
unfolding *InvariantUniq-def*
by *auto*
from ‹*invariantsHoldInState stateB F0 decisionVars*›
have ∗∗: *uniq* (*elements* (*getM stateB*)) *consistent* (*elements* (*getM stateB*)) *vars* (*elements* (*getM stateB*)) ⊆ *vars F0* ∪ *decisionVars*
unfolding *invariantsHoldInState-def*
unfolding *InvariantVarsM-def*
unfolding *InvariantConsistent-def*
unfolding *InvariantUniq-def*
by *auto*
{
assume *appliedDecide stateA stateB decisionVars*
hence (*getM stateB*, *getM stateA*) ∈ *lexLess*
unfolding *appliedDecide-def*
by (*auto simp add*:*lexLessAppend*)
with ∗ ∗∗
have ((*getM stateB*), (*getM stateA*)) ∈ *lexLessRestricted* (*vars F0* ∪ *decisionVars*)
unfolding *lexLessRestricted-def*

by *auto*
}
moreover
{
assume *appliedUnitPropagate stateA stateB F0 decisionVars*
hence (*getM stateB*, *getM stateA*) ∈ *lexLess*
unfolding *appliedUnitPropagate-def*
by (*auto simp add*:*lexLessAppend*)
with ∗ ∗∗
have (*getM stateB*, *getM stateA*) ∈ *lexLessRestricted* (*vars F0* ∪ *decisionVars*)
unfolding *lexLessRestricted-def*
by *auto*
}
moreover
{
assume *appliedBackjump stateA stateB*
then obtain *l*::*Literal* **and** *level*::*nat*
where
getConflictFlag stateA = *True*
isBackjumpLevel level l (*getC stateA*) (*getM stateA*)
getF stateB = *getF stateA*
getM stateB = *prefixToLevel level* (*getM stateA*) @ [(*l*, *False*)]
getConflictFlag stateB = *False*
getC stateB = []
unfolding *appliedBackjump-def*
by *auto*

from ‹*isBackjumpLevel level l* (*getC stateA*) (*getM stateA*)›
have *isLastAssertedLiteral* (*opposite l*) (*oppositeLiteralList* (*getC stateA*)) (*elements* (*getM stateA*))
unfolding *isBackjumpLevel-def*
by *simp*
hence (*opposite l*) *el elements* (*getM stateA*)
unfolding *isLastAssertedLiteral-def*
by *simp*
hence *elementLevel* (*opposite l*) (*getM stateA*) <= *currentLevel* (*getM stateA*)
by (*simp add*: *elementLevelLeqCurrentLevel*)
moreover
from ‹*isBackjumpLevel level l* (*getC stateA*) (*getM stateA*)›
have *0* ≤ *level* **and** *level* < *elementLevel* (*opposite l*) (*getM stateA*)

unfolding *isBackjumpLevel-def*
using ‹*isLastAssertedLiteral* (*opposite l*) (*oppositeLiteralList* (*getC stateA*)) (*elements* (*getM stateA*))›
by *auto*
ultimately
have *level* < *currentLevel* (*getM stateA*)

by *simp*
with ‹*0* ≤ *level*› ‹*getM stateB* = *prefixToLevel level* (*getM stateA*) @ [(*l*, *False*)]›
have (*getM stateB*, *getM stateA*) ∈ *lexLess*
by (*simp add*:*lexLessBackjump*)
with ∗ ∗∗
have (*getM stateB*, *getM stateA*) ∈ *lexLessRestricted* (*vars F0* ∪ *decisionVars*)
unfolding *lexLessRestricted-def*
by *auto*
}
ultimately
show *?thesis*
using *assms*
by *auto*
qed

Next we show that $Conflict$ decreases the conflict flag in the *boolLess* ordering.

lemma *conflictFlagIsDecreasedByConflict*:
fixes *stateA*::*State* **and** *stateB*::*State*
assumes *appliedConflict stateA stateB*
shows *getM stateA* = *getM stateB* **and** (*getConflictFlag stateB*, *getConflictFlag stateA*) ∈ *boolLess*
using *assms*
unfolding *appliedConflict-def*
unfolding *boolLess-def*
by *auto*

Next we show that $Explain$ decreases the conflict clause with respect to the *multLess* clause ordering.

lemma *conflictClauseIsDecreasedByExplain*:
fixes *stateA*::*State* **and** *stateB*::*State*
assumes *appliedExplain stateA stateB*
shows
getM stateA = *getM stateB* **and**
getConflictFlag stateA = *getConflictFlag stateB* **and**
(*getC stateB*, *getC stateA*) ∈ *multLess* (*getM stateA*)
proof−
from ‹*appliedExplain stateA stateB*›
obtain *l*::*Literal* **and** *reason*::*Clause* **where**
getConflictFlag stateA = *True*
l el (*getC stateA*)
isReason reason (*opposite l*) (*elements* (*getM stateA*))
getF stateB = *getF stateA*
getM stateB = *getM stateA*
getConflictFlag stateB = *True*
getC stateB = *resolve* (*getC stateA*) *reason l*
unfolding *appliedExplain-def*

by *auto*
thus *getM stateA* = *getM stateB getConflictFlag stateA* = *getConflictFlag stateB* (*getC stateB*, *getC stateA*) ∈ *multLess* (*getM stateA*)
using *multLessResolve*[*of opposite l getC stateA reason getM stateA*]
by *auto*
qed

Finally, we show that *Learn* decreases the formula in the *learnLess* formula ordering.

lemma *formulaIsDecreasedByLearn*:
fixes *stateA*::*State* **and** *stateB*::*State*
assumes *appliedLearn stateA stateB*
shows
getM stateA = *getM stateB* **and**
getConflictFlag stateA = *getConflictFlag stateB* **and**
getC stateA = *getC stateB* **and**
(*getF stateB*, *getF stateA*) ∈ *learnLess* (*getC stateA*)
proof−
from ‹*appliedLearn stateA stateB*›
have
getConflictFlag stateA = *True*
¬ *getC stateA el getF stateA*
getF stateB = *getF stateA* @ [*getC stateA*]
getM stateB = *getM stateA*
getConflictFlag stateB = *True*
getC stateB = *getC stateA*
unfolding *appliedLearn-def*
by *auto*
thus
getM stateA = *getM stateB*
getConflictFlag stateA = *getConflictFlag stateB*
getC stateA = *getC stateB*
(*getF stateB*, *getF stateA*) ∈ *learnLess* (*getC stateA*)
unfolding *learnLess-def*
by *auto*
qed

Now we can prove that every rule application decreases a state with respect to the constructed termination ordering.

lemma *stateIsDecreasedByValidTransitions*:
fixes *stateA*::*State* **and** *stateB*::*State*
assumes *invariantsHoldInState stateA F0 decisionVars* **and** *transition stateA stateB F0 decisionVars*
shows (*stateB*, *stateA*) ∈ *terminationLess F0 decisionVars*
proof−
{
assume *appliedDecide stateA stateB decisionVars* ∨ *appliedUnitPropagate stateA stateB F0 decisionVars* ∨ *appliedBackjump stateA stateB*

with ‹*invariantsHoldInState stateA F0 decisionVars*›
have (*getM stateB*, *getM stateA*) ∈ *lexLessRestricted* (*vars F0* ∪ *decisionVars*)
using *trailIsDecreasedByDeciedUnitPropagateAndBackjump*
by *simp*
hence (*stateB*, *stateA*) ∈ *lexLessState F0 decisionVars*
unfolding *lexLessState-def*
by *simp*
hence (*stateB*, *stateA*) ∈ *terminationLess F0 decisionVars*
unfolding *terminationLess-def*
by *simp*
}
moreover
{
assume *appliedConflict stateA stateB*
hence *getM stateA* = *getM stateB* (*getConflictFlag stateB*, *getConflictFlag stateA*) ∈ *boolLess*
using *conflictFlagIsDecreasedByConflict*
by *auto*
hence (*stateB*, *stateA*) ∈ *boolLessState*
unfolding *boolLessState-def*
by *simp*
hence (*stateB*, *stateA*) ∈ *terminationLess F0 decisionVars*
unfolding *terminationLess-def*
by *simp*
}
moreover
{
assume *appliedExplain stateA stateB*
hence *getM stateA* = *getM stateB*
getConflictFlag stateA = *getConflictFlag stateB*
(*getC stateB*, *getC stateA*) ∈ *multLess* (*getM stateA*)
using *conflictClauseIsDecreasedByExplain*
by *auto*
hence (*stateB*, *stateA*) ∈ *multLessState*
unfolding *multLessState-def*
unfolding *multLess-def*
by *simp*
hence (*stateB*, *stateA*) ∈ *terminationLess F0 decisionVars*
unfolding *terminationLess-def*
by *simp*
}
moreover
{
assume *appliedLearn stateA stateB*
hence
getM stateA = *getM stateB*
getConflictFlag stateA = *getConflictFlag stateB*
getC stateA = *getC stateB*

```
      (getF stateB, getF stateA) ∈ learnLess (getC stateA)
      using formulaIsDecreasedByLearn
      by auto
    hence (stateB, stateA) ∈ learnLessState
      unfolding learnLessState-def
      by simp
    hence (stateB, stateA) ∈ terminationLess F0 decisionVars
      unfolding terminationLess-def
      by simp
  }
  ultimately
  show ?thesis
    using ‹transition stateA stateB F0 decisionVars›
    unfolding transition-def
    by auto
qed
```

The minimal states with respect to the termination ordering are final i.e., no further transition rules are applicable.

```
definition
isMinimalState stateMin F0 decisionVars == (∀ state::State. (state,
stateMin) ∉ terminationLess F0 decisionVars)

lemma minimalStatesAreFinal:
  fixes stateA::State
  assumes
   invariantsHoldInState state F0 decisionVars and isMinimalState
state F0 decisionVars
  shows isFinalState state F0 decisionVars
proof−
  {
    assume ¬ ?thesis
    then obtain state'::State
      where transition state state' F0 decisionVars
      unfolding isFinalState-def
      by auto
    with ‹invariantsHoldInState state F0 decisionVars›
    have (state', state) ∈ terminationLess F0 decisionVars
     using stateIsDecreasedByValidTransitions[of state F0 decisionVars
state']
      unfolding transition-def
      by auto
    with ‹isMinimalState state F0 decisionVars›
    have False
      unfolding isMinimalState-def
      by auto
  }
  thus ?thesis
    by auto
```

qed

We now prove that termination ordering is well founded. We start with several auxiliary lemmas, one for each component of the termination ordering.

lemma *wfLexLessState*:
fixes *decisionVars* :: *Variable set* **and** *F0* :: *Formula*
assumes *finite decisionVars*
shows *wf* (*lexLessState F0 decisionVars*)
unfolding *wf-eq-minimal*
proof−
show $\forall$ *Q state*. *state* $\in$ *Q* $\longrightarrow$ ($\exists$ *stateMin*$\in$*Q*. $\forall$ *state'*. (*state'*, *stateMin*) $\in$ *lexLessState F0 decisionVars* $\longrightarrow$ *state'* $\notin$ *Q*)
proof−
{
fix *Q* :: *State set* **and** *state* :: *State*
assume *state* $\in$ *Q*
let *?Q1* = {*M*::*LiteralTrail*. $\exists$ *state*. *state* $\in$ *Q* $\wedge$ (*getM state*) = *M*}
from ‹*state* $\in$ *Q*›
have *getM state* $\in$ *?Q1*
by *auto*
from ‹*finite decisionVars*›
have *finite* (*vars F0* $\cup$ *decisionVars*)
using *finiteVarsFormula*[*of F0*]
by *simp*
hence *wf* (*lexLessRestricted* (*vars F0* $\cup$ *decisionVars*))
using *wfLexLessRestricted*[*of vars F0* $\cup$ *decisionVars*]
by *simp*
with ‹*getM state* $\in$ *?Q1*›
obtain *Mmin* **where** *Mmin* $\in$ *?Q1* $\forall$ *M'*. (*M'*, *Mmin*) $\in$ *lexLessRestricted* (*vars F0* $\cup$ *decisionVars*) $\longrightarrow$ *M'* $\notin$ *?Q1*
unfolding *wf-eq-minimal*
apply (*erule-tac x=?Q1* **in** *allE*)
apply (*erule-tac x=getM state* **in** *allE*)
by *auto*
from ‹*Mmin* $\in$ *?Q1*› **obtain** *stateMin*
where *stateMin* $\in$ *Q* (*getM stateMin*) = *Mmin*
by *auto*
have $\forall$ *state'*. (*state'*, *stateMin*) $\in$ *lexLessState F0 decisionVars* $\longrightarrow$ *state'* $\notin$ *Q*
proof
fix *state'*
show (*state'*, *stateMin*) $\in$ *lexLessState F0 decisionVars* $\longrightarrow$ *state'* $\notin$ *Q*
proof
assume (*state'*, *stateMin*) $\in$ *lexLessState F0 decisionVars*
hence (*getM state'*, *getM stateMin*) $\in$ *lexLessRestricted* (*vars F0* $\cup$ *decisionVars*)

unfolding *lexLessState-def*
by *auto*
from ‹∀ M′. (M′, Mmin) ∈ lexLessRestricted (vars F0 ∪ decisionVars) ⟶ M′ ∉ ?Q1›
‹(getM state′, getM stateMin) ∈ lexLessRestricted (vars F0 ∪ decisionVars)› ‹getM stateMin = Mmin›
have *getM state′ ∉ ?Q1*
by *simp*
with ‹*getM stateMin = Mmin*›
show *state′ ∉ Q*
by *auto*
qed
qed
with ‹*stateMin ∈ Q*›
have ∃ *stateMin ∈ Q. (∀ state′. (state′, stateMin) ∈ lexLessState F0 decisionVars ⟶ state′ ∉ Q)*
by *auto*
}
thus *?thesis*
by *auto*
qed
qed

lemma *wfBoolLessState*:
shows *wf boolLessState*
unfolding *wf-eq-minimal*
proof−
show ∀ *Q state. state ∈ Q ⟶ (∃ stateMin∈Q. ∀ state′. (state′, stateMin) ∈ boolLessState ⟶ state′ ∉ Q)*
proof−
{
fix *Q* :: *State set* **and** *state* :: *State*
assume *state ∈ Q*
let *?M = (getM state)*
let *?Q1 = {b::bool. ∃ state. state ∈ Q ∧ (getM state) = ?M ∧ (getConflictFlag state) = b}*
from ‹*state ∈ Q*›
have *getConflictFlag state ∈ ?Q1*
by *auto*
with *wfBoolLess*
obtain *bMin* **where** *bMin ∈ ?Q1 ∀ b′. (b′, bMin) ∈ boolLess ⟶ b′ ∉ ?Q1*
unfolding *wf-eq-minimal*
apply (*erule-tac x=?Q1* **in** *allE*)
apply (*erule-tac x=getConflictFlag state* **in** *allE*)
by *auto*
from ‹*bMin ∈ ?Q1*› **obtain** *stateMin*
where *stateMin ∈ Q (getM stateMin) = ?M getConflictFlag stateMin = bMin*

by *auto*
have ∀ *state′*. (*state′*, *stateMin*) ∈ *boolLessState* ⟶ *state′* ∉ *Q*
proof
fix *state′*
show (*state′*, *stateMin*) ∈ *boolLessState* ⟶ *state′* ∉ *Q*
proof
assume (*state′*, *stateMin*) ∈ *boolLessState*
with ‹*getM stateMin* = *?M*›
have *getM state′* = *getM stateMin* (*getConflictFlag state′*, *getConflictFlag stateMin*) ∈ *boolLess*
unfolding *boolLessState-def*
by *auto*
from ‹∀ *b′*. (*b′*, *bMin*) ∈ *boolLess* ⟶ *b′* ∉ *?Q1*›
‹(*getConflictFlag state′*, *getConflictFlag stateMin*) ∈ *boolLess*›
‹*getConflictFlag stateMin* = *bMin*›
have *getConflictFlag state′* ∉ *?Q1*
by *simp*
with ‹*getM state′* = *getM stateMin*› ‹*getM stateMin* = *?M*›
show *state′* ∉ *Q*
by *auto*
qed
qed
with ‹*stateMin* ∈ *Q*›
have ∃ *stateMin* ∈ *Q*. (∀ *state′*. (*state′*, *stateMin*) ∈ *boolLessState* ⟶ *state′* ∉ *Q*)
by *auto*
}
thus *?thesis*
by *auto*
qed
qed

lemma *wfMultLessState*:
shows *wf multLessState*
unfolding *wf-eq-minimal*
proof−
show ∀ *Q state*. *state* ∈ *Q* ⟶ (∃ *stateMin* ∈ *Q*. ∀ *state′*. (*state′*, *stateMin*) ∈ *multLessState* ⟶ *state′* ∉ *Q*)
proof−
{
fix *Q* :: *State set* **and** *state* :: *State*
assume *state* ∈ *Q*
let *?M* = (*getM state*)
let *?Q1* = {*C*::*Clause*. ∃ *state*. *state* ∈ *Q* ∧ (*getM state*) = *?M* ∧ (*getC state*) = *C*}
from ‹*state* ∈ *Q*›
have *getC state* ∈ *?Q1*
by *auto*
with *wfMultLess*[*of ?M*]

obtain *Cmin* **where** *Cmin* ∈ *?Q1* ∀ *C′*. (*C′*, *Cmin*) ∈ *multLess* *?M* ⟶ *C′* ∉ *?Q1*
unfolding *wf-eq-minimal*
apply (*erule-tac x=?Q1* **in** *allE*)
apply (*erule-tac x=getC state* **in** *allE*)
by *auto*
from ‹*Cmin* ∈ *?Q1*› **obtain** *stateMin*
where *stateMin* ∈ *Q* (*getM stateMin*) = *?M getC stateMin* = *Cmin*
by *auto*
have ∀ *state′*. (*state′*, *stateMin*) ∈ *multLessState* ⟶ *state′* ∉ *Q*
proof
fix *state′*
show (*state′*, *stateMin*) ∈ *multLessState* ⟶ *state′* ∉ *Q*
proof
assume (*state′*, *stateMin*) ∈ *multLessState*
with ‹*getM stateMin* = *?M*›
have *getM state′* = *getM stateMin* (*getC state′*, *getC stateMin*) ∈ *multLess ?M*
unfolding *multLessState-def*
by *auto*
from ‹∀ *C′*. (*C′*, *Cmin*) ∈ *multLess ?M* ⟶ *C′* ∉ *?Q1*›
‹(*getC state′*, *getC stateMin*) ∈ *multLess ?M*› ‹*getC stateMin* = *Cmin*›
have *getC state′* ∉ *?Q1*
by *simp*
with ‹*getM state′* = *getM stateMin*› ‹*getM stateMin* = *?M*›
show *state′* ∉ *Q*
by *auto*
qed
qed
with ‹*stateMin* ∈ *Q*›
have ∃ *stateMin* ∈ *Q*. (∀ *state′*. (*state′*, *stateMin*) ∈ *multLessState* ⟶ *state′* ∉ *Q*)
by *auto*
}
thus *?thesis*
by *auto*
qed
qed

lemma *wfLearnLessState*:
shows *wf learnLessState*
unfolding *wf-eq-minimal*
proof−
show ∀ *Q state*. *state* ∈ *Q* ⟶ (∃ *stateMin* ∈ *Q*. ∀ *state′*. (*state′*, *stateMin*) ∈ *learnLessState* ⟶ *state′* ∉ *Q*)
proof−
{

fix *Q* :: *State set* **and** *state* :: *State*
assume *state* ∈ *Q*
let *?M* = (*getM state*)
let *?C* = (*getC state*)
let *?conflictFlag* = (*getConflictFlag state*)
let *?Q1* = {*F*::*Formula*. ∃ *state*. *state* ∈ *Q* ∧ (*getM state*) = *?M* ∧ (*getConflictFlag state*) = *?conflictFlag* ∧ (*getC state*) = *?C* ∧ (*getF state*) = *F*}
from ‹*state* ∈ *Q*›
have *getF state* ∈ *?Q1*
by *auto*
with *wfLearnLess*[*of ?C*]
obtain *Fmin* **where** *Fmin* ∈ *?Q1* ∀ *F'*. (*F'*, *Fmin*) ∈ *learnLess ?C* ⟶ *F'* ∉ *?Q1*
unfolding *wf-eq-minimal*
apply (*erule-tac x*=*?Q1* **in** *allE*)
apply (*erule-tac x*=*getF state* **in** *allE*)
by *auto*
from ‹*Fmin* ∈ *?Q1*› **obtain** *stateMin*
where *stateMin* ∈ *Q* (*getM stateMin*) = *?M getC stateMin* = *?C getConflictFlag stateMin* = *?conflictFlag getF stateMin* = *Fmin*
by *auto*
have ∀ *state'*. (*state'*, *stateMin*) ∈ *learnLessState* ⟶ *state'* ∉ *Q*
proof
fix *state'*
show (*state'*, *stateMin*) ∈ *learnLessState* ⟶ *state'* ∉ *Q*
proof
assume (*state'*, *stateMin*) ∈ *learnLessState*
with ‹*getM stateMin* = *?M*› ‹*getC stateMin* = *?C*› ‹*getConflictFlag stateMin* = *?conflictFlag*›
have *getM state'* = *getM stateMin getC state'* = *getC stateMin*

getConflictFlag state' = *getConflictFlag stateMin* (*getF state'*, *getF stateMin*) ∈ *learnLess ?C*
unfolding *learnLessState-def*
by *auto*
from ‹∀ *F'*. (*F'*, *Fmin*) ∈ *learnLess ?C* ⟶ *F'* ∉ *?Q1*›
‹(*getF state'*, *getF stateMin*) ∈ *learnLess ?C*› ‹*getF stateMin* = *Fmin*›
have *getF state'* ∉ *?Q1*
by *simp*
with ‹*getM state'* = *getM stateMin*› ‹*getC state'* = *getC stateMin*› ‹*getConflictFlag state'* = *getConflictFlag stateMin*›
‹*getM stateMin* = *?M*› ‹*getC stateMin* = *?C*› ‹*getConflictFlag stateMin* = *?conflictFlag*› ‹*getF stateMin* = *Fmin*›
show *state'* ∉ *Q*
by *auto*
qed
qed

with ‹*stateMin* ∈ *Q*›
have ∃ *stateMin* ∈ *Q*. (∀ *state'*. (*state'*, *stateMin*) ∈ *learnLessState* ⟶ *state'* ∉ *Q*)
by *auto*
}
thus *?thesis*
by *auto*
qed
qed

Now we can prove the following key lemma which shows that the termination ordering is well founded.

lemma *wfTerminationLess*:
fixes *decisionVars*::*Variable set* **and** *F0*::*Formula*
assumes *finite decisionVars*
shows *wf* (*terminationLess F0 decisionVars*)
unfolding *wf-eq-minimal*
proof−
show ∀ *Q state*. *state* ∈ *Q* ⟶ (∃ *stateMin* ∈ *Q*. ∀ *state'*. (*state'*, *stateMin*) ∈ *terminationLess F0 decisionVars* ⟶ *state'* ∉ *Q*)
proof−
{
fix *Q*::*State set*
fix *state*::*State*
assume *state* ∈ *Q*

from ‹*finite decisionVars*›
have *wf* (*lexLessState F0 decisionVars*)
using *wfLexLessState*[*of decisionVars F0*]
by *simp*

with ‹*state* ∈ *Q*› **obtain** *state0*
where *state0* ∈ *Q* ∀ *state'*. (*state'*, *state0*) ∈ *lexLessState F0 decisionVars* ⟶ *state'* ∉ *Q*
unfolding *wf-eq-minimal*
by *auto*
let *?Q0* = {*state*. *state* ∈ *Q* ∧ (*getM state*) = (*getM state0*)}
from ‹*state0* ∈ *Q*›
have *state0* ∈ *?Q0*
by *simp*
have *wf boolLessState*
using *wfBoolLessState*
.
with ‹*state0* ∈ *Q*› **obtain** *state1*
where *state1* ∈ *?Q0* ∀ *state'*. (*state'*, *state1*) ∈ *boolLessState* ⟶ *state'* ∉ *?Q0*
unfolding *wf-eq-minimal*
apply (*erule-tac x=?Q0* **in** *allE*)
apply (*erule-tac x=state0* **in** *allE*)

by *auto*
let *?Q1* = {*state. state* ∈ *Q* ∧ *getM state* = *getM state0* ∧ *getConflictFlag state* = *getConflictFlag state1*}
from ‹*state1* ∈ *?Q0*›
have *state1* ∈ *?Q1*
by *simp*
have *wf multLessState*
using *wfMultLessState*
.
with ‹*state1* ∈ *?Q1*› **obtain** *state2*
where *state2* ∈ *?Q1* ∀ *state'*. (*state'*, *state2*) ∈ *multLessState* ⟶ *state'* ∉ *?Q1*
unfolding *wf-eq-minimal*
apply (*erule-tac x=?Q1* **in** *allE*)
apply (*erule-tac x=state1* **in** *allE*)
by *auto*
let *?Q2* = {*state. state* ∈ *Q* ∧ *getM state* = *getM state0* ∧ *getConflictFlag state* = *getConflictFlag state1* ∧ *getC state* = *getC state2*}
from ‹*state2* ∈ *?Q1*›
have *state2* ∈ *?Q2*
by *simp*
have *wf learnLessState*
using *wfLearnLessState*
.
with ‹*state2* ∈ *?Q2*› **obtain** *state3*
where *state3* ∈ *?Q2* ∀ *state'*. (*state'*, *state3*) ∈ *learnLessState* ⟶ *state'* ∉ *?Q2*
unfolding *wf-eq-minimal*
apply (*erule-tac x=?Q2* **in** *allE*)
apply (*erule-tac x=state2* **in** *allE*)
by *auto*
from ‹*state3* ∈ *?Q2*›
have *state3* ∈ *Q*
by *simp*
from ‹*state1* ∈ *?Q0*›
have *getM state1* = *getM state0*
by *simp*
from ‹*state2* ∈ *?Q1*›
have *getM state2* = *getM state0 getConflictFlag state2* = *getConflictFlag state1*
by *auto*
from ‹*state3* ∈ *?Q2*›
have *getM state3* = *getM state0 getConflictFlag state3* = *getConflictFlag state1 getC state3* = *getC state2*
by *auto*
let *?stateMin* = *state3*
have ∀ *state'*. (*state'*, *?stateMin*) ∈ *terminationLess F0 decisionVars* ⟶ *state'* ∉ *Q*

proof
fix *state'*
show (*state'*, *?stateMin*) ∈ *terminationLess F0 decisionVars* ⟶ *state'* ∉ *Q*
proof
assume (*state'*, *?stateMin*) ∈ *terminationLess F0 decisionVars*
hence
(*state'*, *?stateMin*) ∈ *lexLessState F0 decisionVars* ∨
(*state'*, *?stateMin*) ∈ *boolLessState* ∨
(*state'*, *?stateMin*) ∈ *multLessState* ∨
(*state'*, *?stateMin*) ∈ *learnLessState*
unfolding *terminationLess-def*
by *auto*
moreover
{
assume (*state'*, *?stateMin*) ∈ *lexLessState F0 decisionVars*
with ‹*getM state3* = *getM state0*›
have (*state'*, *state0*) ∈ *lexLessState F0 decisionVars*
unfolding *lexLessState-def*
by *simp*
with ‹∀ *state'*. (*state'*, *state0*) ∈ *lexLessState F0 decisionVars* ⟶ *state'* ∉ *Q*›
have *state'* ∉ *Q*
by *simp*
}
moreover
{
assume (*state'*, *?stateMin*) ∈ *boolLessState*
from ‹*?stateMin* ∈ *?Q2*›
‹*getM state1* = *getM state0*›
have *getConflictFlag state3* = *getConflictFlag state1 getM state3* = *getM state1*
by *auto*
with ‹(*state'*, *?stateMin*) ∈ *boolLessState*›
have (*state'*, *state1*) ∈ *boolLessState*
unfolding *boolLessState-def*
by *simp*
with ‹∀ *state'*. (*state'*, *state1*) ∈ *boolLessState* ⟶ *state'* ∉ *?Q0*›
have *state'* ∉ *?Q0*
by *simp*
from ‹(*state'*, *state1*) ∈ *boolLessState*› ‹*getM state1* = *getM state0*›
have *getM state'* = *getM state0*
unfolding *boolLessState-def*
by *auto*
with ‹*state'* ∉ *?Q0*›
have *state'* ∉ *Q*
by *simp*

}
moreover
{
assume (*state'*, *?stateMin*) ∈ *multLessState*
from ‹*?stateMin* ∈ *?Q2*›
‹*getM state1* = *getM state0*› ‹*getM state2* = *getM state0*›
‹*getConflictFlag state2* = *getConflictFlag state1*›
have *getC state3* = *getC state2 getConflictFlag state3* = *getConflictFlag state2 getM state3* = *getM state2*
by *auto*
with ‹(*state'*, *?stateMin*) ∈ *multLessState*›
have (*state'*, *state2*) ∈ *multLessState*
unfolding *multLessState-def*
by *auto*
with ‹∀ *state'*. (*state'*, *state2*) ∈ *multLessState* ⟶ *state'* ∉ *?Q1*›
have *state'* ∉ *?Q1*
by *simp*
from ‹(*state'*, *state2*) ∈ *multLessState*› ‹*getM state2* = *getM state0*› ‹*getConflictFlag state2* = *getConflictFlag state1*›
have *getM state'* = *getM state0 getConflictFlag state'* = *getConflictFlag state1*
unfolding *multLessState-def*
by *auto*
with ‹*state'* ∉ *?Q1*›
have *state'* ∉ *Q*
by *simp*
}
moreover
{
assume (*state'*, *?stateMin*) ∈ *learnLessState*
with ‹∀ *state'*. (*state'*, *?stateMin*) ∈ *learnLessState* ⟶ *state'* ∉ *?Q2*›
have *state'* ∉ *?Q2*
by *simp*
from ‹(*state'*, *?stateMin*) ∈ *learnLessState*›
‹*getM state3* = *getM state0*› ‹*getConflictFlag state3* = *getConflictFlag state1*› ‹*getC state3* = *getC state2*›
have *getM state'* = *getM state0 getConflictFlag state'* = *getConflictFlag state1 getC state'* = *getC state2*
unfolding *learnLessState-def*
by *auto*
with ‹*state'* ∉ *?Q2*›
have *state'* ∉ *Q*
by *simp*
}
ultimately
show *state'* ∉ *Q*
by *auto*

```
      qed
    qed
     with ‹?stateMin ∈ Q› have (∃ stateMin ∈ Q. ∀ state'. (state',
stateMin) ∈ terminationLess F0 decisionVars ⟶ state' ∉ Q)
      by auto
  }
  thus ?thesis
    by simp
qed
qed
```

Using the termination ordering we show that the transition relation is well founded on states reachable from initial state.

```
theorem wfTransitionRelation:
  fixes decisionVars :: Variable set and F0 :: Formula
  assumes finite decisionVars and isInitialState state0 F0
  shows wf {(stateB, stateA).
             (state0, stateA) ∈ transitionRelation F0 decisionVars ∧
(transition stateA stateB F0 decisionVars)}

proof−
  let ?rel = {(stateB, stateA).
               (state0, stateA) ∈ transitionRelation F0 decisionVars ∧
(transition stateA stateB F0 decisionVars)}
  let ?rel'= terminationLess F0 decisionVars

  have ∀ x y. (x, y) ∈ ?rel ⟶ (x, y) ∈ ?rel'
  proof−
    {
      fix stateA::State and stateB::State
      assume (stateB, stateA) ∈ ?rel
      hence (stateB, stateA) ∈ ?rel'
        using ‹isInitialState state0 F0›
        using invariantsHoldInValidRunsFromInitialState[of state0 F0
stateA decisionVars]
         using stateIsDecreasedByValidTransitions[of stateA F0 deci-
sionVars stateB]
        by simp
    }
    thus ?thesis
      by simp
  qed
  moreover
  have wf ?rel'
    using ‹finite decisionVars›
    by (rule wfTerminationLess)
  ultimately
  show ?thesis
    using wellFoundedEmbed[of ?rel ?rel']
```

by *simp*
qed

We will now give two corollaries of the previous theorem. First is a weak termination result that shows that there is a terminating run from every intial state to the final one.

corollary
fixes *decisionVars* :: *Variable set* **and** *F0* :: *Formula* **and** *state0* :: *State*
assumes *finite decisionVars* **and** *isInitialState state0 F0*
shows ∃ *state*. (*state0*, *state*) ∈ *transitionRelation F0 decisionVars* ∧ *isFinalState state F0 decisionVars*
proof−
{
assume ¬ *?thesis*
let *?Q* = {*state*. (*state0*, *state*) ∈ *transitionRelation F0 decisionVars*}
let *?rel* = {(*stateB*, *stateA*). (*state0*, *stateA*) ∈ *transitionRelation F0 decisionVars* ∧
transition stateA stateB F0 decisionVars}
have *state0* ∈ *?Q*
unfolding *transitionRelation-def*
by *simp*
hence ∃ *state*. *state* ∈ *?Q*
by *auto*

from *assms*
have *wf ?rel*
using *wfTransitionRelation*[*of decisionVars state0 F0*]
by *auto*
hence ∀ *Q*. (∃ *x*. *x* ∈ *Q*) ⟶ (∃ *stateMin* ∈ *Q*. ∀ *state*. (*state*, *stateMin*) ∈ *?rel* ⟶ *state* ∉ *Q*)
unfolding *wf-eq-minimal*
by *simp*
hence (∃ *x*. *x* ∈ *?Q*) ⟶ (∃ *stateMin* ∈ *?Q*. ∀ *state*. (*state*, *stateMin*) ∈ *?rel* ⟶ *state* ∉ *?Q*)
by *rule*
with ‹∃ *state*. *state* ∈ *?Q*›
have ∃ *stateMin* ∈ *?Q*. ∀ *state*. (*state*, *stateMin*) ∈ *?rel* ⟶ *state* ∉ *?Q*
by *simp*
then obtain *stateMin*
where *stateMin* ∈ *?Q* **and** ∀ *state*. (*state*, *stateMin*) ∈ *?rel* ⟶ *state* ∉ *?Q*
by *auto*

from ‹*stateMin* ∈ *?Q*›
have (*state0*, *stateMin*) ∈ *transitionRelation F0 decisionVars*
by *simp*

with ‹¬ *?thesis*›
have ¬ *isFinalState stateMin F0 decisionVars*
by *simp*
then obtain *state'*::*State*
where *transition stateMin state' F0 decisionVars*
unfolding *isFinalState-def*
by *auto*
have (*state'*, *stateMin*) ∈ *?rel*
using ‹(*state0*, *stateMin*) ∈ *transitionRelation F0 decisionVars*›
‹*transition stateMin state' F0 decisionVars*›
by *simp*
with ‹∀ *state*. (*state*, *stateMin*) ∈ *?rel* ⟶ *state* ∉ *?Q*›
have *state'* ∉ *?Q*
by *force*
moreover
from ‹(*state0*, *stateMin*) ∈ *transitionRelation F0 decisionVars*› ‹*transition stateMin state' F0 decisionVars*›
have *state'* ∈ *?Q*
unfolding *transitionRelation-def*
using *rtrancl-into-rtrancl*[*of state0 stateMin* {(*stateA*, *stateB*). *transition stateA stateB F0 decisionVars*} *state'*]
by *simp*
ultimately
have *False*
by *simp*
}
thus *?thesis*
by *auto*
qed

Now we prove the final strong termination result which states that there cannot be infinite chains of transitions. If there is an infinite transition chain that starts from an initial state, its elements would for a set that would contain initial state and for every element of that set there would be another element of that set that is directly reachable from it. We show that no such set exists.

corollary *noInfiniteTransitionChains*:
fixes *F0*::*Formula* **and** *decisionVars*::*Variable set*
assumes *finite decisionVars*
shows ¬ (∃ *Q*::(*State set*). ∃ *state0* ∈ *Q*. *isInitialState state0 F0* ∧
(∀ *state* ∈ *Q*. (∃ *state'* ∈ *Q*. *transition state state' F0 decisionVars*))
)
proof−
{
assume ¬ *?thesis*

then obtain *Q*::*State set* **and** *state0*::*State*
  where *isInitialState state0 F0 state0* ∈ *Q*
    ∀ *state* ∈ *Q*. (∃ *state'* ∈ *Q*. *transition state state' F0 decisionVars*)
  by *auto*
let *?rel* = {(*stateB*, *stateA*). (*state0*, *stateA*) ∈ *transitionRelation F0 decisionVars* ∧
    transition stateA stateB F0 decisionVars}
from ‹*finite decisionVars*› ‹*isInitialState state0 F0*›
have *wf ?rel*
  using *wfTransitionRelation*
  by *simp*
hence *wfmin*: ∀ *Q x*. *x* ∈ *Q* ⟶
    (∃ *z*∈*Q*. ∀ *y*. (*y*, *z*) ∈ *?rel* ⟶ *y* ∉ *Q*)
  unfolding *wf-eq-minimal*
  by *simp*
let *?Q* = {*state* ∈ *Q*. (*state0*, *state*) ∈ *transitionRelation F0 decisionVars*}
from ‹*state0* ∈ *Q*›
have *state0* ∈ *?Q*
  unfolding *transitionRelation-def*
  by *simp*
with *wfmin*
obtain *stateMin*::*State*
  where *stateMin* ∈ *?Q* **and** ∀ *y*. (*y*, *stateMin*) ∈ *?rel* ⟶ *y* ∉ *?Q*
  apply (*erule-tac x*=*?Q* **in** *allE*)
  by *auto*

from ‹*stateMin* ∈ *?Q*›
have *stateMin* ∈ *Q* (*state0*, *stateMin*) ∈ *transitionRelation F0 decisionVars*
  by *auto*
with ‹∀ *state* ∈ *Q*. (∃ *state'* ∈ *Q*. *transition state state' F0 decisionVars*)›
obtain *state'*::*State*
  where *state'* ∈ *Q transition stateMin state' F0 decisionVars*
  by *auto*

with ‹(*state0*, *stateMin*) ∈ *transitionRelation F0 decisionVars*›
have (*state'*, *stateMin*) ∈ *?rel*
  by *simp*
with ‹∀ *y*. (*y*, *stateMin*) ∈ *?rel* ⟶ *y* ∉ *?Q*›
have *state'* ∉ *?Q*
  by *force*

from ‹*state'* ∈ *Q*› ‹(*state0*, *stateMin*) ∈ *transitionRelation F0 decisionVars*›
  ‹*transition stateMin state' F0 decisionVars*›
have *state'* ∈ *?Q*

unfolding *transitionRelation-def*
using *rtrancl-into-rtrancl*[*of state0 stateMin* {(*stateA*, *stateB*). *transition stateA stateB F0 decisionVars*} *state'*]
by *simp*
with ‹*state'* ∉ *?Q*›
have *False*
by *simp*
}
thus *?thesis*
by *force*
qed

7.5 Completeness

In this section we will first show that each final state is either *SAT* or *UNSAT* state.

lemma *finalNonConflictState*:
fixes *state*::*State* **and** *FO* :: *Formula*
assumes
getConflictFlag state = *False* **and**
¬ *applicableDecide state decisionVars* **and**
¬ *applicableConflict state*
shows ¬ *formulaFalse* (*getF state*) (*elements* (*getM state*)) **and**
vars (*elements* (*getM state*)) ⊇ *decisionVars*
proof−
from ‹¬ *applicableConflict state*› ‹*getConflictFlag state* = *False*›
show ¬ *formulaFalse* (*getF state*) (*elements* (*getM state*))
unfolding *applicableConflictCharacterization*
by (*auto simp add*:*formulaFalseIffContainsFalseClause formulaEntailsItsClauses*)
show *vars* (*elements* (*getM state*)) ⊇ *decisionVars*
proof
fix *x* :: *Variable*
let *?l* = *Pos x*
assume *x* ∈ *decisionVars*
hence *var ?l* = *x* **and** *var ?l* ∈ *decisionVars* **and** *var* (*opposite ?l*) ∈ *decisionVars*
by *auto*
with ‹¬ *applicableDecide state decisionVars*›
have *literalTrue ?l* (*elements* (*getM state*)) ∨ *literalFalse ?l* (*elements* (*getM state*))
unfolding *applicableDecideCharacterization*
by *force*
with ‹*var ?l* = *x*›
show *x* ∈ *vars* (*elements* (*getM state*))
using *valuationContainsItsLiteralsVariable*[*of ?l elements* (*getM state*)]
using *valuationContainsItsLiteralsVariable*[*of opposite ?l elements* (*getM state*)]

by *auto*
qed
qed

lemma *finalConflictingState*:
fixes *state* :: *State*
assumes
InvariantUniq (*getM state*) **and**
InvariantReasonClauses (*getF state*) (*getM state*) **and**
InvariantCFalse (*getConflictFlag state*) (*getM state*) (*getC state*) **and**
¬ *applicableExplain state* **and**
¬ *applicableBackjump state* **and**
getConflictFlag state
shows
getC state = []
proof (*cases* ∀ *l*. *l el getC state* ⟶ *opposite l el decisions* (*getM state*))
case *True*
{
assume *getC state* ≠ []
let *?l* = *getLastAssertedLiteral* (*oppositeLiteralList* (*getC state*)) (*elements* (*getM state*))

from ‹*InvariantUniq* (*getM state*)›
have *uniq* (*elements* (*getM state*))
unfolding *InvariantUniq-def*
.

from ‹*getConflictFlag state*› ‹*InvariantCFalse* (*getConflictFlag state*) (*getM state*) (*getC state*)›
have *clauseFalse* (*getC state*) (*elements* (*getM state*))
unfolding *InvariantCFalse-def*
by *simp*

with ‹*getC state* ≠ []›
‹*InvariantUniq* (*getM state*)›
have *isLastAssertedLiteral ?l* (*oppositeLiteralList* (*getC state*)) (*elements* (*getM state*))
unfolding *InvariantUniq-def*
using *getLastAssertedLiteralCharacterization*
by *simp*

with *True* ‹*uniq* (*elements* (*getM state*))›
have ∃ *level*. (*isBackjumpLevel level* (*opposite ?l*) (*getC state*) (*getM state*))
using *allDecisionsThenExistsBackjumpLevel* [*of getM state getC state opposite ?l*]
by *simp*
then

obtain *level*::*nat* **where**
isBackjumpLevel level (*opposite ?l*) (*getC state*) (*getM state*)
by *auto*
with ‹*getConflictFlag state*›
have *applicableBackjump state*
unfolding *applicableBackjumpCharacterization*
by *auto*
with ‹¬ *applicableBackjump state*›
have *False*
by *simp*
}
thus *?thesis*
by *auto*
next
case *False*
then obtain *literal*::*Literal* **where** *literal el getC state* ¬ *opposite literal el decisions* (*getM state*)
by *auto*
with ‹*InvariantReasonClauses* (*getF state*) (*getM state*)› ‹*InvariantCFalse* (*getConflictFlag state*) (*getM state*) (*getC state*)› ‹*getConflictFlag state*›
have ∃ *c*. *formulaEntailsClause* (*getF state*) *c* ∧ *isReason c* (*opposite literal*) (*elements* (*getM state*))
using *explainApplicableToEachNonDecision*[*of getF state getM state getConflictFlag state getC state opposite literal*]
by *auto*
then obtain *c*::*Clause*
where *formulaEntailsClause* (*getF state*) *c isReason c* (*opposite literal*) (*elements* (*getM state*))
by *auto*
with ‹¬ *applicableExplain state*› ‹*getConflictFlag state*› ‹*literal el* (*getC state*)›
have *False*
unfolding *applicableExplainCharacterization*
by *auto*
thus *?thesis*
by *simp*
qed

lemma *finalStateCharacterizationLemma*:
fixes *state* :: *State*
assumes
InvariantUniq (*getM state*) **and**
InvariantReasonClauses (*getF state*) (*getM state*) **and**
InvariantCFalse (*getConflictFlag state*) (*getM state*) (*getC state*) **and**
¬ *applicableDecide state decisionVars* **and**
¬ *applicableConflict state*
¬ *applicableExplain state* **and**
¬ *applicableBackjump state*

shows
(*getConflictFlag state* = *False* ∧
¬*formulaFalse* (*getF state*) (*elements* (*getM state*)) ∧
vars (*elements* (*getM state*)) ⊇ *decisionVars*) ∨
(*getConflictFlag state* = *True* ∧
getC state = [])
proof (*cases getConflictFlag state*)
case *True*
hence *getC state* = []
using *assms*
using *finalConflictingState*
by *auto*
with *True*
show *?thesis*
by *simp*
next
case *False*
hence ¬*formulaFalse* (*getF state*) (*elements* (*getM state*)) **and** *vars* (*elements* (*getM state*)) ⊇ *decisionVars*
using *assms*
using *finalNonConflictState*
by *auto*
with *False*
show *?thesis*
by *simp*
qed

theorem *finalStateCharacterization*:
fixes *F0* :: *Formula* **and** *decisionVars* :: *Variable set* **and** *state0* :: *State* **and** *state* :: *State*
assumes
isInitialState state0 F0 **and**
(*state0*, *state*) ∈ *transitionRelation F0 decisionVars* **and**
isFinalState state F0 decisionVars
shows
(*getConflictFlag state* = *False* ∧
¬*formulaFalse* (*getF state*) (*elements* (*getM state*)) ∧
vars (*elements* (*getM state*)) ⊇ *decisionVars*) ∨
(*getConflictFlag state* = *True* ∧
getC state = [])

proof−
from ‹*isInitialState state0 F0*› ‹(*state0*, *state*) ∈ *transitionRelation F0 decisionVars*›
have *invariantsHoldInState state F0 decisionVars*
using *invariantsHoldInValidRunsFromInitialState*
by *simp*
hence

*: *InvariantUniq* (*getM state*)
InvariantReasonClauses (*getF state*) (*getM state*)
InvariantCFalse (*getConflictFlag state*) (*getM state*) (*getC state*)
unfolding *invariantsHoldInState-def*
by *auto*

from ‹*isFinalState state F0 decisionVars*›
have **:
¬ *applicableDecide state decisionVars*
¬ *applicableConflict state*
¬ *applicableExplain state*
¬ *applicableLearn state*
¬ *applicableBackjump state*
unfolding *finalStateNonApplicable*
by *auto*

from * **
show *?thesis*
using *finalStateCharacterizationLemma*[*of state decisionVars*]
by *simp*
qed

Completeness theorems are easy consequences of this characterization and soundness.

theorem *completenessForSAT*:
fixes *F0* :: *Formula* **and** *decisionVars* :: *Variable set* **and** *state0* :: *State* **and** *state* :: *State*
assumes
satisfiable F0 **and**

isInitialState state0 F0 **and**
(*state0*, *state*) ∈ *transitionRelation F0 decisionVars* **and**
isFinalState state F0 decisionVars

shows *getConflictFlag state* = *False* ∧ ¬*formulaFalse* (*getF state*) (*elements* (*getM state*)) ∧
vars (*elements* (*getM state*)) ⊇ *decisionVars*

proof−
from *assms*
have *: (*getConflictFlag state* = *False* ∧
¬*formulaFalse* (*getF state*) (*elements* (*getM state*)) ∧
vars (*elements* (*getM state*)) ⊇ *decisionVars*) ∨
(*getConflictFlag state* = *True* ∧
getC state = [])
using *finalStateCharacterization*[*of state0 F0 state decisionVars*]
by *auto*
{
assume ¬ (*getConflictFlag state* = *False*)

```
    with ∗
    have getConflictFlag state = True getC state = []
      by auto
    with assms
      have ¬ satisfiable F0
      using soundnessForUNSAT
      by simp
    with ‹satisfiable F0›
    have False
      by simp
  }
  with ∗ show ?thesis
    by auto
qed


theorem completenessForUNSAT:
  fixes F0 :: Formula and decisionVars :: Variable set and state0 ::
State and state :: State
  assumes
  vars F0 ⊆ decisionVars and

  ¬ satisfiable F0 and

  isInitialState state0 F0 and
  (state0, state) ∈ transitionRelation F0 decisionVars and
  isFinalState state F0 decisionVars

  shows
  getConflictFlag state = True ∧ getC state = []

proof−
  from assms
  have ∗: (getConflictFlag state = False ∧
             ¬formulaFalse (getF state) (elements (getM state)) ∧
             vars (elements (getM state)) ⊇ decisionVars) ∨
          (getConflictFlag state = True ∧
             getC state = [])
    using finalStateCharacterization[of state0 F0 state decisionVars]
    by auto
  {
    assume ¬ getConflictFlag state = True
    with ∗
     have getConflictFlag state = False ∧ ¬formulaFalse (getF state)
(elements (getM state)) ∧ vars (elements (getM state)) ⊇ decisionVars
      by simp
    with assms
    have satisfiable F0
      using soundnessForSAT[of F0 decisionVars state0 state]
```

unfolding *satisfiable-def*
by *auto*
with ‹¬ *satisfiable F0*›
have *False*
by *simp*
}
with ∗ **show** *?thesis*
by *auto*
qed

theorem *partialCorrectness*:
fixes *F0* :: *Formula* **and** *decisionVars* :: *Variable set* **and** *state0* :: *State* **and** *state* :: *State*
assumes
vars F0 ⊆ *decisionVars* **and**

isInitialState state0 F0 **and**
(*state0*, *state*) ∈ *transitionRelation F0 decisionVars* **and**
isFinalState state F0 decisionVars

shows
satisfiable F0 = (¬ *getConflictFlag state*)

using *assms*
using *completenessForUNSAT*[*of F0 decisionVars state0 state*]
using *completenessForSAT*[*of F0 state0 state decisionVars*]
by *auto*

end

8 Functional implementation of a SAT solver with Two Watch literal propagation.

theory *SatSolverCode*
imports *SatSolverVerification HOL−Library.Code-Target-Numeral*
begin

8.1 Specification

lemma [*code-unfold*]:
fixes *literal* :: *Literal* **and** *clause* :: *Clause*
shows *literal el clause* = *List.member clause literal*
by (*auto simp add*: *member-def*)

datatype *ExtendedBool* = *TRUE* | *FALSE* | *UNDEF*

record *State* =

— Satisfiability flag: UNDEF, TRUE or FALSE
getSATFlag :: *ExtendedBool*
— Formula
getF :: *Formula*
— Assertion Trail
getM :: *LiteralTrail*
— Conflict flag
getConflictFlag :: *bool* — raised iff M falsifies F
— Conflict clause index
getConflictClause :: *nat* — corresponding clause from F is false in M
— Unit propagation queue
getQ :: *Literal list*
— Unit propagation graph
getReason :: *Literal* ⇒ *nat option* — index of a clause that is a reason for propagation of a literal
— Two-watch literal scheme
— clause indices instead of clauses are used
getWatch1 :: *nat* ⇒ *Literal option* — First watch of a clause
getWatch2 :: *nat* ⇒ *Literal option* — Second watch of a clause
getWatchList :: *Literal* ⇒ *nat list* — Watch list of a given literal
— Conflict analysis data structures
getC :: *Clause* — Conflict analysis clause - always false in M
getCl :: *Literal* — Last asserted literal in (opposite getC)
getCll :: *Literal* — Second last asserted literal in (opposite getC)
getCn :: *nat* — Number of literals of (opposite getC) on the (currentLevel M)

definition
setWatch1 :: *nat* ⇒ *Literal* ⇒ *State* ⇒ *State*
where
setWatch1 clause literal state =
  state⦇ *getWatch1* := (*getWatch1 state*)(*clause* := *Some literal*),
    getWatchList := (*getWatchList state*)(*literal* := *clause* # (*getWatchList state literal*))
  ⦈

declare *setWatch1-def*[*code-unfold*]

definition
setWatch2 :: *nat* ⇒ *Literal* ⇒ *State* ⇒ *State*
where
setWatch2 clause literal state =
  state⦇ *getWatch2* := (*getWatch2 state*)(*clause* := *Some literal*),
    getWatchList := (*getWatchList state*)(*literal* := *clause* # (*getWatchList state literal*))
  ⦈

declare *setWatch2-def*[*code-unfold*]

definition
swapWatches :: *nat* ⇒ *State* ⇒ *State*
where
swapWatches clause state ==
  state⦇ *getWatch1* := (*getWatch1 state*)(*clause* := (*getWatch2 state clause*)),
      getWatch2 := (*getWatch2 state*)(*clause* := (*getWatch1 state clause*))
    ⦈

declare *swapWatches-def*[*code-unfold*]

primrec *getNonWatchedUnfalsifiedLiteral* :: *Clause* ⇒ *Literal* ⇒ *Literal* ⇒ *LiteralTrail* ⇒ *Literal option*
where
getNonWatchedUnfalsifiedLiteral [] *w1 w2 M* = *None* |
getNonWatchedUnfalsifiedLiteral (*literal* # *clause*) *w1 w2 M* =
  (*if literal* ≠ *w1* ∧
    literal ≠ *w2* ∧
    ¬ (*literalFalse literal* (*elements M*)) *then*
      Some literal
  else
      getNonWatchedUnfalsifiedLiteral clause w1 w2 M
)

definition
setReason :: *Literal* ⇒ *nat* ⇒ *State* ⇒ *State*
where
setReason literal clause state =
  state⦇ *getReason* := (*getReason state*)(*literal* := *Some clause*) ⦈

declare *setReason-def*[*code-unfold*]

primrec *notifyWatches-loop*::*Literal* ⇒ *nat list* ⇒ *nat list* ⇒ *State* ⇒ *State*
where
notifyWatches-loop literal [] *newWl state* = *state*⦇ *getWatchList* := (*getWatchList state*)(*literal* := *newWl*) ⦈ |
notifyWatches-loop literal (*clause* # *list'*) *newWl state* =
  (*let state'* = (*if Some literal* = (*getWatch1 state clause*) *then*
          (*swapWatches clause state*)
        else
          state) *in*
  case (*getWatch1 state' clause*) *of*

```
        None ⇒ state
    |   Some w1 ⇒ (
    case (getWatch2 state' clause) of
        None ⇒ state
    |   Some w2 ⇒
    (if (literalTrue w1 (elements (getM state'))) then
        notifyWatches-loop literal list' (clause # newWl) state'
     else
       (case (getNonWatchedUnfalsifiedLiteral (nth (getF state') clause)
w1 w2 (getM state')) of
            Some l' ⇒
                notifyWatches-loop literal list' newWl (setWatch2 clause
l' state')
          | None ⇒
                (if (literalFalse w1 (elements (getM state'))) then
                           let state'' = (state'(| getConflictFlag := True,
getConflictClause := clause |)) in
                  notifyWatches-loop literal list' (clause # newWl) state''
                else
                     let state'' = state'(| getQ := (if w1 el (getQ state')
then
                                                (getQ state')
                                            else
                                                (getQ state') @ [w1]
                                            )
                                  |) in
                   let state''' = (setReason w1 clause state'') in
                  notifyWatches-loop literal list' (clause # newWl) state'''
                )
        )
    )
    )
    )
```

definition
notifyWatches :: *Literal* ⇒ *State* ⇒ *State*
where
notifyWatches literal state ==
    notifyWatches-loop literal (*getWatchList state literal*) [] *state*

declare *notifyWatches-def*[*code-unfold*]

definition
assertLiteral :: *Literal* ⇒ *bool* ⇒ *State* ⇒ *State*
where
assertLiteral literal decision state ==
    let state' = (*state*(| *getM* := (*getM state*) @ [(*literal*, *decision*)] |))

in
notifyWatches (*opposite literal*) *state′*

definition
applyUnitPropagate :: *State* ⇒ *State*
where
applyUnitPropagate state =
(*let state′* = (*assertLiteral* (*hd* (*getQ state*)) *False state*) *in*
state′⦇ *getQ* := *tl* (*getQ state′*)⦈)

partial-function (*tailrec*)
exhaustiveUnitPropagate :: *State* ⇒ *State*
where
exhaustiveUnitPropagate-unfold[*code*]:
exhaustiveUnitPropagate state =
(*if* (*getConflictFlag state*) ∨ (*getQ state*) = [] *then*
state
else
exhaustiveUnitPropagate (*applyUnitPropagate state*)
)

inductive
exhaustiveUnitPropagate-dom :: *State* ⇒ *bool*
where
step: (¬ *getConflictFlag state* ⟹ *getQ state* ≠ []
⟹ *exhaustiveUnitPropagate-dom* (*applyUnitPropagate state*))
⟹ *exhaustiveUnitPropagate-dom state*

definition
addClause :: *Clause* ⇒ *State* ⇒ *State*
where
addClause clause state =
(*let clause′* = (*remdups* (*removeFalseLiterals clause* (*elements* (*getM state*)))) *in*
(*if* (*clauseTrue clause′* (*elements* (*getM state*))) *then*
state
else (*if clause′*=[] *then*
state⦇ *getSATFlag* := *FALSE* ⦈
else (*if* (*length clause′* = *1*) *then*
let state′ = (*assertLiteral* (*hd clause′*) *False state*) *in*
exhaustiveUnitPropagate state′
else (*if* (*clauseTautology clause′*) *then*
state
else

```
        let clauseIndex = length (getF state) in
        let state'   = state⦇ getF := (getF state) @ [clause']⦈ in
        let state''  = setWatch1 clauseIndex (nth clause' 0) state' in
        let state''' = setWatch2 clauseIndex (nth clause' 1) state'' in
        state'''
  )))
 ))
```

definition
initialState :: *State*
where

```
initialState =
   ⦇ getSATFlag = UNDEF,
     getF = [],
     getM = [],
     getConflictFlag = False,
     getConflictClause = 0,
     getQ = [],
     getReason = λ l. None,
     getWatch1 = λ c. None,
     getWatch2 = λ c. None,
     getWatchList = λ l. [],
     getC = [],
     getCl = (Pos 0),
     getCll = (Pos 0),
     getCn = 0
   ⦈
```

primrec *initialize* :: *Formula* ⇒ *State* ⇒ *State*
where
initialize [] *state* = *state* |
initialize (*clause* # *formula*) *state* = *initialize formula* (*addClause clause state*)

definition
findLastAssertedLiteral :: *State* ⇒ *State*
where
findLastAssertedLiteral state =
state ⦇ *getCl* := *getLastAssertedLiteral* (*oppositeLiteralList* (*getC state*)) (*elements* (*getM state*)) ⦈

definition
countCurrentLevelLiterals :: *State* ⇒ *State*
where
countCurrentLevelLiterals state =
  (*let cl* = *currentLevel* (*getM state*) *in*
      state ⦇ *getCn* := *length* (*filter* (λ *l*. *elementLevel* (*opposite l*)

(*getM state*) = *cl*) (*getC state*)) |))

definition *setConflictAnalysisClause* :: *Clause* ⇒ *State* ⇒ *State*
where
setConflictAnalysisClause clause state =
  (*let oppM0* = *oppositeLiteralList* (*elements* (*prefixToLevel 0* (*getM state*))) *in*
    let state' = *state* (| *getC* := *remdups* (*list-diff clause oppM0*) |) *in*
      countCurrentLevelLiterals (*findLastAssertedLiteral state'*)
)

definition
applyConflict :: *State* ⇒ *State*
where
applyConflict state =
   (*let conflictClause* = (*nth* (*getF state*) (*getConflictClause state*)) *in*
    setConflictAnalysisClause conflictClause state)

definition
applyExplain :: *Literal* ⇒ *State* ⇒ *State*
where
applyExplain literal state =
    (*case* (*getReason state literal*) *of*
        None ⇒
            state
    | *Some reason* ⇒
            let res = *resolve* (*getC state*) (*nth* (*getF state*) *reason*) (*opposite literal*) *in*
            setConflictAnalysisClause res state

)

partial-function (*tailrec*)
applyExplainUIP :: *State* ⇒ *State*
where
applyExplainUIP-unfold:
applyExplainUIP state =
    (*if* (*getCn state* = *1*) *then*
        state
     else
        applyExplainUIP (*applyExplain* (*getCl state*) *state*)
)

inductive
applyExplainUIP-dom :: *State* ⇒ *bool*
where
step:

(*getCn state* ≠ *1*
  ⟹ *applyExplainUIP-dom* (*applyExplain* (*getCl state*) *state*))
 ⟹ *applyExplainUIP-dom state*

definition
applyLearn :: *State* ⇒ *State*
where
applyLearn state =
    (*if getC state* = [*opposite* (*getCl state*)] *then*
      state
    else
      let state′ = *state*⦇ *getF* := (*getF state*) @ [*getC state*] ⦈ *in*
      let l = (*getCl state*) *in*
    let ll = (*getLastAssertedLiteral* (*removeAll l* (*oppositeLiteralList* (*getC state*))) (*elements* (*getM state*))) *in*
      let clauseIndex = *length* (*getF state*) *in*
      let state″ = *setWatch1 clauseIndex* (*opposite l*) *state′ in*
      let state‴ = *setWatch2 clauseIndex* (*opposite ll*) *state″ in*
      state‴⦇ *getCll* := *ll* ⦈
)

definition
getBackjumpLevel :: *State* ⇒ *nat*
where
getBackjumpLevel state ==
  (*if getC state* = [*opposite* (*getCl state*)] *then*
    0
  else
    elementLevel (*getCll state*) (*getM state*)
)

definition
applyBackjump :: *State* ⇒ *State*
where
applyBackjump state =
  (*let l* = (*getCl state*) *in*
  let level = *getBackjumpLevel state in*
  let state′ = *state*⦇ *getConflictFlag* := *False*, *getQ* := [], *getM* := (*prefixToLevel level* (*getM state*))⦈ *in*
  let state″ = (*if level* > *0 then setReason* (*opposite l*) (*length* (*getF state*) − *1*) *state′ else state′*) *in*
  assertLiteral (*opposite l*) *False state″*
)

axiomatization *selectLiteral* :: *State* ⇒ *Variable set* ⇒ *Literal*

where
selectLiteral-def:
$Vbl - vars\ (elements\ (getM\ state)) \neq \{\} \longrightarrow$
$\quad var\ (selectLiteral\ state\ Vbl) \in (Vbl - vars\ (elements\ (getM\ state)))$

definition
applyDecide :: *State* ⇒ *Variable set* ⇒ *State*
where
applyDecide state Vbl =
  assertLiteral (*selectLiteral state Vbl*) *True state*

definition
solve-loop-body :: *State* ⇒ *Variable set* ⇒ *State*
where
```
solve-loop-body state Vbl =
  (let state' = exhaustiveUnitPropagate state in
  (if (getConflictFlag state') then
    (if (currentLevel (getM state')) = 0 then
      state'⦇ getSATFlag := FALSE ⦈
     else
      (applyBackjump
      (applyLearn
      (applyExplainUIP
      (applyConflict
        state'
      )
      )
      )
      )
     )
   else
    (if (vars (elements (getM state')) ⊇ Vbl) then
      state'⦇ getSATFlag := TRUE ⦈
     else
      applyDecide state' Vbl
    )
  )
  )
```

partial-function (*tailrec*)
solve-loop :: *State* ⇒ *Variable set* ⇒ *State*
where
solve-loop-unfold:
```
solve-loop state Vbl =
  (if (getSATFlag state) ≠ UNDEF then
    state
```

else
let state′ = solve-loop-body state Vbl in
solve-loop state′ Vbl
)

inductive
solve-loop-dom :: *State* ⇒ *Variable set* ⇒ *bool*
where
step:
(*getSATFlag state* = *UNDEF*
⟹ *solve-loop-dom* (*solve-loop-body state Vbl*) *Vbl*)
⟹ *solve-loop-dom state Vbl*

definition *solve*::*Formula* ⇒ *ExtendedBool*
where
solve F0 =
(*getSATFlag*
(*solve-loop*
(*initialize F0 initialState*) (*vars F0*)
)
)

definition
InvariantWatchListsContainOnlyClausesFromF :: (*Literal* ⇒ *nat list*) ⇒ *Formula* ⇒ *bool*
where
InvariantWatchListsContainOnlyClausesFromF Wl F =
(∀ (*l*::*Literal*) (*c*::*nat*). *c* ∈ *set* (*Wl l*) ⟶ *0* ≤ *c* ∧ *c* < *length F*)

definition
InvariantWatchListsUniq :: (*Literal* ⇒ *nat list*) ⇒ *bool*
where
InvariantWatchListsUniq Wl =
(∀ *l*. *uniq* (*Wl l*))

definition
InvariantWatchListsCharacterization :: (*Literal* ⇒ *nat list*) ⇒ (*nat* ⇒ *Literal option*) ⇒ (*nat* ⇒ *Literal option*) ⇒ *bool*
where

InvariantWatchListsCharacterization Wl w1 w2 =
  (∀ (*c*::*nat*) (*l*::*Literal*). *c* ∈ *set* (*Wl l*) = (*Some l* = (*w1 c*) ∨ *Some l* = (*w2 c*)))

definition
InvariantWatchesEl :: *Formula* ⇒ (*nat* ⇒ *Literal option*) ⇒ (*nat* ⇒ *Literal option*) ⇒ *bool*
where
InvariantWatchesEl formula watch1 watch2 ==
  ∀ (*clause*::*nat*). *0* ≤ *clause* ∧ *clause* < *length formula* ⟶
    (∃ (*w1*::*Literal*) (*w2*::*Literal*). *watch1 clause* = *Some w1* ∧ *watch2 clause* = *Some w2* ∧
      w1 el (*nth formula clause*) ∧ *w2 el* (*nth formula clause*))

definition
InvariantWatchesDiffer :: *Formula* ⇒ (*nat* ⇒ *Literal option*) ⇒ (*nat* ⇒ *Literal option*) ⇒ *bool*
where
InvariantWatchesDiffer formula watch1 watch2 ==
  ∀ (*clause*::*nat*). *0* ≤ *clause* ∧ *clause* < *length formula* ⟶ *watch1 clause* ≠ *watch2 clause*

definition
watchCharacterizationCondition::*Literal* ⇒ *Literal* ⇒ *LiteralTrail* ⇒ *Clause* ⇒ *bool*
where
watchCharacterizationCondition w1 w2 M clause =
  (*literalFalse w1* (*elements M*) ⟶
    ((∃ *l*. *l el clause* ∧ *literalTrue l* (*elements M*) ∧ *elementLevel l M* ≤ *elementLevel* (*opposite w1*) *M*) ∨
      (∀ *l*. *l el clause* ∧ *l* ≠ *w1* ∧ *l* ≠ *w2* ⟶
        literalFalse l (*elements M*) ∧ *elementLevel* (*opposite l*) *M* ≤ *elementLevel* (*opposite w1*) *M*)
)
)

definition
InvariantWatchCharacterization::*Formula* ⇒ (*nat* ⇒ *Literal option*) ⇒ (*nat* ⇒ *Literal option*) ⇒ *LiteralTrail* ⇒ *bool*
where
InvariantWatchCharacterization F watch1 watch2 M =
  (∀ *c w1 w2*. (*0* ≤ *c* ∧ *c* < *length F* ∧ *Some w1* = *watch1 c* ∧ *Some w2* = *watch2 c*) ⟶
    watchCharacterizationCondition w1 w2 M (*nth F c*) ∧
    watchCharacterizationCondition w2 w1 M (*nth F c*)

)

definition
InvariantQCharacterization :: *bool* ⇒ *Literal list* ⇒ *Formula* ⇒ *LiteralTrail* ⇒ *bool*
where
InvariantQCharacterization conflictFlag Q F M ==
¬ *conflictFlag* ⟶ (∀ (*l*::*Literal*). *l el Q* = (∃ (*c*::*Clause*). *c el F* ∧ *isUnitClause c l* (*elements M*)))

definition
InvariantUniqQ :: *Literal list* ⇒ *bool*
where
InvariantUniqQ Q =
uniq Q

definition
InvariantConflictFlagCharacterization :: *bool* ⇒ *Formula* ⇒ *LiteralTrail* ⇒ *bool*
where
InvariantConflictFlagCharacterization conflictFlag F M ==
conflictFlag = *formulaFalse F* (*elements M*)

definition
InvariantNoDecisionsWhenConflict :: *Formula* ⇒ *LiteralTrail* ⇒ *nat* ⇒ *bool*
where
InvariantNoDecisionsWhenConflict F M level=
(∀ *level'*. *level'* < *level* ⟶
¬ *formulaFalse F* (*elements* (*prefixToLevel level' M*))
)

definition
InvariantNoDecisionsWhenUnit :: *Formula* ⇒ *LiteralTrail* ⇒ *nat* ⇒ *bool*
where
InvariantNoDecisionsWhenUnit F M level =
(∀ *level'*. *level'* < *level* ⟶
¬ (∃ *clause literal*. *clause el F* ∧
isUnitClause clause literal (*elements* (*prefixToLevel level' M*)))
)

definition *InvariantEquivalentZL* :: *Formula* ⇒ *LiteralTrail* ⇒ *Formula* ⇒ *bool*
where
InvariantEquivalentZL F M F0 =
equivalentFormulae (*F* @ *val2form* (*elements* (*prefixToLevel 0 M*))) *F0*

definition
InvariantGetReasonIsReason :: (*Literal* ⇒ *nat option*) ⇒ *Formula* ⇒ *LiteralTrail* ⇒ *Literal set* ⇒ *bool*
where
InvariantGetReasonIsReason GetReason F M Q ==
∀ *literal*. (*literal el* (*elements M*) ∧ ¬ *literal el* (*decisions M*) ∧ *elementLevel literal M* > *0* ⟶
(∃ (*reason*::*nat*). (*GetReason literal*) = *Some reason* ∧ *0* ≤ *reason* ∧ *reason* < *length F* ∧
isReason (*nth F reason*) *literal* (*elements M*)
)
) ∧
(*currentLevel M* > *0* ∧ *literal* ∈ *Q* ⟶
(∃ (*reason*::*nat*). (*GetReason literal*) = *Some reason* ∧ *0* ≤ *reason* ∧ *reason* < *length F* ∧
(*isUnitClause* (*nth F reason*) *literal* (*elements M*) ∨ *clauseFalse* (*nth F reason*) (*elements M*))
)
)

definition
InvariantConflictClauseCharacterization :: *bool* ⇒ *nat* ⇒ *Formula* ⇒ *LiteralTrail* ⇒ *bool*
where
InvariantConflictClauseCharacterization conflictFlag conflictClause F M ==
conflictFlag ⟶ (*conflictClause* < *length F* ∧
clauseFalse (*nth F conflictClause*) (*elements M*))

definition
InvariantClCharacterization :: *Literal* ⇒ *Clause* ⇒ *LiteralTrail* ⇒ *bool*

where
InvariantClCharacterization Cl C M ==
isLastAssertedLiteral Cl (*oppositeLiteralList C*) (*elements M*)

definition
InvariantCllCharacterization :: *Literal* ⇒ *Literal* ⇒ *Clause* ⇒ *LiteralTrail* ⇒ *bool*
where

InvariantCllCharacterization Cl Cll C M ==
  set C ≠ {*opposite Cl*} ⟶
    isLastAssertedLiteral Cll (*removeAll Cl* (*oppositeLiteralList C*)) (*elements M*)

definition
InvariantClCurrentLevel :: *Literal* ⇒ *LiteralTrail* ⇒ *bool*
where
InvariantClCurrentLevel Cl M ==
  elementLevel Cl M = *currentLevel M*

definition
InvariantCnCharacterization :: *nat* ⇒ *Clause* ⇒ *LiteralTrail* ⇒ *bool*
where
InvariantCnCharacterization Cn C M ==
  Cn = *length* (*filter* (λ *l*. *elementLevel* (*opposite l*) *M* = *currentLevel M*) (*remdups C*))

definition
InvariantUniqC :: *Clause* ⇒ *bool*
where
InvariantUniqC clause = *uniq clause*

definition
InvariantVarsQ :: *Literal list* ⇒ *Formula* ⇒ *Variable set* ⇒ *bool*
where
InvariantVarsQ Q F0 Vbl ==
  vars Q ⊆ *vars F0* ∪ *Vbl*

end

theory *AssertLiteral*
imports *SatSolverCode*
begin

lemma *getNonWatchedUnfalsifiedLiteralSomeCharacterization*:
fixes *clause* :: *Clause* **and** *w1* :: *Literal* **and** *w2* :: *Literal* **and** *M* :: *LiteralTrail* **and** *l* :: *Literal*
assumes
  getNonWatchedUnfalsifiedLiteral clause w1 w2 M = *Some l*
shows

l el clause l ≠ w1 l ≠ w2 ¬ literalFalse l (*elements M*)
using *assms*
by (*induct clause*) (*auto split*: *if-split-asm*)

lemma *getNonWatchedUnfalsifiedLiteralNoneCharacterization*:
fixes *clause* :: *Clause* **and** *w1* :: *Literal* **and** *w2* :: *Literal* **and** *M* :: *LiteralTrail*
assumes
getNonWatchedUnfalsifiedLiteral clause w1 w2 M = None
shows
∀ *l. l el clause* ∧ *l ≠ w1* ∧ *l ≠ w2* ⟶ *literalFalse l* (*elements M*)
using *assms*
by (*induct clause*) (*auto split*: *if-split-asm*)

lemma *swapWatchesEffect*:
fixes *clause*::*nat* **and** *state*::*State* **and** *clause'*::*nat*
shows
getWatch1 (*swapWatches clause state*) *clause'* = (*if clause* = *clause'* *then getWatch2 state clause' else getWatch1 state clause'*) **and**
getWatch2 (*swapWatches clause state*) *clause'* = (*if clause* = *clause'* *then getWatch1 state clause' else getWatch2 state clause'*)
unfolding *swapWatches-def*
by *auto*

lemma *notifyWatchesLoopPreservedVariables*:
fixes *literal* :: *Literal* **and** *Wl* :: *nat list* **and** *newWl* :: *nat list* **and** *state* :: *State*
assumes
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
and
∀ (*c*::*nat*). *c* ∈ *set Wl* ⟶ *0* ≤ *c* ∧ *c* < *length* (*getF state*)
shows
let state' = (*notifyWatches-loop literal Wl newWl state*) *in*
(*getM state'*) = (*getM state*) ∧
(*getF state'*) = (*getF state*) ∧
(*getSATFlag state'*) = (*getSATFlag state*) ∧
isPrefix (*getQ state*) (*getQ state'*)

using *assms*
proof (*induct Wl arbitrary*: *newWl state*)
case *Nil*

```
  thus ?case
    unfolding isPrefix-def
    by simp
next
  case (Cons clause Wl')
  from ‹∀ (c::nat). c ∈ set (clause # Wl') ⟶ 0 ≤ c ∧ c < length
(getF state)›
  have 0 ≤ clause ∧ clause < length (getF state)
    by auto
  then obtain wa::Literal and wb::Literal
     where getWatch1 state clause = Some wa and getWatch2 state
clause = Some wb
    using Cons
    unfolding InvariantWatchesEl-def
    by auto
  show ?case
  proof (cases Some literal = getWatch1 state clause)
    case True
    let ?state' = swapWatches clause state
    let ?w1 = wb
    have getWatch1 ?state' clause = Some ?w1
      using ‹getWatch2 state clause = Some wb›
      unfolding swapWatches-def
      by auto
    let ?w2 = wa
    have getWatch2 ?state' clause = Some ?w2
      using ‹getWatch1 state clause = Some wa›
      unfolding swapWatches-def
      by auto
    show ?thesis
    proof (cases literalTrue ?w1 (elements (getM ?state')))
      case True

      from Cons(2)
         have InvariantWatchesEl (getF ?state') (getWatch1 ?state')
(getWatch2 ?state')
        unfolding InvariantWatchesEl-def
        unfolding swapWatches-def
        by auto
      moreover
      have getM ?state' = getM state ∧
        getF ?state' = getF state ∧
        getSATFlag ?state' = getSATFlag state ∧
        getQ ?state' = getQ state

        unfolding swapWatches-def
        by simp
      ultimately
      show ?thesis
```

using *Cons*(*1*)[*of ?state′ clause # newWl*]
using *Cons*(*3*)
using ‹*getWatch1 ?state′ clause = Some ?w1*›
using ‹*getWatch2 ?state′ clause = Some ?w2*›
using ‹*Some literal = getWatch1 state clause*›
using ‹*literalTrue ?w1* (*elements* (*getM ?state′*))›
by (*simp add*:*Let-def*)
next
case *False*
show *?thesis*
proof (*cases getNonWatchedUnfalsifiedLiteral* (*nth* (*getF ?state′*) *clause*) *?w1 ?w2* (*getM ?state′*))
case (*Some l′*)
hence *l′ el* (*nth* (*getF ?state′*) *clause*)
using *getNonWatchedUnfalsifiedLiteralSomeCharacterization*
by *simp*

let *?state″ = setWatch2 clause l′ ?state′*

from *Cons*(*2*)
have *InvariantWatchesEl* (*getF ?state″*) (*getWatch1 ?state″*) (*getWatch2 ?state″*)
using ‹*l′ el* (*nth* (*getF ?state′*) *clause*)›
unfolding *InvariantWatchesEl-def*
unfolding *swapWatches-def*
unfolding *setWatch2-def*
by *auto*
moreover
have *getM ?state″ = getM state ∧*
getF ?state″ = getF state ∧
getSATFlag ?state″ = getSATFlag state ∧
getQ ?state″ = getQ state
unfolding *swapWatches-def*
unfolding *setWatch2-def*
by *simp*
ultimately
show *?thesis*
using *Cons*(*1*)[*of ?state″ newWl*]
using *Cons*(*3*)
using ‹*getWatch1 ?state′ clause = Some ?w1*›
using ‹*getWatch2 ?state′ clause = Some ?w2*›
using ‹*Some literal = getWatch1 state clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state′*))›
using *Some*
by (*simp add*: *Let-def*)
next
case *None*
show *?thesis*
proof (*cases literalFalse ?w1* (*elements* (*getM ?state′*)))

case *True*
let *?state''* = *?state'*⦇*getConflictFlag* := *True*, *getConflictClause* := *clause*⦈

from *Cons*(*2*)
have *InvariantWatchesEl* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)
unfolding *InvariantWatchesEl-def*
unfolding *swapWatches-def*
by *auto*
moreover
have *getM ?state''* = *getM state* ∧
getF ?state'' = *getF state* ∧
getSATFlag ?state'' = *getSATFlag state* ∧
getQ ?state'' = *getQ state*
unfolding *swapWatches-def*
by *simp*
ultimately
show *?thesis*
using *Cons*(*1*)[*of ?state'' clause # newWl*]
using *Cons*(*3*)
using ‹*getWatch1 ?state' clause* = *Some ?w1*›
using ‹*getWatch2 ?state' clause* = *Some ?w2*›
using ‹*Some literal* = *getWatch1 state clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state'*))›
using *None*
using ‹*literalFalse ?w1* (*elements* (*getM ?state'*))›
by (*simp add*: *Let-def*)
next
case *False*
let *?state''* = *setReason ?w1 clause* (*?state'*⦇*getQ* := (*if ?w1 el* (*getQ ?state'*) *then* (*getQ ?state'*) *else* (*getQ ?state'*) @ [*?w1*])⦈)
from *Cons*(*2*)
have *InvariantWatchesEl* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)
unfolding *InvariantWatchesEl-def*
unfolding *swapWatches-def*
unfolding *setReason-def*
by *auto*
moreover
have *getM ?state''* = *getM state* ∧
getF ?state'' = *getF state* ∧
getSATFlag ?state'' = *getSATFlag state* ∧
getQ ?state'' = (*if ?w1 el* (*getQ state*) *then* (*getQ state*) *else* (*getQ state*) @ [*?w1*])
unfolding *swapWatches-def*
unfolding *setReason-def*
by *auto*
ultimately

show *?thesis*
using *Cons*(*1*)[*of ?state'' clause # newWl*]
using *Cons*(*3*)
using ‹*getWatch1 ?state' clause = Some ?w1*›
using ‹*getWatch2 ?state' clause = Some ?w2*›
using ‹*Some literal = getWatch1 state clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state'*))›
using *None*
using ‹¬ *literalFalse ?w1* (*elements* (*getM ?state'*))›
unfolding *isPrefix-def*
by (*auto simp add*: *Let-def split*: *if-split-asm*)
qed
qed
qed
next
case *False*
let *?state' = state*
let *?w1 = wa*
have *getWatch1 ?state' clause = Some ?w1*
using ‹*getWatch1 state clause = Some wa*›
unfolding *swapWatches-def*
by *auto*
let *?w2 = wb*
have *getWatch2 ?state' clause = Some ?w2*
using ‹*getWatch2 state clause = Some wb*›
unfolding *swapWatches-def*
by *auto*
show *?thesis*
proof (*cases literalTrue ?w1* (*elements* (*getM ?state'*)))
case *True*
thus *?thesis*
using *Cons*
using ‹¬ *Some literal = getWatch1 state clause*›
using ‹*getWatch1 ?state' clause = Some ?w1*›
using ‹*getWatch2 ?state' clause = Some ?w2*›
using ‹*literalTrue ?w1* (*elements* (*getM ?state'*))›
by (*simp add*:*Let-def*)
next
case *False*
show *?thesis*
proof (*cases getNonWatchedUnfalsifiedLiteral* (*nth* (*getF ?state'*) *clause*) *?w1 ?w2* (*getM ?state'*))
case (*Some l'*)
hence *l' el* (*nth* (*getF ?state'*)) *clause*
using *getNonWatchedUnfalsifiedLiteralSomeCharacterization*
by *simp*

let *?state'' = setWatch2 clause l' ?state'*

```
      from Cons(2)
       have InvariantWatchesEl (getF ?state'') (getWatch1 ?state'')
(getWatch2 ?state'')
        using ‹l' el (nth (getF ?state') clause›
        unfolding InvariantWatchesEl-def
        unfolding setWatch2-def
        by auto
      moreover
      have getM ?state'' = getM state ∧
        getF ?state'' = getF state ∧
        getSATFlag ?state'' = getSATFlag state ∧
        getQ ?state'' = getQ state
        unfolding setWatch2-def
        by simp
      ultimately
      show ?thesis
        using Cons(1)[of ?state'']
        using Cons(3)
        using ‹getWatch1 ?state' clause = Some ?w1›
        using ‹getWatch2 ?state' clause = Some ?w2›
        using ‹¬ Some literal = getWatch1 state clause›
        using ‹¬ literalTrue ?w1 (elements (getM ?state'))›
        using Some
        by (simp add: Let-def)
    next
      case None
      show ?thesis
      proof (cases literalFalse ?w1 (elements (getM ?state')))
        case True
      let ?state'' = ?state'⦇getConflictFlag := True, getConflictClause
:= clause⦈

        from Cons(2)
        have InvariantWatchesEl (getF ?state'') (getWatch1 ?state'')
(getWatch2 ?state'')
          unfolding InvariantWatchesEl-def
          by auto
        moreover
        have getM ?state'' = getM state ∧
          getF ?state'' = getF state ∧
          getSATFlag ?state'' = getSATFlag state ∧
          getQ ?state'' = getQ state
          by simp
        ultimately
        show ?thesis
          using Cons(1)[of ?state'']
          using Cons(3)
          using ‹getWatch1 ?state' clause = Some ?w1›
          using ‹getWatch2 ?state' clause = Some ?w2›
```

```
          using ‹¬ Some literal = getWatch1 state clause›
          using ‹¬ literalTrue ?w1 (elements (getM ?state'))›
          using None
          using ‹literalFalse ?w1 (elements (getM ?state'))›
          by (simp add: Let-def)
      next
        case False
        let ?state'' = setReason ?w1 clause (?state'⦇getQ := (if ?w1
el (getQ ?state') then (getQ ?state') else (getQ ?state') @ [?w1])⦈)
        from Cons(2)
        have InvariantWatchesEl (getF ?state'') (getWatch1 ?state'')
(getWatch2 ?state'')
          unfolding InvariantWatchesEl-def
          unfolding setReason-def
          by auto
        moreover
        have getM ?state'' = getM state ∧
          getF ?state'' = getF state ∧
          getSATFlag ?state'' = getSATFlag state ∧
          getQ ?state'' = (if ?w1 el (getQ state) then (getQ state) else
(getQ state) @ [?w1])
          unfolding setReason-def
          by simp
        ultimately
        show ?thesis
          using Cons(1)[of ?state'']
          using Cons(3)
          using ‹getWatch1 ?state' clause = Some ?w1›
          using ‹getWatch2 ?state' clause = Some ?w2›
          using ‹¬ Some literal = getWatch1 state clause›
          using ‹¬ literalTrue ?w1 (elements (getM ?state'))›
          using None
          using ‹¬ literalFalse ?w1 (elements (getM ?state'))›
          unfolding isPrefix-def
          by (auto simp add: Let-def split: if-split-asm)
      qed
    qed
  qed
 qed
qed
```

lemma *notifyWatchesStartQIreleveant*:
fixes *literal* :: *Literal* **and** *Wl* :: *nat list* **and** *newWl* :: *nat list* **and** *state* :: *State*
assumes
InvariantWatchesEl (*getF stateA*) (*getWatch1 stateA*) (*getWatch2 stateA*) **and**
∀ (*c*::*nat*). *c* ∈ *set Wl* ⟶ *0* ≤ *c* ∧ *c* < *length* (*getF stateA*) **and**

getM stateA = *getM stateB* **and**
getF stateA = *getF stateB* **and**
getWatch1 stateA = *getWatch1 stateB* **and**
getWatch2 stateA = *getWatch2 stateB* **and**
getConflictFlag stateA = *getConflictFlag stateB* **and**
getSATFlag stateA = *getSATFlag stateB*
shows
let state′ = (*notifyWatches-loop literal Wl newWl stateA*) *in*
let state′′ = (*notifyWatches-loop literal Wl newWl stateB*) *in*
(*getM state′*) = (*getM state′′*) ∧
(*getF state′*) = (*getF state′′*) ∧
(*getSATFlag state′*) = (*getSATFlag state′′*) ∧
(*getConflictFlag state′*) = (*getConflictFlag state′′*)

using *assms*
proof (*induct Wl arbitrary*: *newWl stateA stateB*)
case *Nil*
thus *?case*
by *simp*
next
case (*Cons clause Wl′*)
from ‹∀ (*c*::*nat*). *c* ∈ *set* (*clause* # *Wl′*) ⟶ *0* ≤ *c* ∧ *c* < *length* (*getF stateA*)›
have *0* ≤ *clause* ∧ *clause* < *length* (*getF stateA*)
by *auto*
then obtain *wa*::*Literal* **and** *wb*::*Literal*
where *getWatch1 stateA clause* = *Some wa* **and** *getWatch2 stateA clause* = *Some wb*
using *Cons*
unfolding *InvariantWatchesEl-def*
by *auto*
show *?case*
proof (*cases Some literal* = *getWatch1 stateA clause*)
case *True*
hence *Some literal* = *getWatch1 stateB clause*
using ‹*getWatch1 stateA* = *getWatch1 stateB*›
by *simp*

let *?state′A* = *swapWatches clause stateA*
let *?state′B* = *swapWatches clause stateB*

have
getM ?state′A = *getM ?state′B*
getF ?state′A = *getF ?state′B*
getWatch1 ?state′A = *getWatch1 ?state′B*
getWatch2 ?state′A = *getWatch2 ?state′B*
getConflictFlag ?state′A = *getConflictFlag ?state′B*
getSATFlag ?state′A = *getSATFlag ?state′B*
using *Cons*

unfolding *swapWatches-def*
by *auto*

let *?w1* = *wb*
have *getWatch1 ?state'A clause* = *Some ?w1*
using ‹*getWatch2 stateA clause* = *Some wb*›
unfolding *swapWatches-def*
by *auto*
hence *getWatch1 ?state'B clause* = *Some ?w1*
using ‹*getWatch1 ?state'A* = *getWatch1 ?state'B*›
by *simp*
let *?w2* = *wa*
have *getWatch2 ?state'A clause* = *Some ?w2*
using ‹*getWatch1 stateA clause* = *Some wa*›
unfolding *swapWatches-def*
by *auto*
hence *getWatch2 ?state'B clause* = *Some ?w2*
using ‹*getWatch2 ?state'A* = *getWatch2 ?state'B*›
by *simp*

show *?thesis*
proof (*cases literalTrue ?w1* (*elements* (*getM ?state'A*)))
case *True*
hence *literalTrue ?w1* (*elements* (*getM ?state'B*))
using ‹*getM ?state'A* = *getM ?state'B*›
by *simp*

from *Cons*(*2*)
have *InvariantWatchesEl* (*getF ?state'A*) (*getWatch1 ?state'A*)
(*getWatch2 ?state'A*)
unfolding *InvariantWatchesEl-def*
unfolding *swapWatches-def*
by *auto*
moreover
have *getM ?state'A* = *getM stateA* ∧
getF ?state'A = *getF stateA* ∧
getSATFlag ?state'A = *getSATFlag stateA* ∧
getQ ?state'A = *getQ stateA*

unfolding *swapWatches-def*
by *simp*
moreover
have *getM ?state'B* = *getM stateB* ∧
getF ?state'B = *getF stateB* ∧
getSATFlag ?state'B = *getSATFlag stateB* ∧
getQ ?state'B = *getQ stateB*

unfolding *swapWatches-def*
by *simp*

```
      ultimately
      show ?thesis
        using Cons(1)[of ?state'A ?state'B clause # newWl]
        using ‹getM ?state'A = getM ?state'B›
        using ‹getF ?state'A = getF ?state'B›
        using ‹getWatch1 ?state'A = getWatch1 ?state'B›
        using ‹getWatch2 ?state'A = getWatch2 ?state'B›
        using ‹getConflictFlag ?state'A = getConflictFlag ?state'B›
        using ‹getSATFlag ?state'A = getSATFlag ?state'B›
        using Cons(3)
        using ‹getWatch1 ?state'A clause = Some ?w1›
        using ‹getWatch2 ?state'A clause = Some ?w2›
        using ‹getWatch1 ?state'B clause = Some ?w1›
        using ‹getWatch2 ?state'B clause = Some ?w2›
        using ‹Some literal = getWatch1 stateA clause›
        using ‹Some literal = getWatch1 stateB clause›
        using ‹literalTrue ?w1 (elements (getM ?state'A))›
        using ‹literalTrue ?w1 (elements (getM ?state'B))›
        by (simp add:Let-def)
    next
      case False
      hence ¬ literalTrue ?w1 (elements (getM ?state'B))
        using ‹getM ?state'A = getM ?state'B›
        by simp
      show ?thesis
    proof (cases getNonWatchedUnfalsifiedLiteral (nth (getF ?state'A)
clause) ?w1 ?w2 (getM ?state'A))
        case (Some l')
          hence getNonWatchedUnfalsifiedLiteral (nth (getF ?state'B)
clause) ?w1 ?w2 (getM ?state'B) = Some l'
          using ‹getF ?state'A = getF ?state'B›
          using ‹getM ?state'A = getM ?state'B›
          by simp

        have l' el (nth (getF ?state'A) clause)
          using Some
          using getNonWatchedUnfalsifiedLiteralSomeCharacterization
          by simp
        hence l' el (nth (getF ?state'B) clause)
          using ‹getF ?state'A = getF ?state'B›
          by simp


        let ?state''A = setWatch2 clause l' ?state'A
        let ?state''B = setWatch2 clause l' ?state'B

        have
          getM ?state''A = getM ?state''B
          getF ?state''A = getF ?state''B
```

getWatch1 ?state''A = getWatch1 ?state''B
getWatch2 ?state''A = getWatch2 ?state''B
getConflictFlag ?state''A = getConflictFlag ?state''B
getSATFlag ?state''A = getSATFlag ?state''B
using *Cons*
unfolding *setWatch2-def*
unfolding *swapWatches-def*
by *auto*

from *Cons(2)*
have *InvariantWatchesEl (getF ?state''A) (getWatch1 ?state''A) (getWatch2 ?state''A)*
using ‹*l' el (nth (getF ?state'A) clause)*›
unfolding *InvariantWatchesEl-def*
unfolding *swapWatches-def*
unfolding *setWatch2-def*
by *auto*
moreover
have *getM ?state''A = getM stateA* ∧
getF ?state''A = getF stateA ∧
getSATFlag ?state''A = getSATFlag stateA ∧
getQ ?state''A = getQ stateA
unfolding *swapWatches-def*
unfolding *setWatch2-def*
by *simp*
moreover
have *getM ?state''B = getM stateB* ∧
getF ?state''B = getF stateB ∧
getSATFlag ?state''B = getSATFlag stateB ∧
getQ ?state''B = getQ stateB

unfolding *swapWatches-def*
unfolding *setWatch2-def*
by *simp*
ultimately
show *?thesis*
using *Cons(1)[of ?state''A ?state''B newWl]*
using ‹*getM ?state''A = getM ?state''B*›
using ‹*getF ?state''A = getF ?state''B*›
using ‹*getWatch1 ?state''A = getWatch1 ?state''B*›
using ‹*getWatch2 ?state''A = getWatch2 ?state''B*›
using ‹*getConflictFlag ?state''A = getConflictFlag ?state''B*›
using ‹*getSATFlag ?state''A = getSATFlag ?state''B*›
using *Cons(3)*
using ‹*getWatch1 ?state'A clause = Some ?w1*›
using ‹*getWatch2 ?state'A clause = Some ?w2*›
using ‹*getWatch1 ?state'B clause = Some ?w1*›
using ‹*getWatch2 ?state'B clause = Some ?w2*›

```
        using ‹Some literal = getWatch1 stateA clause›
        using ‹Some literal = getWatch1 stateB clause›
        using ‹¬ literalTrue ?w1 (elements (getM ?state'A))›
        using ‹¬ literalTrue ?w1 (elements (getM ?state'B))›
        using ‹getNonWatchedUnfalsifiedLiteral (nth (getF ?state'A)
clause) ?w1 ?w2 (getM ?state'A) = Some l'›
        using ‹getNonWatchedUnfalsifiedLiteral (nth (getF ?state'B)
clause) ?w1 ?w2 (getM ?state'B) = Some l'›
        by (simp add:Let-def)
    next
      case None
        hence getNonWatchedUnfalsifiedLiteral (nth (getF ?state'B)
clause) ?w1 ?w2 (getM ?state'B) = None
       using ‹getF ?state'A = getF ?state'B› ‹getM ?state'A = getM
?state'B›
        by simp
      show ?thesis
      proof (cases literalFalse ?w1 (elements (getM ?state'A)))
        case True
        hence literalFalse ?w1 (elements (getM ?state'B))
          using ‹getM ?state'A = getM ?state'B›
          by simp

       let ?state''A = ?state'A(|getConflictFlag := True, getConflict-
Clause := clause|)
       let ?state''B = ?state'B(|getConflictFlag := True, getConflict-
Clause := clause|)
        have
          getM ?state''A = getM ?state''B
          getF ?state''A = getF ?state''B
          getWatch1 ?state''A = getWatch1 ?state''B
          getWatch2 ?state''A = getWatch2 ?state''B
          getConflictFlag ?state''A = getConflictFlag ?state''B
          getSATFlag ?state''A = getSATFlag ?state''B
          using Cons
          unfolding swapWatches-def
          by auto

        from Cons(2)
      have InvariantWatchesEl (getF ?state''A) (getWatch1 ?state''A)
(getWatch2 ?state''A)
          unfolding InvariantWatchesEl-def
          unfolding swapWatches-def
          by auto
        moreover
        have getM ?state''A = getM stateA ∧
        getF ?state''A = getF stateA ∧
        getSATFlag ?state''A = getSATFlag stateA ∧
          getQ ?state''A = getQ stateA
```

unfolding *swapWatches-def*
by *simp*
moreover
have *getM ?state''B = getM stateB* ∧
getF ?state''B = getF stateB ∧
getSATFlag ?state''B = getSATFlag stateB ∧
getQ ?state''B = getQ stateB
unfolding *swapWatches-def*
by *simp*
ultimately
show *?thesis*
using *Cons(4) Cons(5)*
using *Cons(1)[of ?state''A ?state''B clause # newWl]*
using ‹*getM ?state''A = getM ?state''B*›
using ‹*getF ?state''A = getF ?state''B*›
using ‹*getWatch1 ?state''A = getWatch1 ?state''B*›
using ‹*getWatch2 ?state''A = getWatch2 ?state''B*›
using ‹*getConflictFlag ?state''A = getConflictFlag ?state''B*›
using ‹*getSATFlag ?state''A = getSATFlag ?state''B*›
using *Cons(3)*
using ‹*getWatch1 ?state'A clause = Some ?w1*›
using ‹*getWatch2 ?state'A clause = Some ?w2*›
using ‹*getWatch1 ?state'B clause = Some ?w1*›
using ‹*getWatch2 ?state'B clause = Some ?w2*›
using ‹*Some literal = getWatch1 stateA clause*›
using ‹*Some literal = getWatch1 stateB clause*›
using ‹¬ *literalTrue ?w1 (elements (getM ?state'A))*›
using ‹¬ *literalTrue ?w1 (elements (getM ?state'B))*›
using ‹*getNonWatchedUnfalsifiedLiteral (nth (getF ?state'A) clause) ?w1 ?w2 (getM ?state'A) = None*›
using ‹*getNonWatchedUnfalsifiedLiteral (nth (getF ?state'B) clause) ?w1 ?w2 (getM ?state'B) = None*›
using ‹*literalFalse ?w1 (elements (getM ?state'A))*›
using ‹*literalFalse ?w1 (elements (getM ?state'B))*›
by *(simp add:Let-def)*
next
case *False*
hence ¬ *literalFalse ?w1 (elements (getM ?state'B))*
using ‹*getM ?state'A = getM ?state'B*›
by *simp*
let *?state''A = setReason ?w1 clause (?state'A(|getQ := (if ?w1 el (getQ ?state'A) then (getQ ?state'A) else (getQ ?state'A) @ [?w1])|))*
let *?state''B = setReason ?w1 clause (?state'B(|getQ := (if ?w1 el (getQ ?state'B) then (getQ ?state'B) else (getQ ?state'B) @ [?w1])|))*

have
getM ?state''A = getM ?state''B

getF ?state''A = getF ?state''B
getWatch1 ?state''A = getWatch1 ?state''B
getWatch2 ?state''A = getWatch2 ?state''B
getConflictFlag ?state''A = getConflictFlag ?state''B
getSATFlag ?state''A = getSATFlag ?state''B
using *Cons*
unfolding *setReason-def*
unfolding *swapWatches-def*
by *auto*

from *Cons*(*2*)
have *InvariantWatchesEl* (*getF ?state''A*) (*getWatch1 ?state''A*) (*getWatch2 ?state''A*)
unfolding *InvariantWatchesEl-def*
unfolding *swapWatches-def*
unfolding *setReason-def*
by *auto*
moreover
have *getM ?state''A = getM stateA* ∧
getF ?state''A = getF stateA ∧
getSATFlag ?state''A = getSATFlag stateA ∧
getQ ?state''A = (*if ?w1 el* (*getQ stateA*) *then* (*getQ stateA*) *else* (*getQ stateA*) @ [*?w1*])
unfolding *swapWatches-def*
unfolding *setReason-def*
by *auto*
moreover
have *getM ?state''B = getM stateB* ∧
getF ?state''B = getF stateB ∧
getSATFlag ?state''B = getSATFlag stateB ∧
getQ ?state''B = (*if ?w1 el* (*getQ stateB*) *then* (*getQ stateB*) *else* (*getQ stateB*) @ [*?w1*])
unfolding *swapWatches-def*
unfolding *setReason-def*
by *auto*
ultimately
show *?thesis*
using *Cons*(*4*) *Cons*(*5*)
using *Cons*(*1*)[*of ?state''A ?state''B clause # newWl*]
using ‹*getM ?state''A = getM ?state''B*›
using ‹*getF ?state''A = getF ?state''B*›
using ‹*getWatch1 ?state''A = getWatch1 ?state''B*›
using ‹*getWatch2 ?state''A = getWatch2 ?state''B*›
using ‹*getConflictFlag ?state''A = getConflictFlag ?state''B*›
using ‹*getSATFlag ?state''A = getSATFlag ?state''B*›
using *Cons*(*3*)
using ‹*getWatch1 ?state'A clause = Some ?w1*›
using ‹*getWatch2 ?state'A clause = Some ?w2*›
using ‹*getWatch1 ?state'B clause = Some ?w1*›

```
        using ‹getWatch2 ?state'B clause = Some ?w2›
        using ‹Some literal = getWatch1 stateA clause›
        using ‹Some literal = getWatch1 stateB clause›
        using ‹¬ literalTrue ?w1 (elements (getM ?state'A))›
        using ‹¬ literalTrue ?w1 (elements (getM ?state'B))›
       using ‹getNonWatchedUnfalsifiedLiteral (nth (getF ?state'A)
clause) ?w1 ?w2 (getM ?state'A) = None›
       using ‹getNonWatchedUnfalsifiedLiteral (nth (getF ?state'B)
clause) ?w1 ?w2 (getM ?state'B) = None›
        using ‹¬ literalFalse ?w1 (elements (getM ?state'A))›
        using ‹¬ literalFalse ?w1 (elements (getM ?state'B))›
        by (simp add:Let-def)
      qed
     qed
    qed
  next
    case False
    hence Some literal ≠ getWatch1 stateB clause
      using Cons
      by simp

    let ?state'A = stateA
    let ?state'B = stateB

    have
      getM ?state'A = getM ?state'B
      getF ?state'A = getF ?state'B
      getWatch1 ?state'A = getWatch1 ?state'B
      getWatch2 ?state'A = getWatch2 ?state'B
      getConflictFlag ?state'A = getConflictFlag ?state'B
      getSATFlag ?state'A = getSATFlag ?state'B
      using Cons
      by auto

    let ?w1 = wa
    have getWatch1 ?state'A clause = Some ?w1
      using ‹getWatch1 stateA clause = Some wa›
      by auto
    hence getWatch1 ?state'B clause = Some ?w1
      using Cons
      by simp
    let ?w2 = wb
    have getWatch2 ?state'A clause = Some ?w2
      using ‹getWatch2 stateA clause = Some wb›
      by auto
    hence getWatch2 ?state'B clause = Some ?w2
      using Cons
      by simp
```

```
    show ?thesis
    proof (cases literalTrue ?w1 (elements (getM ?state'A)))
      case True
      hence literalTrue ?w1 (elements (getM ?state'B))
        using Cons
        by simp

      show ?thesis
        using Cons(1)[of ?state'A ?state'B clause # newWl]
         using Cons(2) Cons(3) Cons(4) Cons(5) Cons(6) Cons(7)
Cons(8) Cons(9)
        using ‹¬ Some literal = getWatch1 stateA clause›
        using ‹¬ Some literal = getWatch1 stateB clause›
        using ‹getWatch1 ?state'A clause = Some ?w1›
        using ‹getWatch1 ?state'B clause = Some ?w1›
        using ‹getWatch2 ?state'A clause = Some ?w2›
        using ‹getWatch2 ?state'B clause = Some ?w2›
        using ‹literalTrue ?w1 (elements (getM ?state'A))›
        using ‹literalTrue ?w1 (elements (getM ?state'B))›
        by (simp add:Let-def)
    next
      case False
      hence ¬ literalTrue ?w1 (elements (getM ?state'B))
        using ‹getM ?state'A = getM ?state'B›
        by simp
      show ?thesis
    proof (cases getNonWatchedUnfalsifiedLiteral (nth (getF ?state'A)
clause) ?w1 ?w2 (getM ?state'A))
        case (Some l')
         hence getNonWatchedUnfalsifiedLiteral (nth (getF ?state'B)
clause) ?w1 ?w2 (getM ?state'B) = Some l'
          using ‹getF ?state'A = getF ?state'B›
          using ‹getM ?state'A = getM ?state'B›
          by simp

        have l' el (nth (getF ?state'A) clause)
          using Some
          using getNonWatchedUnfalsifiedLiteralSomeCharacterization
          by simp
        hence l' el (nth (getF ?state'B) clause)
          using ‹getF ?state'A = getF ?state'B›
          by simp

        let ?state''A = setWatch2 clause l' ?state'A
        let ?state''B = setWatch2 clause l' ?state'B

        have
          getM ?state''A = getM ?state''B
          getF ?state''A = getF ?state''B
```

getWatch1 ?state''A = getWatch1 ?state''B
getWatch2 ?state''A = getWatch2 ?state''B
getConflictFlag ?state''A = getConflictFlag ?state''B
getSATFlag ?state''A = getSATFlag ?state''B
using *Cons*
unfolding *setWatch2-def*
by *auto*

from *Cons*(*2*)
have *InvariantWatchesEl* (*getF ?state''A*) (*getWatch1 ?state''A*) (*getWatch2 ?state''A*)
using ‹*l' el* (*nth* (*getF ?state'A*) *clause*)›
unfolding *InvariantWatchesEl-def*
unfolding *setWatch2-def*
by *auto*
moreover
have *getM ?state''A = getM stateA* ∧
getF ?state''A = getF stateA ∧
getSATFlag ?state''A = getSATFlag stateA ∧
getQ ?state''A = getQ stateA
unfolding *setWatch2-def*
by *simp*
ultimately
show *?thesis*
using *Cons*(*1*)[*of ?state''A ?state''B newWl*]
using ‹*getM ?state''A = getM ?state''B*›
using ‹*getF ?state''A = getF ?state''B*›
using ‹*getWatch1 ?state''A = getWatch1 ?state''B*›
using ‹*getWatch2 ?state''A = getWatch2 ?state''B*›
using ‹*getConflictFlag ?state''A = getConflictFlag ?state''B*›
using ‹*getSATFlag ?state''A = getSATFlag ?state''B*›
using *Cons*(*3*)
using ‹*getWatch1 ?state'A clause = Some ?w1*›
using ‹*getWatch2 ?state'A clause = Some ?w2*›
using ‹*getWatch1 ?state'B clause = Some ?w1*›
using ‹*getWatch2 ?state'B clause = Some ?w2*›
using ‹¬ *Some literal = getWatch1 stateA clause*›
using ‹¬ *Some literal = getWatch1 stateB clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state'A*))›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state'B*))›
using ‹*getNonWatchedUnfalsifiedLiteral* (*nth* (*getF ?state'A*) *clause*) *?w1 ?w2* (*getM ?state'A*) = *Some l'*›
using ‹*getNonWatchedUnfalsifiedLiteral* (*nth* (*getF ?state'B*) *clause*) *?w1 ?w2* (*getM ?state'B*) = *Some l'*›
by (*simp add:Let-def*)
next
case *None*
hence *getNonWatchedUnfalsifiedLiteral* (*nth* (*getF ?state'B*)

clause) *?w1 ?w2* (*getM ?state'B*) = *None*
using ‹*getF ?state'A* = *getF ?state'B*› ‹*getM ?state'A* = *getM ?state'B*›
by *simp*
show *?thesis*
proof (*cases literalFalse ?w1* (*elements* (*getM ?state'A*)))
case *True*
hence *literalFalse ?w1* (*elements* (*getM ?state'B*))
using ‹*getM ?state'A* = *getM ?state'B*›
by *simp*

let *?state''A* = *?state'A*(|*getConflictFlag* := *True*, *getConflictClause* := *clause*|)
let *?state''B* = *?state'B*(|*getConflictFlag* := *True*, *getConflictClause* := *clause*|)
have
getM ?state''A = *getM ?state''B*
getF ?state''A = *getF ?state''B*
getWatch1 ?state''A = *getWatch1 ?state''B*
getWatch2 ?state''A = *getWatch2 ?state''B*
getConflictFlag ?state''A = *getConflictFlag ?state''B*
getSATFlag ?state''A = *getSATFlag ?state''B*
using *Cons*
by *auto*

from *Cons*(*2*)
have *InvariantWatchesEl* (*getF ?state''A*) (*getWatch1 ?state''A*) (*getWatch2 ?state''A*)
unfolding *InvariantWatchesEl-def*
by *auto*
moreover
have *getM ?state''A* = *getM stateA* ∧
getF ?state''A = *getF stateA* ∧
getSATFlag ?state''A = *getSATFlag stateA* ∧
getQ ?state''A = *getQ stateA*
by *simp*
ultimately
show *?thesis*
using *Cons*(*4*) *Cons*(*5*)
using *Cons*(*1*)[*of ?state''A ?state''B clause # newWl*]
using ‹*getM ?state''A* = *getM ?state''B*›
using ‹*getF ?state''A* = *getF ?state''B*›
using ‹*getWatch1 ?state''A* = *getWatch1 ?state''B*›
using ‹*getWatch2 ?state''A* = *getWatch2 ?state''B*›
using ‹*getConflictFlag ?state''A* = *getConflictFlag ?state''B*›
using ‹*getSATFlag ?state''A* = *getSATFlag ?state''B*›
using *Cons*(*3*)
using ‹*getWatch1 ?state'A clause* = *Some ?w1*›
using ‹*getWatch2 ?state'A clause* = *Some ?w2*›

using ‹*getWatch1 ?state'B clause = Some ?w1*›
using ‹*getWatch2 ?state'B clause = Some ?w2*›
using ‹¬ *Some literal = getWatch1 stateA clause*›
using ‹¬ *Some literal = getWatch1 stateB clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state'A*))›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state'B*))›
using ‹*getNonWatchedUnfalsifiedLiteral* (*nth* (*getF ?state'A*) *clause*) *?w1 ?w2* (*getM ?state'A*) = *None*›
using ‹*getNonWatchedUnfalsifiedLiteral* (*nth* (*getF ?state'B*) *clause*) *?w1 ?w2* (*getM ?state'B*) = *None*›
using ‹*literalFalse ?w1* (*elements* (*getM ?state'A*))›
using ‹*literalFalse ?w1* (*elements* (*getM ?state'B*))›
by (*simp add*:*Let-def*)
next
case *False*
hence ¬ *literalFalse ?w1* (*elements* (*getM ?state'B*))
using ‹*getM ?state'A = getM ?state'B*›
by *simp*
let *?state''A = setReason ?w1 clause* (*?state'A*⦇*getQ* := (*if ?w1 el* (*getQ ?state'A*) *then* (*getQ ?state'A*) *else* (*getQ ?state'A*) @ [*?w1*])⦈)
let *?state''B = setReason ?w1 clause* (*?state'B*⦇*getQ* := (*if ?w1 el* (*getQ ?state'B*) *then* (*getQ ?state'B*) *else* (*getQ ?state'B*) @ [*?w1*])⦈)

have
getM ?state''A = getM ?state''B
getF ?state''A = getF ?state''B
getWatch1 ?state''A = getWatch1 ?state''B
getWatch2 ?state''A = getWatch2 ?state''B
getConflictFlag ?state''A = getConflictFlag ?state''B
getSATFlag ?state''A = getSATFlag ?state''B
using *Cons*
unfolding *setReason-def*
by *auto*

from *Cons*(*2*)
have *InvariantWatchesEl* (*getF ?state''A*) (*getWatch1 ?state''A*) (*getWatch2 ?state''A*)
unfolding *InvariantWatchesEl-def*
unfolding *setReason-def*
by *auto*
moreover
have *getM ?state''A = getM stateA* ∧
getF ?state''A = getF stateA ∧
getSATFlag ?state''A = getSATFlag stateA ∧
getQ ?state''A = (*if ?w1 el* (*getQ stateA*) *then* (*getQ stateA*) *else* (*getQ stateA*) @ [*?w1*])
unfolding *setReason-def*

```
        by auto
      ultimately
      show ?thesis
        using Cons(4) Cons(5)
        using Cons(1)[of ?state''A ?state''B clause # newWl]
        using ‹getM ?state''A = getM ?state''B›
        using ‹getF ?state''A = getF ?state''B›
        using ‹getWatch1 ?state''A = getWatch1 ?state''B›
        using ‹getWatch2 ?state''A = getWatch2 ?state''B›
        using ‹getConflictFlag ?state''A = getConflictFlag ?state''B›
        using ‹getSATFlag ?state''A = getSATFlag ?state''B›
        using Cons(3)
        using ‹getWatch1 ?state'A clause = Some ?w1›
        using ‹getWatch2 ?state'A clause = Some ?w2›
        using ‹getWatch1 ?state'B clause = Some ?w1›
        using ‹getWatch2 ?state'B clause = Some ?w2›
        using ‹¬ Some literal = getWatch1 stateA clause›
        using ‹¬ Some literal = getWatch1 stateB clause›
        using ‹¬ literalTrue ?w1 (elements (getM ?state'A))›
        using ‹¬ literalTrue ?w1 (elements (getM ?state'B))›
        using ‹getNonWatchedUnfalsifiedLiteral (nth (getF ?state'A)
clause) ?w1 ?w2 (getM ?state'A) = None›
        using ‹getNonWatchedUnfalsifiedLiteral (nth (getF ?state'B)
clause) ?w1 ?w2 (getM ?state'B) = None›
        using ‹¬ literalFalse ?w1 (elements (getM ?state'A))›
        using ‹¬ literalFalse ?w1 (elements (getM ?state'B))›
        by (simp add:Let-def)
    qed
   qed
  qed
 qed
qed

lemma notifyWatchesLoopPreservedWatches:
fixes literal :: Literal and Wl :: nat list and newWl :: nat list and
state :: State
assumes
 InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
and
 ∀ (c::nat). c ∈ set Wl ⟶ 0 ≤ c ∧ c < length (getF state)
shows
 let state' = (notifyWatches-loop literal Wl newWl state) in
   ∀ c. c ∉ set Wl ⟶ (getWatch1 state' c) = (getWatch1 state c) ∧
(getWatch2 state' c) = (getWatch2 state c)

using assms
proof (induct Wl arbitrary: newWl state)
 case Nil
 thus ?case
```

```
    by simp
next
  case (Cons clause Wl')
  from ‹∀ (c::nat). c ∈ set (clause # Wl') ⟶ 0 ≤ c ∧ c < length
(getF state)›
  have 0 ≤ clause ∧ clause < length (getF state)
    by auto
  then obtain wa::Literal and wb::Literal
     where getWatch1 state clause = Some wa and getWatch2 state
clause = Some wb
    using Cons
    unfolding InvariantWatchesEl-def
    by auto
  show ?case
  proof (cases Some literal = getWatch1 state clause)
    case True
    let ?state' = swapWatches clause state
    let ?w1 = wb
    have getWatch1 ?state' clause = Some ?w1
      using ‹getWatch2 state clause = Some wb›
      unfolding swapWatches-def
      by auto
    let ?w2 = wa
    have getWatch2 ?state' clause = Some ?w2
      using ‹getWatch1 state clause = Some wa›
      unfolding swapWatches-def
      by auto
    show ?thesis
    proof (cases literalTrue ?w1 (elements (getM ?state')))
      case True

      from Cons(2)
         have InvariantWatchesEl (getF ?state') (getWatch1 ?state')
(getWatch2 ?state')
        unfolding InvariantWatchesEl-def
        unfolding swapWatches-def
        by auto
      moreover
      have getM ?state' = getM state ∧
        getF ?state' = getF state
        unfolding swapWatches-def
        by simp
      ultimately
      show ?thesis
        using Cons(1)[of ?state' clause # newWl]
        using Cons(3)
        using ‹getWatch1 ?state' clause = Some ?w1›
        using ‹getWatch2 ?state' clause = Some ?w2›
        using ‹Some literal = getWatch1 state clause›
```

```
      using ‹literalTrue ?w1 (elements (getM ?state'))›
      apply (simp add:Let-def)
      unfolding swapWatches-def
      by simp
  next
    case False
    show ?thesis
    proof (cases getNonWatchedUnfalsifiedLiteral (nth (getF ?state')
clause) ?w1 ?w2 (getM ?state'))
      case (Some l')
      hence l' el (nth (getF ?state') clause)
        using getNonWatchedUnfalsifiedLiteralSomeCharacterization
        by simp

      let ?state'' = setWatch2 clause l' ?state'

      from Cons(2)
      have InvariantWatchesEl (getF ?state'') (getWatch1 ?state'')
(getWatch2 ?state'')
        using ‹l' el (nth (getF ?state') clause)›
        unfolding InvariantWatchesEl-def
        unfolding swapWatches-def
        unfolding setWatch2-def
        by auto
      moreover
      have getM ?state'' = getM state ∧
        getF ?state'' = getF state
        unfolding swapWatches-def
        unfolding setWatch2-def
        by simp
      ultimately
      show ?thesis
        using Cons(1)[of ?state'' newWl]
        using Cons(3)
        using ‹getWatch1 ?state' clause = Some ?w1›
        using ‹getWatch2 ?state' clause = Some ?w2›
        using ‹Some literal = getWatch1 state clause›
        using ‹¬ literalTrue ?w1 (elements (getM ?state'))›
        using Some
        apply (simp add: Let-def)
        unfolding setWatch2-def
        unfolding swapWatches-def
        by simp
    next
      case None
      show ?thesis
      proof (cases literalFalse ?w1 (elements (getM ?state')))
        case True
      let ?state'' = ?state'(|getConflictFlag := True, getConflictClause
```

:= *clause*⦈

from *Cons*(*2*)
have *InvariantWatchesEl* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)
unfolding *InvariantWatchesEl-def*
unfolding *swapWatches-def*
by *auto*
moreover
have *getM ?state''* = *getM state* ∧
getF ?state'' = *getF state*
unfolding *swapWatches-def*
by *simp*
ultimately
show *?thesis*
using *Cons*(*1*)[*of ?state'' clause # newWl*]
using *Cons*(*3*)
using ‹*getWatch1 ?state' clause* = *Some ?w1*›
using ‹*getWatch2 ?state' clause* = *Some ?w2*›
using ‹*Some literal* = *getWatch1 state clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state'*))›
using *None*
using ‹*literalFalse ?w1* (*elements* (*getM ?state'*))›
apply (*simp add*: *Let-def*)
unfolding *swapWatches-def*
by *simp*
next
case *False*
let *?state''* = *setReason ?w1 clause* (*?state'*⦇*getQ* := (*if ?w1 el* (*getQ ?state'*) *then* (*getQ ?state'*) *else* (*getQ ?state'*) @ [*?w1*])⦈)

from *Cons*(*2*)
have *InvariantWatchesEl* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)
unfolding *InvariantWatchesEl-def*
unfolding *swapWatches-def*
unfolding *setReason-def*
by *auto*
moreover
have *getM ?state''* = *getM state* ∧
getF ?state'' = *getF state*
unfolding *swapWatches-def*
unfolding *setReason-def*
by *simp*
ultimately
show *?thesis*
using *Cons*(*1*)[*of ?state'' clause # newWl*]
using *Cons*(*3*)
using ‹*getWatch1 ?state' clause* = *Some ?w1*›

```
        using ‹getWatch2 ?state' clause = Some ?w2›
        using ‹Some literal = getWatch1 state clause›
        using ‹¬ literalTrue ?w1 (elements (getM ?state'))›
        using None
        using ‹¬ literalFalse ?w1 (elements (getM ?state'))›
        apply (simp add: Let-def)
        unfolding setReason-def
        unfolding swapWatches-def
        by simp
      qed
    qed
  qed
next
  case False
  let ?state' = state
  let ?w1 = wa
  have getWatch1 ?state' clause = Some ?w1
    using ‹getWatch1 state clause = Some wa›
    unfolding swapWatches-def
    by auto
  let ?w2 = wb
  have getWatch2 ?state' clause = Some ?w2
    using ‹getWatch2 state clause = Some wb›
    unfolding swapWatches-def
    by auto
  show ?thesis
  proof (cases literalTrue ?w1 (elements (getM ?state')))
    case True
    thus ?thesis
      using Cons
      using ‹¬ Some literal = getWatch1 state clause›
      using ‹getWatch1 ?state' clause = Some ?w1›
      using ‹getWatch2 ?state' clause = Some ?w2›
      using ‹literalTrue ?w1 (elements (getM ?state'))›
      by (simp add:Let-def)
  next
    case False
    show ?thesis
    proof (cases getNonWatchedUnfalsifiedLiteral (nth (getF ?state')
clause) ?w1 ?w2 (getM ?state'))
      case (Some l')
      hence l' el (nth (getF ?state')) clause
        using getNonWatchedUnfalsifiedLiteralSomeCharacterization
        by simp

      let ?state'' = setWatch2 clause l' ?state'

      from Cons(2)
      have InvariantWatchesEl (getF ?state'') (getWatch1 ?state'')
```

(*getWatch2 ?state''*)
  using ‹*l' el* (*nth* (*getF ?state'*)) *clause*›
  unfolding *InvariantWatchesEl-def*
  unfolding *setWatch2-def*
  by *auto*
 moreover
 have *getM ?state''* = *getM state* ∧
  getF ?state'' = *getF state*
  unfolding *setWatch2-def*
  by *simp*
 ultimately
 show *?thesis*
  using *Cons*(*1*)[*of ?state''*]
  using *Cons*(*3*)
  using ‹*getWatch1 ?state' clause* = *Some ?w1*›
  using ‹*getWatch2 ?state' clause* = *Some ?w2*›
  using ‹¬ *Some literal* = *getWatch1 state clause*›
  using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state'*))›
  using *Some*
  apply (*simp add*: *Let-def*)
  unfolding *setWatch2-def*
  by *simp*
next
 case *None*
 show *?thesis*
 proof (*cases literalFalse ?w1* (*elements* (*getM ?state'*)))
  case *True*
 let *?state''* = *?state'*(|*getConflictFlag* := *True*, *getConflictClause*
:= *clause*|)

  from *Cons*(*2*)
  have *InvariantWatchesEl* (*getF ?state''*) (*getWatch1 ?state''*)
(*getWatch2 ?state''*)
   unfolding *InvariantWatchesEl-def*
   by *auto*
  moreover
  have *getM ?state''* = *getM state* ∧
   getF ?state'' = *getF state*
   by *simp*
  ultimately
  show *?thesis*
   using *Cons*(*1*)[*of ?state''*]
   using *Cons*(*3*)
   using ‹*getWatch1 ?state' clause* = *Some ?w1*›
   using ‹*getWatch2 ?state' clause* = *Some ?w2*›
   using ‹¬ *Some literal* = *getWatch1 state clause*›
   using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state'*))›
   using *None*
   using ‹*literalFalse ?w1* (*elements* (*getM ?state'*))›

```
          by (simp add: Let-def)
        next
          case False
          let ?state'' = setReason ?w1 clause (?state'⦇getQ := (if ?w1
el (getQ ?state') then (getQ ?state') else (getQ ?state') @ [?w1])⦈)
          from Cons(2)
          have InvariantWatchesEl (getF ?state'') (getWatch1 ?state'')
(getWatch2 ?state'')
            unfolding InvariantWatchesEl-def
            unfolding setReason-def
            by auto
          moreover
          have getM ?state'' = getM state ∧
            getF ?state'' = getF state
            unfolding setReason-def
            by simp
          ultimately
          show ?thesis
            using Cons(1)[of ?state'']
            using Cons(3)
            using ‹getWatch1 ?state' clause = Some ?w1›
            using ‹getWatch2 ?state' clause = Some ?w2›
            using ‹¬ Some literal = getWatch1 state clause›
            using ‹¬ literalTrue ?w1 (elements (getM ?state'))›
            using None
            using ‹¬ literalFalse ?w1 (elements (getM ?state'))›
            apply (simp add: Let-def)
            unfolding setReason-def
            by simp
        qed
      qed
    qed
  qed
qed

lemma InvariantWatchesElNotifyWatchesLoop:
fixes literal :: Literal and Wl :: nat list and newWl :: nat list and
state :: State
assumes
  InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
and
  ∀ (c::nat). c ∈ set Wl ⟶ 0 ≤ c ∧ c < length (getF state)
shows
  let state' = (notifyWatches-loop literal Wl newWl state) in
      InvariantWatchesEl (getF state') (getWatch1 state') (getWatch2
state')
using assms
proof (induct Wl arbitrary: newWl state)
  case Nil
```

thus *?case*
by *simp*
next
case (*Cons clause Wl'*)
from ‹∀ (*c*::*nat*). *c* ∈ *set* (*clause* # *Wl'*) ⟶ *0* ≤ *c* ∧ *c* < *length* (*getF state*)›
have *0* ≤ *clause* **and** *clause* < *length* (*getF state*)
by *auto*
then obtain *wa*::*Literal* **and** *wb*::*Literal*
where *getWatch1 state clause* = *Some wa* **and** *getWatch2 state clause* = *Some wb*
using *Cons*
unfolding *InvariantWatchesEl-def*
by *auto*
show *?case*
proof (*cases Some literal* = *getWatch1 state clause*)
case *True*
let *?state'* = *swapWatches clause state*
let *?w1* = *wb*
have *getWatch1 ?state' clause* = *Some ?w1*
using ‹*getWatch2 state clause* = *Some wb*›
unfolding *swapWatches-def*
by *auto*
let *?w2* = *wa*
have *getWatch2 ?state' clause* = *Some ?w2*
using ‹*getWatch1 state clause* = *Some wa*›
unfolding *swapWatches-def*
by *auto*
show *?thesis*
proof (*cases literalTrue ?w1* (*elements* (*getM ?state'*)))
case *True*

from *Cons*(*2*)
have *InvariantWatchesEl* (*getF ?state'*) (*getWatch1 ?state'*) (*getWatch2 ?state'*)
unfolding *InvariantWatchesEl-def*
unfolding *swapWatches-def*
by *auto*
moreover
have *getF ?state'* = *getF state*
unfolding *swapWatches-def*
by *simp*
ultimately
show *?thesis*
using *Cons*
using ‹*Some literal* = *getWatch1 state clause*›
using ‹*getWatch1 ?state' clause* = *Some ?w1*›
using ‹*getWatch2 ?state' clause* = *Some ?w2*›
using ‹*literalTrue ?w1* (*elements* (*getM ?state'*))›

```
        by (simp add: Let-def)
    next
      case False
      show ?thesis
     proof (cases getNonWatchedUnfalsifiedLiteral (nth (getF ?state′)
clause) ?w1 ?w2 (getM ?state′))
        case (Some l′)
        hence l′ el (nth (getF ?state′) clause)
          using getNonWatchedUnfalsifiedLiteralSomeCharacterization
          by simp

        let ?state′′ = setWatch2 clause l′ ?state′

        from Cons(2)
         have InvariantWatchesEl (getF ?state′′) (getWatch1 ?state′′)
(getWatch2 ?state′′)
          using ‹l′ el (nth (getF ?state′) clause)›
          unfolding InvariantWatchesEl-def
          unfolding swapWatches-def
          unfolding setWatch2-def
          by auto
        moreover
        have getF ?state′′ = getF state
          unfolding swapWatches-def
          unfolding setWatch2-def
          by simp
        ultimately
        show ?thesis
          using Cons
          using ‹getWatch1 ?state′ clause = Some ?w1›
          using ‹getWatch2 ?state′ clause = Some ?w2›
          using ‹Some literal = getWatch1 state clause›
          using ‹¬ literalTrue ?w1 (elements (getM ?state′))›
          using Some
          by (simp add: Let-def)
      next
        case None
        show ?thesis
        proof (cases literalFalse ?w1 (elements (getM ?state′)))
          case True
       let ?state′′ = ?state′⦇getConflictFlag := True, getConflictClause
:= clause⦈

          from Cons(2)
          have InvariantWatchesEl (getF ?state′′) (getWatch1 ?state′′)
(getWatch2 ?state′′)
            unfolding InvariantWatchesEl-def
            unfolding swapWatches-def
            by auto
```

```
        moreover
        have getF ?state'' = getF state
          unfolding swapWatches-def
          by simp
        ultimately
        show ?thesis
          using Cons
          using ‹getWatch1 ?state' clause = Some ?w1›
          using ‹getWatch2 ?state' clause = Some ?w2›
          using ‹Some literal = getWatch1 state clause›
          using ‹¬ literalTrue ?w1 (elements (getM ?state'))›
          using None
          using ‹literalFalse ?w1 (elements (getM ?state'))›
          by (simp add: Let-def)
      next
        case False
        let ?state'' = setReason ?w1 clause (?state'⦇getQ := (if ?w1
el (getQ ?state') then (getQ ?state') else (getQ ?state') @ [?w1])⦈)

        from Cons(2)
        have InvariantWatchesEl (getF ?state'') (getWatch1 ?state'')
(getWatch2 ?state'')
          unfolding InvariantWatchesEl-def
          unfolding swapWatches-def
          unfolding setReason-def
          by auto
        moreover
        have getF ?state'' = getF state
          unfolding swapWatches-def
          unfolding setReason-def
          by simp
        ultimately
        show ?thesis
          using Cons
          using ‹getWatch1 ?state' clause = Some ?w1›
          using ‹getWatch2 ?state' clause = Some ?w2›
          using ‹Some literal = getWatch1 state clause›
          using ‹¬ literalTrue ?w1 (elements (getM ?state'))›
          using None
          using ‹¬ literalFalse ?w1 (elements (getM ?state'))›
          by (simp add: Let-def)
      qed
    qed
  qed
next
  case False
  let ?state' = state
  let ?w1 = wa
  have getWatch1 ?state' clause = Some ?w1
```

```
      using ‹getWatch1 state clause = Some wa›
      unfolding swapWatches-def
      by auto
    let ?w2 = wb
    have getWatch2 ?state' clause = Some ?w2
      using ‹getWatch2 state clause = Some wb›
      unfolding swapWatches-def
      by auto
    show ?thesis
    proof (cases literalTrue ?w1 (elements (getM ?state')))
      case True
      thus ?thesis
        using Cons
        using ‹¬ Some literal = getWatch1 state clause›
        using ‹getWatch1 ?state' clause = Some ?w1›
        using ‹getWatch2 ?state' clause = Some ?w2›
        using ‹literalTrue ?w1 (elements (getM ?state'))›
        by (simp add:Let-def)
    next
      case False
      show ?thesis
     proof (cases getNonWatchedUnfalsifiedLiteral (nth (getF ?state')
clause) ?w1 ?w2 (getM ?state'))
        case (Some l')
        hence l' el (nth (getF ?state') clause)
          using getNonWatchedUnfalsifiedLiteralSomeCharacterization
          by simp

        let ?state'' = setWatch2 clause l' ?state'

        from Cons
         have InvariantWatchesEl (getF ?state'') (getWatch1 ?state'')
(getWatch2 ?state'')
          using ‹l' el (nth (getF ?state') clause)›
          unfolding InvariantWatchesEl-def
          unfolding setWatch2-def
          by auto
        moreover
        have getF ?state'' = getF state
          unfolding setWatch2-def
          by simp
        ultimately
        show ?thesis
          using Cons
          using ‹getWatch1 ?state' clause = Some ?w1›
          using ‹getWatch2 ?state' clause = Some ?w2›
          using ‹¬ Some literal = getWatch1 state clause›
          using ‹¬ literalTrue ?w1 (elements (getM ?state'))›
          using Some
```

by (*simp add*: *Let-def*)
next
case *None*
show *?thesis*
proof (*cases literalFalse ?w1* (*elements* (*getM ?state'*)))
case *True*
let *?state''* = *?state'*⦇*getConflictFlag* := *True*, *getConflictClause* := *clause*⦈

from *Cons*
have *InvariantWatchesEl* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)
unfolding *InvariantWatchesEl-def*
by *auto*
moreover
have *getF ?state''* = *getF state*
by *simp*
ultimately
show *?thesis*
using *Cons*
using ‹*getWatch1 ?state' clause* = *Some ?w1*›
using ‹*getWatch2 ?state' clause* = *Some ?w2*›
using ‹¬ *Some literal* = *getWatch1 state clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state'*))›
using *None*
using ‹*literalFalse ?w1* (*elements* (*getM ?state'*))›
by (*simp add*: *Let-def*)
next
case *False*
let *?state''* = *setReason ?w1 clause* (*?state'*⦇*getQ* := (*if ?w1 el* (*getQ ?state'*) *then* (*getQ ?state'*) *else* (*getQ ?state'*) @ [*?w1*])⦈)
from *Cons*(*2*)
have *InvariantWatchesEl* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)
unfolding *InvariantWatchesEl-def*
unfolding *setReason-def*
by *auto*
moreover
have *getF ?state''* = *getF state*
unfolding *setReason-def*
by *simp*
ultimately
show *?thesis*
using *Cons*
using ‹*getWatch1 ?state' clause* = *Some ?w1*›
using ‹*getWatch2 ?state' clause* = *Some ?w2*›
using ‹¬ *Some literal* = *getWatch1 state clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state'*))›
using *None*

using ‹¬ *literalFalse ?w1* (*elements* (*getM ?state'*))›
by (*simp add*: *Let-def*)
qed
qed
qed
qed
qed

lemma *InvariantWatchesDifferNotifyWatchesLoop*:
fixes *literal* :: *Literal* **and** *Wl* :: *nat list* **and** *newWl* :: *nat list* **and** *state* :: *State*
assumes
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
and
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
∀ (*c*::*nat*). *c* ∈ *set Wl* ⟶ *0* ≤ *c* ∧ *c* < *length* (*getF state*)
shows
let state' = (*notifyWatches-loop literal Wl newWl state*) *in*
InvariantWatchesDiffer (*getF state'*) (*getWatch1 state'*) (*getWatch2 state'*)
using *assms*
proof (*induct Wl arbitrary*: *newWl state*)
case *Nil*
thus *?case*
by *simp*
next
case (*Cons clause Wl'*)
from ‹∀ (*c*::*nat*). *c* ∈ *set* (*clause* # *Wl'*) ⟶ *0* ≤ *c* ∧ *c* < *length* (*getF state*)›
have *0* ≤ *clause* **and** *clause* < *length* (*getF state*)
by *auto*
then obtain *wa*::*Literal* **and** *wb*::*Literal*
where *getWatch1 state clause* = *Some wa* **and** *getWatch2 state clause* = *Some wb*
using *Cons*
unfolding *InvariantWatchesEl-def*
by *auto*
show *?case*
proof (*cases Some literal* = *getWatch1 state clause*)
case *True*
let *?state'* = *swapWatches clause state*
let *?w1* = *wb*
have *getWatch1 ?state' clause* = *Some ?w1*
using ‹*getWatch2 state clause* = *Some wb*›
unfolding *swapWatches-def*
by *auto*
let *?w2* = *wa*
have *getWatch2 ?state' clause* = *Some ?w2*

```
      using ‹getWatch1 state clause = Some wa›
      unfolding swapWatches-def
      by auto
    show ?thesis
    proof (cases literalTrue ?w1 (elements (getM ?state')))
      case True

      from Cons(2)
         have InvariantWatchesEl (getF ?state') (getWatch1 ?state')
(getWatch2 ?state')
        unfolding InvariantWatchesEl-def
        unfolding swapWatches-def
        by auto
      moreover
      from Cons(3)
       have InvariantWatchesDiffer (getF ?state') (getWatch1 ?state')
(getWatch2 ?state')
        unfolding InvariantWatchesDiffer-def
        unfolding swapWatches-def
        by auto
      moreover
      have getF ?state' = getF state
        unfolding swapWatches-def
        by simp
      ultimately
      show ?thesis
        using Cons(1)[of ?state' clause # newWl]
        using Cons(4)
        using ‹Some literal = getWatch1 state clause›
        using ‹getWatch1 ?state' clause = Some ?w1›
        using ‹getWatch2 ?state' clause = Some ?w2›
        using ‹literalTrue ?w1 (elements (getM ?state'))›
        by (simp add: Let-def)
    next
      case False
      show ?thesis
       proof (cases getNonWatchedUnfalsifiedLiteral (nth (getF ?state')
clause) ?w1 ?w2 (getM ?state'))
        case (Some l')
        hence l' el (nth (getF ?state') clause) l' ≠ literal l' ≠ ?w1 l' ≠
?w2
          using getNonWatchedUnfalsifiedLiteralSomeCharacterization
          using ‹getWatch1 ?state' clause = Some ?w1›
          using ‹getWatch2 ?state' clause = Some ?w2›
          using ‹Some literal = getWatch1 state clause›
          unfolding swapWatches-def
          by auto

        let ?state'' = setWatch2 clause l' ?state'
```

```
        from Cons(2)
         have InvariantWatchesEl (getF ?state'') (getWatch1 ?state'')
(getWatch2 ?state'')
          using ‹l' el (nth (getF ?state') clause)›
          unfolding InvariantWatchesEl-def
          unfolding swapWatches-def
          unfolding setWatch2-def
          by auto
        moreover
        from Cons(3)
      have InvariantWatchesDiffer (getF ?state'') (getWatch1 ?state'')
(getWatch2 ?state'')
          using ‹l' ≠ ?w1›
          using ‹getWatch1 ?state' clause = Some ?w1›
          using ‹getWatch2 ?state' clause = Some ?w2›
          unfolding InvariantWatchesDiffer-def
          unfolding swapWatches-def
          unfolding setWatch2-def
          by auto
        moreover
        have getF ?state'' = getF state
          unfolding swapWatches-def
          unfolding setWatch2-def
          by simp
        ultimately
        show ?thesis
          using Cons
          using ‹getWatch1 ?state' clause = Some ?w1›
          using ‹getWatch2 ?state' clause = Some ?w2›
          using ‹Some literal = getWatch1 state clause›
          using ‹¬ literalTrue ?w1 (elements (getM ?state'))›
          using Some
          by (simp add: Let-def)
      next
        case None
        show ?thesis
        proof (cases literalFalse ?w1 (elements (getM ?state')))
          case True
       let ?state'' = ?state'(|getConflictFlag := True, getConflictClause
:= clause|)

          from Cons(2)
          have InvariantWatchesEl (getF ?state'') (getWatch1 ?state'')
(getWatch2 ?state'')
            unfolding InvariantWatchesEl-def
            unfolding swapWatches-def
            by auto
          moreover
```

from *Cons*(*3*)
have *InvariantWatchesDiffer* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)
unfolding *InvariantWatchesDiffer-def*
unfolding *swapWatches-def*
by *auto*
moreover
have *getF ?state''* = *getF state*
unfolding *swapWatches-def*
by *simp*
ultimately
show *?thesis*
using *Cons*
using ‹*getWatch1 ?state' clause* = *Some ?w1*›
using ‹*getWatch2 ?state' clause* = *Some ?w2*›
using ‹*Some literal* = *getWatch1 state clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state'*))›
using *None*
using ‹*literalFalse ?w1* (*elements* (*getM ?state'*))›
by (*simp add*: *Let-def*)
next
case *False*
let *?state''* = *setReason ?w1 clause* (*?state'*⦇*getQ* := (*if ?w1 el* (*getQ ?state'*) *then* (*getQ ?state'*) *else* (*getQ ?state'*) @ [*?w1*])⦈)

from *Cons*(*2*)
have *InvariantWatchesEl* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)
unfolding *InvariantWatchesEl-def*
unfolding *swapWatches-def*
unfolding *setReason-def*
by *auto*
moreover
from *Cons*(*3*)
have *InvariantWatchesDiffer* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)
unfolding *InvariantWatchesDiffer-def*
unfolding *swapWatches-def*
unfolding *setReason-def*
by *auto*
moreover
have *getF ?state''* = *getF state*
unfolding *swapWatches-def*
unfolding *setReason-def*
by *simp*
ultimately
show *?thesis*
using *Cons*
using ‹*getWatch1 ?state' clause* = *Some ?w1*›

```
        using ‹getWatch2 ?state' clause = Some ?w2›
        using ‹Some literal = getWatch1 state clause›
        using ‹¬ literalTrue ?w1 (elements (getM ?state'))›
        using None
        using ‹¬ literalFalse ?w1 (elements (getM ?state'))›
        by (simp add: Let-def)
      qed
    qed
  qed
next
  case False
  let ?state' = state
  let ?w1 = wa
  have getWatch1 ?state' clause = Some ?w1
    using ‹getWatch1 state clause = Some wa›
    unfolding swapWatches-def
    by auto
  let ?w2 = wb
  have getWatch2 ?state' clause = Some ?w2
    using ‹getWatch2 state clause = Some wb›
    unfolding swapWatches-def
    by auto
  show ?thesis
  proof (cases literalTrue ?w1 (elements (getM ?state')))
    case True
    thus ?thesis
      using Cons
      using ‹¬ Some literal = getWatch1 state clause›
      using ‹getWatch1 ?state' clause = Some ?w1›
      using ‹getWatch2 ?state' clause = Some ?w2›
      using ‹literalTrue ?w1 (elements (getM ?state'))›
      by (simp add:Let-def)
  next
    case False
    show ?thesis
     proof (cases getNonWatchedUnfalsifiedLiteral (nth (getF ?state')
clause) ?w1 ?w2 (getM ?state'))
      case (Some l')
      hence l' el (nth (getF ?state') clause) l' ≠ ?w1 l' ≠ ?w2
        using getNonWatchedUnfalsifiedLiteralSomeCharacterization
        using ‹getWatch1 ?state' clause = Some ?w1›
        using ‹getWatch2 ?state' clause = Some ?w2›
        unfolding swapWatches-def
        by auto

      let ?state'' = setWatch2 clause l' ?state'

      from Cons(2)
       have InvariantWatchesEl (getF ?state'') (getWatch1 ?state'')
```

(*getWatch2 ?state''*)
using ‹*l' el* (*nth* (*getF ?state'*) *clause*)›
unfolding *InvariantWatchesEl-def*
unfolding *setWatch2-def*
by *auto*
moreover
from *Cons*(*3*)
have *InvariantWatchesDiffer* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)
using ‹*l'* ≠ *?w1*›
using ‹*getWatch1 ?state' clause* = *Some ?w1*›
using ‹*getWatch2 ?state' clause* = *Some ?w2*›
unfolding *InvariantWatchesDiffer-def*
unfolding *setWatch2-def*
by *auto*
moreover
have *getF ?state''* = *getF state*
unfolding *setWatch2-def*
by *simp*
ultimately
show *?thesis*
using *Cons*
using ‹*getWatch1 ?state' clause* = *Some ?w1*›
using ‹*getWatch2 ?state' clause* = *Some ?w2*›
using ‹¬ *Some literal* = *getWatch1 state clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state'*))›
using *Some*
by (*simp add*: *Let-def*)
next
case *None*
show *?thesis*
proof (*cases literalFalse ?w1* (*elements* (*getM ?state'*)))
case *True*
let *?state''* = *?state'*⦇*getConflictFlag* := *True*, *getConflictClause* := *clause*⦈

from *Cons*(*2*)
have *InvariantWatchesEl* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)
unfolding *InvariantWatchesEl-def*
by *auto*
moreover
from *Cons*(*3*)
have *InvariantWatchesDiffer* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)
unfolding *InvariantWatchesDiffer-def*
by *auto*
moreover
have *getF ?state''* = *getF state*

```
          by simp
        ultimately
        show ?thesis
          using Cons
          using ‹getWatch1 ?state' clause = Some ?w1›
          using ‹getWatch2 ?state' clause = Some ?w2›
          using ‹¬ Some literal = getWatch1 state clause›
          using ‹¬ literalTrue ?w1 (elements (getM ?state'))›
          using None
          using ‹literalFalse ?w1 (elements (getM ?state'))›
          by (simp add: Let-def)
      next
        case False
        let ?state'' = setReason ?w1 clause (?state'⦇getQ := (if ?w1
el (getQ ?state') then (getQ ?state') else (getQ ?state') @ [?w1])⦈)

        from Cons(2)
        have InvariantWatchesEl (getF ?state'') (getWatch1 ?state'')
(getWatch2 ?state'')
          unfolding InvariantWatchesEl-def
          unfolding setReason-def
          by auto
        moreover
        from Cons(3)
        have InvariantWatchesDiffer (getF ?state'') (getWatch1
?state'') (getWatch2 ?state'')
          unfolding InvariantWatchesDiffer-def
          unfolding setReason-def
          by auto
        moreover
        have getF ?state'' = getF state
          unfolding setReason-def
          by simp
        ultimately
        show ?thesis
          using Cons
          using ‹getWatch1 ?state' clause = Some ?w1›
          using ‹getWatch2 ?state' clause = Some ?w2›
          using ‹¬ Some literal = getWatch1 state clause›
          using ‹¬ literalTrue ?w1 (elements (getM ?state'))›
          using None
          using ‹¬ literalFalse ?w1 (elements (getM ?state'))›
          by (simp add: Let-def)
      qed
    qed
  qed
 qed
qed
```

lemma *InvariantWatchListsContainOnlyClausesFromFNotifyWatches-Loop*:
fixes *literal* :: *Literal* **and** *Wl* :: *nat list* **and** *newWl* :: *nat list* **and** *state* :: *State*
assumes
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
∀ (*c*::*nat*). *c* ∈ *set Wl* ∨ *c* ∈ *set newWl* ⟶ *0* ≤ *c* ∧ *c* < *length* (*getF state*)
shows
let state′ = (*notifyWatches-loop literal Wl newWl state*) *in*
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state′*) (*getF state′*)
using *assms*
proof (*induct Wl arbitrary*: *newWl state*)
case *Nil*
thus *?case*
unfolding *InvariantWatchListsContainOnlyClausesFromF-def*
by *simp*
next
case (*Cons clause Wl′*)
from ‹∀ *c*. *c* ∈ *set* (*clause* # *Wl′*) ∨ *c* ∈ *set newWl* ⟶ *0* ≤ *c* ∧ *c* < *length* (*getF state*)›
have *0* ≤ *clause* **and** *clause* < *length* (*getF state*)
by *auto*
then obtain *wa*::*Literal* **and** *wb*::*Literal*
where *getWatch1 state clause* = *Some wa* **and** *getWatch2 state clause* = *Some wb*
using *Cons*
unfolding *InvariantWatchesEl-def*
by *auto*
show *?case*
proof (*cases Some literal* = *getWatch1 state clause*)
case *True*
let *?state′* = *swapWatches clause state*
let *?w1* = *wb*
have *getWatch1 ?state′ clause* = *Some ?w1*
using ‹*getWatch2 state clause* = *Some wb*›
unfolding *swapWatches-def*
by *auto*
let *?w2* = *wa*
have *getWatch2 ?state′ clause* = *Some ?w2*
using ‹*getWatch1 state clause* = *Some wa*›
unfolding *swapWatches-def*
by *auto*
show *?thesis*

```
proof (cases literalTrue ?w1 (elements (getM ?state')))
  case True

  from Cons(2)
  have InvariantWatchListsContainOnlyClausesFromF (getWatchList
?state') (getF ?state')
    unfolding swapWatches-def
    by auto
  moreover
  from Cons(3)
    have InvariantWatchesEl (getF ?state') (getWatch1 ?state')
(getWatch2 ?state')
    unfolding InvariantWatchesEl-def
    unfolding swapWatches-def
    by auto
  moreover
  have (getF state) = (getF ?state')
    unfolding swapWatches-def
    by simp
  ultimately
  show ?thesis
    using Cons
    using ‹Some literal = getWatch1 state clause›
    using ‹getWatch1 ?state' clause = Some ?w1›
    using ‹getWatch2 ?state' clause = Some ?w2›
    using ‹literalTrue ?w1 (elements (getM ?state'))›
    by (simp add: Let-def)
next
  case False
  show ?thesis
  proof (cases getNonWatchedUnfalsifiedLiteral (nth (getF ?state')
clause) ?w1 ?w2 (getM ?state'))
    case (Some l')
    hence l' el (nth (getF ?state') clause)
      using getNonWatchedUnfalsifiedLiteralSomeCharacterization
      by simp

    let ?state'' = setWatch2 clause l' ?state'

    from Cons(2)
    have InvariantWatchListsContainOnlyClausesFromF (getWatchList
?state'') (getF ?state'')
      using ‹clause < length (getF state)›
      unfolding InvariantWatchListsContainOnlyClausesFromF-def
      unfolding swapWatches-def
      unfolding setWatch2-def
      by auto
    moreover
    from Cons(3)
```

have *InvariantWatchesEl* (*getF ?state''*) (*getWatch1 ?state''*)
(*getWatch2 ?state''*)
using ‹*l' el* (*nth* (*getF ?state'*) *clause*)›
unfolding *InvariantWatchesEl-def*
unfolding *swapWatches-def*
unfolding *setWatch2-def*
by *auto*
moreover
have (*getF state*) = (*getF ?state''*)
unfolding *swapWatches-def*
unfolding *setWatch2-def*
by *simp*
ultimately
show *?thesis*
using *Cons*
using ‹*getWatch1 ?state' clause* = *Some ?w1*›
using ‹*getWatch2 ?state' clause* = *Some ?w2*›
using ‹*Some literal* = *getWatch1 state clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state'*))›
using *Some*
by (*simp add*: *Let-def*)
next
case *None*
show *?thesis*
proof (*cases literalFalse ?w1* (*elements* (*getM ?state'*)))
case *True*
let *?state''* = *?state'*⦇*getConflictFlag* := *True*, *getConflictClause*
:= *clause*⦈

from *Cons*(*2*)
have *InvariantWatchListsContainOnlyClausesFromF* (*getWatchList*
?state'') (*getF ?state''*)
unfolding *swapWatches-def*
by *auto*
moreover
from *Cons*(*3*)
have *InvariantWatchesEl* (*getF ?state''*) (*getWatch1 ?state''*)
(*getWatch2 ?state''*)
unfolding *InvariantWatchesEl-def*
unfolding *swapWatches-def*
by *auto*
moreover
have (*getF state*) = (*getF ?state''*)
unfolding *swapWatches-def*
by *simp*
ultimately
show *?thesis*
using *Cons*
using ‹*getWatch1 ?state' clause* = *Some ?w1*›

using ‹*getWatch2 ?state′ clause = Some ?w2*›
using ‹*Some literal = getWatch1 state clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state′*))›
using *None*
using ‹*literalFalse ?w1* (*elements* (*getM ?state′*))›
by (*simp add*: *Let-def*)
next
case *False*
let *?state′′* = *setReason ?w1 clause* (*?state′*⦇*getQ* := (*if ?w1 el* (*getQ ?state′*) *then* (*getQ ?state′*) *else* (*getQ ?state′*) @ [*?w1*])⦈)

from *Cons*(*2*)
have *InvariantWatchListsContainOnlyClausesFromF* (*getWatchList ?state′′*) (*getF ?state′′*)
unfolding *swapWatches-def*
unfolding *setReason-def*
by *auto*
moreover
from *Cons*(*3*)
have *InvariantWatchesEl* (*getF ?state′′*) (*getWatch1 ?state′′*) (*getWatch2 ?state′′*)
unfolding *InvariantWatchesEl-def*
unfolding *swapWatches-def*
unfolding *setReason-def*
by *auto*
moreover
have (*getF state*) = (*getF ?state′′*)
unfolding *swapWatches-def*
unfolding *setReason-def*
by *simp*
ultimately
show *?thesis*
using *Cons*
using ‹*getWatch1 ?state′ clause = Some ?w1*›
using ‹*getWatch2 ?state′ clause = Some ?w2*›
using ‹*Some literal = getWatch1 state clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state′*))›
using *None*
using ‹¬ *literalFalse ?w1* (*elements* (*getM ?state′*))›
by (*simp add*: *Let-def*)
qed
qed
qed
next
case *False*
let *?state′* = *state*
let *?w1* = *wa*
have *getWatch1 ?state′ clause = Some ?w1*
using ‹*getWatch1 state clause = Some wa*›

unfolding *swapWatches-def*
by *auto*
let *?w2 = wb*
have *getWatch2 ?state′ clause = Some ?w2*
using ‹*getWatch2 state clause = Some wb*›
unfolding *swapWatches-def*
by *auto*
show *?thesis*
proof (*cases literalTrue ?w1* (*elements* (*getM ?state′*)))
case *True*
thus *?thesis*
using *Cons*
using ‹¬ *Some literal = getWatch1 state clause*›
using ‹*getWatch1 ?state′ clause = Some ?w1*›
using ‹*getWatch2 ?state′ clause = Some ?w2*›
using ‹*literalTrue ?w1* (*elements* (*getM ?state′*))›
by (*simp add*:*Let-def*)
next
case *False*
show *?thesis*
proof (*cases getNonWatchedUnfalsifiedLiteral* (*nth* (*getF ?state′*) *clause*) *?w1 ?w2* (*getM ?state′*))
case (*Some l′*)
hence *l′ el* (*nth* (*getF ?state′*) *clause*)
using *getNonWatchedUnfalsifiedLiteralSomeCharacterization*
by *simp*

let *?state′′ = setWatch2 clause l′ ?state′*

from *Cons*(*2*)
have *InvariantWatchListsContainOnlyClausesFromF* (*getWatchList ?state′′*) (*getF ?state′′*)
using ‹*clause* < *length* (*getF state*)›
unfolding *setWatch2-def*
unfolding *InvariantWatchListsContainOnlyClausesFromF-def*
by *auto*
moreover
from *Cons*(*3*)
have *InvariantWatchesEl* (*getF ?state′′*) (*getWatch1 ?state′′*) (*getWatch2 ?state′′*)
using ‹*l′ el* (*nth* (*getF ?state′*) *clause*)›
unfolding *InvariantWatchesEl-def*
unfolding *setWatch2-def*
by *auto*
moreover
have (*getF state*) = (*getF ?state′′*)
unfolding *setWatch2-def*
by *simp*
ultimately

```
        show ?thesis
          using Cons
          using ‹getWatch1 ?state' clause = Some ?w1›
          using ‹getWatch2 ?state' clause = Some ?w2›
          using ‹¬ Some literal = getWatch1 state clause›
          using ‹¬ literalTrue ?w1 (elements (getM ?state'))›
          using Some
          by (simp add: Let-def)
      next
        case None
        show ?thesis
        proof (cases literalFalse ?w1 (elements (getM ?state')))
          case True
       let ?state'' = ?state'⦇getConflictFlag := True, getConflictClause
:= clause⦈

          from Cons(3)
          have InvariantWatchesEl (getF ?state'') (getWatch1 ?state'')
(getWatch2 ?state'')
            unfolding InvariantWatchesEl-def
            by auto
          moreover
          have getF ?state'' = getF state
            by simp
          ultimately
          show ?thesis
            using Cons
            using ‹getWatch1 ?state' clause = Some ?w1›
            using ‹getWatch2 ?state' clause = Some ?w2›
            using ‹¬ Some literal = getWatch1 state clause›
            using ‹¬ literalTrue ?w1 (elements (getM ?state'))›
            using None
            using ‹literalFalse ?w1 (elements (getM ?state'))›
            by (simp add: Let-def)
        next
          case False
          let ?state'' = setReason ?w1 clause (?state'⦇getQ := (if ?w1
el (getQ ?state') then (getQ ?state') else (getQ ?state') @ [?w1])⦈)

          from Cons(2)
       have InvariantWatchListsContainOnlyClausesFromF (getWatchList
?state'') (getF ?state'')
            unfolding setReason-def
            by auto
          moreover
          from Cons(3)
          have InvariantWatchesEl (getF ?state'') (getWatch1 ?state'')
(getWatch2 ?state'')
            unfolding InvariantWatchesEl-def
```

unfolding *setReason-def*
by *auto*
moreover
have *getF ?state″ = getF state*
unfolding *setReason-def*
by *simp*
ultimately
show *?thesis*
using *Cons*
using ‹*getWatch1 ?state′ clause = Some ?w1*›
using ‹*getWatch2 ?state′ clause = Some ?w2*›
using ‹¬ *Some literal = getWatch1 state clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state′*))›
using *None*
using ‹¬ *literalFalse ?w1* (*elements* (*getM ?state′*))›
by (*simp add*: *Let-def*)
qed
qed
qed
qed
qed

lemma *InvariantWatchListsCharacterizationNotifyWatchesLoop*:
fixes *literal* :: *Literal* **and** *Wl* :: *nat list* **and** *newWl* :: *nat list* **and** *state* :: *State*
assumes
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchListsUniq (*getWatchList state*)
∀ (*c*::*nat*). *c* ∈ *set Wl* ⟶ *0* ≤ *c* ∧ *c* < *length* (*getF state*)
∀ (*c*::*nat*) (*l*::*Literal*). *l* ≠ *literal* ⟶
(*c* ∈ *set* (*getWatchList state l*)) = (*Some l* = *getWatch1 state c* ∨ *Some l* = *getWatch2 state c*)
∀ (*c*::*nat*). (*c* ∈ *set newWl* ∨ *c* ∈ *set Wl*) = (*Some literal* = (*getWatch1 state c*) ∨ *Some literal* = (*getWatch2 state c*))
set Wl ∩ *set newWl* = {}
uniq Wl
uniq newWl
shows
let state′ = (*notifyWatches-loop literal Wl newWl state*) *in*
InvariantWatchListsCharacterization (*getWatchList state′*) (*getWatch1 state′*) (*getWatch2 state′*) ∧
InvariantWatchListsUniq (*getWatchList state′*)
using *assms*
proof (*induct Wl arbitrary*: *newWl state*)
case *Nil*
thus *?case*

```
    unfolding InvariantWatchListsCharacterization-def
    unfolding InvariantWatchListsUniq-def
    by simp
next
  case (Cons clause Wl′)
  from ‹uniq (clause # Wl′)›
  have clause ∉ set Wl′
    by (simp add:uniqAppendIff)

  have set Wl′ ∩ set (clause # newWl) = {}
    using Cons(8)
    using ‹clause ∉ set Wl′›
    by simp

  have uniq Wl′
    using Cons(9)
    using uniqAppendIff
    by simp

  have uniq (clause # newWl)
    using Cons(10) Cons(8)
    using uniqAppendIff
    by force

  from ‹∀ c. c ∈ set (clause # Wl′) ⟶ 0 ≤ c ∧ c < length (getF
state)›
  have 0 ≤ clause and clause < length (getF state)
    by auto
  then obtain wa::Literal and wb::Literal
     where getWatch1 state clause = Some wa and getWatch2 state
clause = Some wb
    using Cons
    unfolding InvariantWatchesEl-def
    by auto
  show ?case
  proof (cases Some literal = getWatch1 state clause)
    case True
    let ?state′ = swapWatches clause state
    let ?w1 = wb
    have getWatch1 ?state′ clause = Some ?w1
      using ‹getWatch2 state clause = Some wb›
      unfolding swapWatches-def
      by auto
    let ?w2 = wa
    have getWatch2 ?state′ clause = Some ?w2
      using ‹getWatch1 state clause = Some wa›
      unfolding swapWatches-def
      by auto
    show ?thesis
```

proof (*cases literalTrue ?w1* (*elements* (*getM ?state'*)))
case *True*

from *Cons*(*2*)
have *InvariantWatchesEl* (*getF ?state'*) (*getWatch1 ?state'*) (*getWatch2 ?state'*)
unfolding *InvariantWatchesEl-def*
unfolding *swapWatches-def*
by *auto*
moreover
from *Cons*(*3*)
have *InvariantWatchesDiffer* (*getF ?state'*) (*getWatch1 ?state'*) (*getWatch2 ?state'*)
unfolding *InvariantWatchesDiffer-def*
unfolding *swapWatches-def*
by *auto*
moreover
from *Cons*(*4*)
have *InvariantWatchListsUniq* (*getWatchList ?state'*)
unfolding *InvariantWatchListsUniq-def*
unfolding *swapWatches-def*
by *auto*
moreover
have (*getF ?state'*) = (*getF state*) **and** (*getWatchList ?state'*) = (*getWatchList state*)
unfolding *swapWatches-def*
by *auto*
moreover
have ∀ *c l*. *l* ≠ *literal* ⟶
(*c* ∈ *set* (*getWatchList ?state' l*)) =
(*Some l* = *getWatch1 ?state' c* ∨ *Some l* = *getWatch2 ?state' c*)
using *Cons*(*6*)
using ‹(*getWatchList ?state'*) = (*getWatchList state*)›
using *swapWatchesEffect*
by *auto*
moreover
have ∀ *c*. (*c* ∈ *set* (*clause* # *newWl*) ∨ *c* ∈ *set Wl'*) =
(*Some literal* = *getWatch1 ?state' c* ∨ *Some literal* = *getWatch2 ?state' c*)
using *Cons*(*7*)
using *swapWatchesEffect*
by *auto*
ultimately
show *?thesis*
using *Cons*(*1*)[*of ?state' clause* # *newWl*]
using *Cons*(*5*)
using ‹*Some literal* = *getWatch1 state clause*›
using ‹*getWatch1 ?state' clause* = *Some ?w1*›

```
        using ‹getWatch2 ?state' clause = Some ?w2›
        using ‹literalTrue ?w1 (elements (getM ?state'))›
        using ‹uniq Wl'›
        using ‹uniq (clause # newWl)›
        using ‹set Wl' ∩ set (clause # newWl) = {}›
        by (simp add: Let-def)
    next
      case False
      show ?thesis
      proof (cases getNonWatchedUnfalsifiedLiteral (nth (getF ?state')
clause) ?w1 ?w2 (getM ?state'))
        case (Some l')
        hence l' el (nth (getF ?state') clause) l' ≠ literal l' ≠ ?w1 l' ≠
?w2
          using getNonWatchedUnfalsifiedLiteralSomeCharacterization
          using ‹getWatch1 ?state' clause = Some ?w1›
          using ‹getWatch2 ?state' clause = Some ?w2›
          using ‹Some literal = getWatch1 state clause›
          unfolding swapWatches-def
          by auto

        let ?state'' = setWatch2 clause l' ?state'

        from Cons(2)
        have InvariantWatchesEl (getF ?state'') (getWatch1 ?state'')
(getWatch2 ?state'')
          using ‹l' el (nth (getF ?state') clause)›
          unfolding InvariantWatchesEl-def
          unfolding swapWatches-def
          unfolding setWatch2-def
          by auto
        moreover
        from Cons(3)
        have InvariantWatchesDiffer (getF ?state'') (getWatch1 ?state'')
(getWatch2 ?state'')
          using ‹getWatch1 ?state' clause = Some ?w1›
          using ‹l' ≠ ?w1›
          unfolding InvariantWatchesDiffer-def
          unfolding swapWatches-def
          unfolding setWatch2-def
          by simp
        moreover
        have clause ∉ set (getWatchList state l')
          using ‹l' ≠ literal›
          using ‹l' ≠ ?w1› ‹l' ≠ ?w2›
          using ‹getWatch1 ?state' clause = Some ?w1›
          using ‹getWatch2 ?state' clause = Some ?w2›
          using Cons(6)
          unfolding swapWatches-def
```

by *simp*
with *Cons(4)*
have *InvariantWatchListsUniq* (*getWatchList ?state''*)
unfolding *InvariantWatchListsUniq-def*
unfolding *swapWatches-def*
unfolding *setWatch2-def*
using *uniqAppendIff*
by *force*
moreover
have (*getF ?state''*) = (*getF state*) **and**
(*getWatchList ?state''*) = (*getWatchList state*)(*l'* := *clause* # (*getWatchList state l'*))
unfolding *swapWatches-def*
unfolding *setWatch2-def*
by *auto*
moreover
have ∀ *c l*. *l* ≠ *literal* ⟶
(*c* ∈ *set* (*getWatchList ?state'' l*)) =
(*Some l* = *getWatch1 ?state'' c* ∨ *Some l* = *getWatch2 ?state'' c*)
proof−
{
fix *c*::*nat* **and** *l*::*Literal*
assume *l* ≠ *literal*
have (*c* ∈ *set* (*getWatchList ?state'' l*)) = (*Some l* = *getWatch1 ?state'' c* ∨ *Some l* = *getWatch2 ?state'' c*)
proof (*cases c* = *clause*)
case *True*
show *?thesis*
proof (*cases l* = *l'*)
case *True*
thus *?thesis*
using ‹*c* = *clause*›
unfolding *setWatch2-def*
by *simp*
next
case *False*
show *?thesis*
using *Cons(6)*
using ‹(*getWatchList ?state''*) = (*getWatchList state*)(*l'* := *clause* # (*getWatchList state l'*))›
using ‹*l* ≠ *l'*›
using ‹*l* ≠ *literal*›
using ‹*getWatch1 ?state' clause* = *Some ?w1*›
using ‹*getWatch2 ?state' clause* = *Some ?w2*›
using ‹*Some literal* = *getWatch1 state clause*›
using ‹*c* = *clause*›
using *swapWatchesEffect*
unfolding *swapWatches-def*

```
          unfolding setWatch2-def
          by simp
        qed
      next
        case False
        thus ?thesis
          using Cons(6)
          using ‹l ≠ literal›
          using ‹(getWatchList ?state'') = (getWatchList state)(l'
:= clause # (getWatchList state l'))›
          using ‹c ≠ clause›
          unfolding setWatch2-def
          using swapWatchesEffect[of clause state c]
          by auto
      qed
    }
    thus ?thesis
      by simp
  qed
  moreover
  have ∀ c. (c ∈ set newWl ∨ c ∈ set Wl') =
  (Some literal = getWatch1 ?state'' c ∨ Some literal = getWatch2
?state'' c)
  proof-
    show ?thesis
    proof
      fix c :: nat
      show (c ∈ set newWl ∨ c ∈ set Wl') =
          (Some literal = getWatch1 ?state'' c ∨ Some literal =
getWatch2 ?state'' c)
      proof
        assume c ∈ set newWl ∨ c ∈ set Wl'
        show Some literal = getWatch1 ?state'' c ∨ Some literal
= getWatch2 ?state'' c
        proof-
          from ‹c ∈ set newWl ∨ c ∈ set Wl'›
          have Some literal = getWatch1 state c ∨ Some literal =
getWatch2 state c
            using Cons(7)
            by auto

          from Cons(8) ‹clause ∉ set Wl'› ‹c ∈ set newWl ∨ c ∈
set Wl'›
          have c ≠ clause
            by auto

          show ?thesis
            using ‹Some literal = getWatch1 state c ∨ Some literal
= getWatch2 state c›
```

```
                using ‹c ≠ clause›
                using swapWatchesEffect
                unfolding setWatch2-def
                by simp
            qed
          next
            assume Some literal = getWatch1 ?state'' c ∨ Some literal
= getWatch2 ?state'' c
            show c ∈ set newWl ∨ c ∈ set Wl'
            proof–
              have Some literal ≠ getWatch1 ?state'' clause ∧  Some
literal ≠ getWatch2 ?state'' clause
                using ‹l' ≠ literal›
                using ‹clause < length (getF state)›
                using ‹InvariantWatchesDiffer (getF state) (getWatch1
state) (getWatch2 state)›
                using ‹getWatch1 ?state' clause = Some ?w1›
                using ‹getWatch2 ?state' clause = Some ?w2›
                using ‹Some literal = getWatch1 state clause›
                unfolding InvariantWatchesDiffer-def
                unfolding setWatch2-def
                unfolding swapWatches-def
                by auto
              thus ?thesis
                using ‹Some literal = getWatch1 ?state'' c ∨ Some
literal = getWatch2 ?state'' c›
                using Cons(7)
                using swapWatchesEffect
                unfolding setWatch2-def
                by (auto split: if-split-asm)
            qed
          qed
        qed
      qed
      moreover
      have ∀ c. (c ∈ set (clause # newWl) ∨ c ∈ set Wl') =
      (Some literal = getWatch1 ?state' c ∨ Some literal = getWatch2
?state' c)
        using Cons(7)
        using swapWatchesEffect
        by auto
      ultimately
      show ?thesis
        using Cons(1)[of ?state'' newWl]
        using Cons(5)
        using ‹uniq Wl'›
        using ‹uniq newWl›
        using ‹set Wl' ∩ set (clause # newWl) = {}›
        using ‹getWatch1 ?state' clause = Some ?w1›
```

using ‹*getWatch2 ?state′ clause* = *Some ?w2*›
using ‹*Some literal* = *getWatch1 state clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state′*))›
using *Some*
by (*simp add*: *Let-def fun-upd-def*)
next
case *None*
show *?thesis*
proof (*cases literalFalse ?w1* (*elements* (*getM ?state′*)))
case *True*
let *?state′′* = *?state′*⦇*getConflictFlag* := *True*, *getConflictClause* := *clause*⦈

from *Cons*(*2*)
have *InvariantWatchesEl* (*getF ?state′′*) (*getWatch1 ?state′′*) (*getWatch2 ?state′′*)
unfolding *InvariantWatchesEl-def*
unfolding *swapWatches-def*
by *auto*
moreover
from *Cons*(*3*)
have *InvariantWatchesDiffer* (*getF ?state′′*) (*getWatch1 ?state′′*) (*getWatch2 ?state′′*)
unfolding *InvariantWatchesDiffer-def*
unfolding *swapWatches-def*
by *auto*
moreover
from *Cons*(*4*)
have *InvariantWatchListsUniq* (*getWatchList ?state′′*)
unfolding *InvariantWatchListsUniq-def*
unfolding *swapWatches-def*
by *auto*
moreover
have (*getF state*) = (*getF ?state′′*) **and** (*getWatchList state*) = (*getWatchList ?state′′*)
unfolding *swapWatches-def*
by *auto*
moreover
have ∀ *c l*. *l* ≠ *literal* ⟶
(*c* ∈ *set* (*getWatchList ?state′′ l*)) =
(*Some l* = *getWatch1 ?state′′ c* ∨ *Some l* = *getWatch2 ?state′′ c*)
using *Cons*(*6*)
using ‹(*getWatchList state*) = (*getWatchList ?state′′*)›
using *swapWatchesEffect*
by *auto*
moreover
have ∀ *c*. (*c* ∈ *set* (*clause* # *newWl*) ∨ *c* ∈ *set Wl′*) =
(*Some literal* = *getWatch1 ?state′′ c* ∨ *Some literal* =

getWatch2 ?state'' c)
using *Cons*(*7*)
using *swapWatchesEffect*
by *auto*
ultimately
show *?thesis*
using *Cons*(*1*)[*of ?state'' clause # newWl*]
using *Cons*(*5*)
using ‹*getWatch1 ?state' clause = Some ?w1*›
using ‹*getWatch2 ?state' clause = Some ?w2*›
using ‹*Some literal = getWatch1 state clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state'*))›
using *None*
using ‹*literalFalse ?w1* (*elements* (*getM ?state'*))›
using ‹*uniq Wl'*›
using ‹*uniq* (*clause # newWl*)›
using ‹*set Wl'* ∩ *set* (*clause # newWl*) = {}›
by (*simp add*: *Let-def*)
next
case *False*
let *?state'' = setReason ?w1 clause* (*?state'*⦇*getQ* := (*if ?w1 el* (*getQ ?state'*) *then* (*getQ ?state'*) *else* (*getQ ?state'*) @ [*?w1*])⦈)

from *Cons*(*2*)
have *InvariantWatchesEl* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)
unfolding *InvariantWatchesEl-def*
unfolding *swapWatches-def*
unfolding *setReason-def*
by *auto*
moreover
from *Cons*(*3*)
have *InvariantWatchesDiffer* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)
unfolding *InvariantWatchesDiffer-def*
unfolding *swapWatches-def*
unfolding *setReason-def*
by *auto*
moreover
from *Cons*(*4*)
have *InvariantWatchListsUniq* (*getWatchList ?state''*)
unfolding *InvariantWatchListsUniq-def*
unfolding *swapWatches-def*
unfolding *setReason-def*
by *auto*
moreover
have (*getF state*) = (*getF ?state''*) **and** (*getWatchList state*) = (*getWatchList ?state''*)
unfolding *swapWatches-def*

```
          unfolding setReason-def
          by auto
        moreover
        have ∀ c l. l ≠ literal ⟶
          (c ∈ set (getWatchList ?state'' l)) =
             (Some l = getWatch1 ?state'' c ∨ Some l = getWatch2
?state'' c)
          using Cons(6)
          using ‹(getWatchList state) = (getWatchList ?state'')›
          using swapWatchesEffect
          unfolding setReason-def
          by auto
        moreover
        have ∀ c. (c ∈ set (clause # newWl) ∨ c ∈ set Wl') =
             (Some literal = getWatch1 ?state'' c ∨ Some literal =
getWatch2 ?state'' c)
          using Cons(7)
          using swapWatchesEffect
          unfolding setReason-def
          by auto
        ultimately
        show ?thesis
          using Cons(1)[of ?state'' clause # newWl]
          using Cons(5)
          using ‹getWatch1 ?state' clause = Some ?w1›
          using ‹getWatch2 ?state' clause = Some ?w2›
          using ‹Some literal = getWatch1 state clause›
          using ‹¬ literalTrue ?w1 (elements (getM ?state'))›
          using None
          using ‹¬ literalFalse ?w1 (elements (getM ?state'))›
          using ‹uniq Wl'›
          using ‹uniq (clause # newWl)›
          using ‹set Wl' ∩ set (clause # newWl) = {}›
          by (simp add: Let-def)
      qed
    qed
  qed
next
  case False
  let ?state' = state
  let ?w1 = wa
  have getWatch1 ?state' clause = Some ?w1
    using ‹getWatch1 state clause = Some wa›
    unfolding swapWatches-def
    by auto
  let ?w2 = wb
  have getWatch2 ?state' clause = Some ?w2
    using ‹getWatch2 state clause = Some wb›
    unfolding swapWatches-def
```

by *auto*

have *Some literal = getWatch2 state clause*
using ‹*getWatch1 ?state′ clause = Some ?w1*›
using ‹*getWatch2 ?state′ clause = Some ?w2*›
using ‹*Some literal ≠ getWatch1 state clause*›
using *Cons*(*7*)
by *force*

show *?thesis*
proof (*cases literalTrue ?w1* (*elements* (*getM ?state′*)))
case *True*
from *Cons*(*7*) **have**
∀ *c*. (*c* ∈ *set* (*clause* # *newWl*) ∨ *c* ∈ *set Wl′*) =
(*Some literal = getWatch1 state c* ∨ *Some literal = getWatch2 state c*)
by *auto*
thus *?thesis*
using *Cons*(*1*)[*of ?state′ clause* # *newWl*]
using *Cons*(*2*) *Cons*(*3*) *Cons*(*4*) *Cons*(*5*) *Cons*(*6*)
using ‹¬ *Some literal = getWatch1 state clause*›
using ‹*getWatch1 ?state′ clause = Some ?w1*›
using ‹*getWatch2 ?state′ clause = Some ?w2*›
using ‹*literalTrue ?w1* (*elements* (*getM ?state′*))›
using ‹*uniq* (*clause* # *newWl*)›
using ‹*uniq Wl′*›
using ‹*set Wl′* ∩ *set* (*clause* # *newWl*) = {}›
by *simp*
next
case *False*
show *?thesis*
proof (*cases getNonWatchedUnfalsifiedLiteral* (*nth* (*getF ?state′*) *clause*) *?w1 ?w2* (*getM ?state′*))
case (*Some l′*)
hence *l′ el* (*nth* (*getF ?state′*) *clause*) *l′* ≠ *literal l′* ≠ *?w1 l′* ≠ *?w2*
using *getNonWatchedUnfalsifiedLiteralSomeCharacterization*
using ‹*Some literal = getWatch2 state clause*›
using ‹*getWatch1 ?state′ clause = Some ?w1*›
using ‹*getWatch2 ?state′ clause = Some ?w2*›
by *auto*

let *?state″ = setWatch2 clause l′ ?state′*

from *Cons*(*2*)
have *InvariantWatchesEl* (*getF ?state″*) (*getWatch1 ?state″*) (*getWatch2 ?state″*)
using ‹*l′ el* (*nth* (*getF ?state′*) *clause*)›
unfolding *InvariantWatchesEl-def*

unfolding *setWatch2-def*
by *auto*
moreover
from *Cons(3)*
have *InvariantWatchesDiffer* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)
using ‹*getWatch1 ?state' clause = Some ?w1*›
using ‹$l' \neq$ *?w1*›
unfolding *InvariantWatchesDiffer-def*
unfolding *setWatch2-def*
by *simp*
moreover
have *clause* $\notin$ *set* (*getWatchList state l'*)
using ‹$l' \neq$ *literal*›
using ‹$l' \neq$ *?w1*› ‹$l' \neq$ *?w2*›
using ‹*getWatch1 ?state' clause = Some ?w1*›
using ‹*getWatch2 ?state' clause = Some ?w2*›
using *Cons(6)*
by *simp*
with *Cons(4)*
have *InvariantWatchListsUniq* (*getWatchList ?state''*)
unfolding *InvariantWatchListsUniq-def*
unfolding *setWatch2-def*
using *uniqAppendIff*
by *force*
moreover
have (*getF ?state''*) = (*getF state*) **and**
(*getWatchList ?state''*) = (*getWatchList state*)(l' := *clause* # (*getWatchList state l'*))
unfolding *setWatch2-def*
by *auto*
moreover
have $\forall$ *c l.* $l \neq$ *literal* $\longrightarrow$
($c \in$ *set* (*getWatchList ?state'' l*)) =
(*Some l = getWatch1 ?state'' c* $\vee$ *Some l = getWatch2 ?state'' c*)
proof−
{
fix *c::nat* **and** *l::Literal*
assume $l \neq$ *literal*
have ($c \in$ *set* (*getWatchList ?state'' l*)) = (*Some l = getWatch1 ?state'' c* $\vee$ *Some l = getWatch2 ?state'' c*)
proof (*cases c = clause*)
case *True*
show *?thesis*
proof (*cases l = l'*)
case *True*
thus *?thesis*
using ‹*c = clause*›

```
            unfolding setWatch2-def
            by simp
        next
          case False
          show ?thesis
            using Cons(6)
            using ‹(getWatchList ?state'') = (getWatchList state)(l'
:= clause # (getWatchList state l'))›
            using ‹l ≠ l'›
            using ‹l ≠ literal›
            using ‹getWatch1 ?state' clause = Some ?w1›
            using ‹getWatch2 ?state' clause = Some ?w2›
            using ‹Some literal = getWatch2 state clause›
            using ‹c = clause›
            unfolding setWatch2-def
            by simp
        qed
      next
        case False
        thus ?thesis
          using Cons(6)
          using ‹l ≠ literal›
          using ‹(getWatchList ?state'') = (getWatchList state)(l'
:= clause # (getWatchList state l'))›
          using ‹c ≠ clause›
          unfolding setWatch2-def
          by auto
      qed
    }
    thus ?thesis
      by simp
  qed
  moreover
  have ∀ c. (c ∈ set newWl ∨ c ∈ set Wl') =
  (Some literal = getWatch1 ?state'' c ∨ Some literal = getWatch2
?state'' c)
  proof−
    show ?thesis
    proof
      fix c :: nat
      show (c ∈ set newWl ∨ c ∈ set Wl') =
           (Some literal = getWatch1 ?state'' c ∨ Some literal =
getWatch2 ?state'' c)
      proof
        assume c ∈ set newWl ∨ c ∈ set Wl'
        show Some literal = getWatch1 ?state'' c ∨ Some literal
= getWatch2 ?state'' c
        proof−
          from ‹c ∈ set newWl ∨ c ∈ set Wl'›
```

have *Some literal* = *getWatch1 state c* ∨ *Some literal* = *getWatch2 state c*
using *Cons*(*7*)
by *auto*

from *Cons*(*8*) ‹*clause* ∉ *set Wl*′› ‹*c* ∈ *set newWl* ∨ *c* ∈ *set Wl*′›
have *c* ≠ *clause*
by *auto*

show *?thesis*
using ‹*Some literal* = *getWatch1 state c* ∨ *Some literal* = *getWatch2 state c*›
using ‹*c* ≠ *clause*›
unfolding *setWatch2-def*
by *simp*
qed
next
assume *Some literal* = *getWatch1 ?state*′′ *c* ∨ *Some literal* = *getWatch2 ?state*′′ *c*
show *c* ∈ *set newWl* ∨ *c* ∈ *set Wl*′
proof−
have *Some literal* ≠ *getWatch1 ?state*′′ *clause* ∧ *Some literal* ≠ *getWatch2 ?state*′′ *clause*
using ‹*l*′ ≠ *literal*›
using ‹*clause* < *length* (*getF state*)›
using ‹*InvariantWatchesDiffer* (*getF state*) (*getWatch1 state*) (*getWatch2 state*)›
using ‹*getWatch1 ?state*′ *clause* = *Some ?w1*›
using ‹*getWatch2 ?state*′ *clause* = *Some ?w2*›
using ‹*Some literal* = *getWatch2 state clause*›
unfolding *InvariantWatchesDiffer-def*
unfolding *setWatch2-def*
by *auto*
thus *?thesis*
using ‹*Some literal* = *getWatch1 ?state*′′ *c* ∨ *Some literal* = *getWatch2 ?state*′′ *c*›
using *Cons*(*7*)
unfolding *setWatch2-def*
by (*auto split*: *if-split-asm*)
qed
qed
qed
qed
moreover
have ∀ *c*. (*c* ∈ *set* (*clause* # *newWl*) ∨ *c* ∈ *set Wl*′) = (*Some literal* = *getWatch1 ?state*′ *c* ∨ *Some literal* = *getWatch2 ?state*′ *c*)
using *Cons*(*7*)

by *auto*
ultimately
show *?thesis*
using *Cons*(*1*)[*of ?state'' newWl*]
using *Cons*(*5*)
using ‹*uniq Wl'*›
using ‹*uniq newWl*›
using ‹*set Wl'* ∩ *set* (*clause* # *newWl*) = {}›
using ‹*getWatch1 ?state' clause* = *Some ?w1*›
using ‹*getWatch2 ?state' clause* = *Some ?w2*›
using ‹¬ *Some literal* = *getWatch1 state clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state'*))›
using *Some*
by (*simp add*: *Let-def fun-upd-def*)
next
case *None*
show *?thesis*
proof (*cases literalFalse ?w1* (*elements* (*getM ?state'*)))
case *True*
let *?state''* = *?state'*⦇*getConflictFlag* := *True*, *getConflictClause* := *clause*⦈

from *Cons*(*2*)
have *InvariantWatchesEl* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)
unfolding *InvariantWatchesEl-def*
by *auto*
moreover
from *Cons*(*3*)
have *InvariantWatchesDiffer* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)
unfolding *InvariantWatchesDiffer-def*
by *auto*
moreover
from *Cons*(*4*)
have *InvariantWatchListsUniq* (*getWatchList ?state''*)
unfolding *InvariantWatchListsUniq-def*
by *auto*
moreover
have (*getF state*) = (*getF ?state''*)
by *auto*
moreover
have ∀ *c l*. *l* ≠ *literal* ⟶
(*c* ∈ *set* (*getWatchList ?state'' l*)) =
(*Some l* = *getWatch1 ?state'' c* ∨ *Some l* = *getWatch2 ?state'' c*)
using *Cons*(*6*)
by *simp*
moreover

have ∀ *c*. (*c* ∈ *set* (*clause* # *newWl*) ∨ *c* ∈ *set Wl'*) =
(*Some literal* = *getWatch1 ?state'' c* ∨ *Some literal* = *getWatch2 ?state'' c*)
using *Cons*(*7*)
by *auto*
ultimately
have *let state'* = *notifyWatches-loop literal Wl'* (*clause* # *newWl*) *?state'' in*
InvariantWatchListsCharacterization (*getWatchList state'*) (*getWatch1 state'*) (*getWatch2 state'*) ∧
InvariantWatchListsUniq (*getWatchList state'*)
using *Cons*(*1*)[*of ?state'' clause* # *newWl*]
using *Cons*(*5*)
using ‹*uniq Wl'*›
using ‹*uniq* (*clause* # *newWl*)›
using ‹*set Wl'* ∩ *set* (*clause* # *newWl*) = {}›
apply (*simp only*: *Let-def*)
by (*simp* (*no-asm-use*)) (*simp*)
thus *?thesis*
using ‹*getWatch1 ?state' clause* = *Some ?w1*›
using ‹*getWatch2 ?state' clause* = *Some ?w2*›
using ‹*Some literal* ≠ *getWatch1 state clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state'*))›
using *None*
using ‹*literalFalse ?w1* (*elements* (*getM ?state'*))›
by (*simp add*: *Let-def*)
next
case *False*
let *?state''* = *setReason ?w1 clause* (*?state'*⦇*getQ* := (*if ?w1 el* (*getQ ?state'*) *then* (*getQ ?state'*) *else* (*getQ ?state'*) @ [*?w1*])⦈)

from *Cons*(*2*)
have *InvariantWatchesEl* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)
unfolding *InvariantWatchesEl-def*
unfolding *setReason-def*
by *auto*
moreover
from *Cons*(*3*)
have *InvariantWatchesDiffer* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)
unfolding *InvariantWatchesDiffer-def*
unfolding *setReason-def*
by *auto*
moreover
from *Cons*(*4*)
have *InvariantWatchListsUniq* (*getWatchList ?state''*)
unfolding *InvariantWatchListsUniq-def*

unfolding *setReason-def*
by *auto*
moreover
have (*getF state*) = (*getF ?state''*)
unfolding *setReason-def*
by *auto*
moreover
have ∀ *c l*. *l* ≠ *literal* ⟶
(*c* ∈ *set* (*getWatchList ?state'' l*)) =
(*Some l* = *getWatch1 ?state'' c* ∨ *Some l* = *getWatch2 ?state'' c*)
using *Cons*(*6*)
unfolding *setReason-def*
by *auto*
moreover
have ∀ *c*. (*c* ∈ *set* (*clause* # *newWl*) ∨ *c* ∈ *set Wl'*) =
(*Some literal* = *getWatch1 ?state'' c* ∨ *Some literal* = *getWatch2 ?state'' c*)
using *Cons*(*7*)
unfolding *setReason-def*
by *auto*
ultimately
show *?thesis*
using *Cons*(*1*)[*of ?state'' clause* # *newWl*]
using *Cons*(*5*)
using ‹*getWatch1 ?state' clause* = *Some ?w1*›
using ‹*getWatch2 ?state' clause* = *Some ?w2*›
using ‹¬ *Some literal* = *getWatch1 state clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state'*))›
using *None*
using ‹¬ *literalFalse ?w1* (*elements* (*getM ?state'*))›
using ‹*uniq Wl'*›
using ‹*uniq* (*clause* # *newWl*)›
using ‹*set Wl'* ∩ *set* (*clause* # *newWl*) = {}›
by (*simp add*: *Let-def*)
qed
qed
qed
qed
qed

lemma *NotifyWatchesLoopWatchCharacterizationEffect*:
fixes *literal* :: *Literal* **and** *Wl* :: *nat list* **and** *newWl* :: *nat list* **and** *state* :: *State*
assumes
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**

InvariantConsistent (*getM state*) **and**
InvariantUniq (*getM state*) **and**
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) *M*
∀ (*c*::*nat*). *c* ∈ *set Wl* ⟶ *0* ≤ *c* ∧ *c* < *length* (*getF state*) **and**
getM state = *M* @ [(*opposite literal*, *decision*)]
uniq Wl
∀ (*c*::*nat*). *c* ∈ *set Wl* ⟶ *Some literal* = (*getWatch1 state c*) ∨ *Some literal* = (*getWatch2 state c*)

shows
let state′ = *notifyWatches-loop literal Wl newWl state in*
∀ (*c*::*nat*). *c* ∈ *set Wl* ⟶ (∀ *w1 w2*.(*Some w1* = (*getWatch1 state′ c*) ∧ *Some w2* = (*getWatch2 state′ c*)) ⟶
(*watchCharacterizationCondition w1 w2* (*getM state′*) (*nth* (*getF state′*) *c*) ∧
watchCharacterizationCondition w2 w1 (*getM state′*) (*nth* (*getF state′*) *c*))
)
using *assms*
proof (*induct Wl arbitrary*: *newWl state*)
case *Nil*
thus *?case*
by *simp*
next
case (*Cons clause Wl′*)
from ‹∀ (*c*::*nat*). *c* ∈ *set* (*clause* # *Wl′*) ⟶ *0* ≤ *c* ∧ *c* < *length* (*getF state*)›
have *0* ≤ *clause* ∧ *clause* < *length* (*getF state*)
by *auto*
then obtain *wa*::*Literal* **and** *wb*::*Literal*
where *getWatch1 state clause* = *Some wa* **and** *getWatch2 state clause* = *Some wb*
using *Cons*
unfolding *InvariantWatchesEl-def*
by *auto*
have *uniq Wl′ clause* ∉ *set Wl′*
using *Cons*(*9*)
by (*auto simp add*: *uniqAppendIff*)
show *?case*
proof (*cases Some literal* = *getWatch1 state clause*)
case *True*
let *?state′* = *swapWatches clause state*
let *?w1* = *wb*
have *getWatch1 ?state′ clause* = *Some ?w1*
using ‹*getWatch2 state clause* = *Some wb*›
unfolding *swapWatches-def*
by *auto*
let *?w2* = *wa*

have *getWatch2 ?state′ clause* = *Some ?w2*
using ‹*getWatch1 state clause* = *Some wa*›
unfolding *swapWatches-def*
by *auto*
with *True* **have**
?w2 = *literal*
unfolding *swapWatches-def*
by *simp*

from ‹*InvariantWatchesEl* (*getF state*) (*getWatch1 state*) (*getWatch2 state*)›
have *?w1 el* (*nth* (*getF state*) *clause*) *?w2 el* (*nth* (*getF state*) *clause*)
using ‹*getWatch1 ?state′ clause* = *Some ?w1*›
using ‹*getWatch2 ?state′ clause* = *Some ?w2*›
using ‹$0 \leq$ *clause* $\wedge$ *clause* < *length* (*getF state*)›
unfolding *InvariantWatchesEl-def*
unfolding *swapWatches-def*
by *auto*

from ‹*InvariantWatchesDiffer* (*getF state*) (*getWatch1 state*) (*getWatch2 state*)›
have *?w1* ≠ *?w2*
using ‹*getWatch1 ?state′ clause* = *Some ?w1*›
using ‹*getWatch2 ?state′ clause* = *Some ?w2*›
using ‹$0 \leq$ *clause* $\wedge$ *clause* < *length* (*getF state*)›
unfolding *InvariantWatchesDiffer-def*
unfolding *swapWatches-def*
by *auto*

have ¬ *literalFalse ?w2* (*elements M*)
using ‹*?w2* = *literal*›
using *Cons*(*5*)
using *Cons*(*8*)
unfolding *InvariantUniq-def*
by (*simp add*: *uniqAppendIff*)

show *?thesis*
proof (*cases literalTrue ?w1* (*elements* (*getM ?state′*)))
case *True*

let *?fState* = *notifyWatches-loop literal Wl′* (*clause* # *newWl*) *?state′*

from *Cons*(*2*)
have *InvariantWatchesEl* (*getF ?state′*) (*getWatch1 ?state′*) (*getWatch2 ?state′*)
unfolding *InvariantWatchesEl-def*
unfolding *swapWatches-def*

by *auto*
moreover
from *Cons*(*3*)
have *InvariantWatchesDiffer* (*getF ?state'*) (*getWatch1 ?state'*) (*getWatch2 ?state'*)
unfolding *InvariantWatchesDiffer-def*
unfolding *swapWatches-def*
by *auto*
moreover
from *Cons*(*4*)
have *InvariantConsistent* (*getM ?state'*)
unfolding *InvariantConsistent-def*
unfolding *swapWatches-def*
by *simp*
moreover
from *Cons*(*5*)
have *InvariantUniq* (*getM ?state'*)
unfolding *InvariantUniq-def*
unfolding *swapWatches-def*
by *simp*
moreover
from *Cons*(*6*)
have *InvariantWatchCharacterization* (*getF ?state'*) (*getWatch1 ?state'*) (*getWatch2 ?state'*) *M*
unfolding *swapWatches-def*
unfolding *InvariantWatchCharacterization-def*
unfolding *watchCharacterizationCondition-def*
by *simp*
moreover
have *getM ?state'* = *getM state*
getF ?state' = *getF state*
unfolding *swapWatches-def*
by *auto*
moreover
have ∀ (*c*::*nat*). *c* ∈ *set Wl'* ⟶ *Some literal* = (*getWatch1 ?state' c*) ∨ *Some literal* = (*getWatch2 ?state' c*)
using *Cons*(*10*)
unfolding *swapWatches-def*
by *auto*
moreover
have *getWatch1 ?fState clause* = *getWatch1 ?state' clause* ∧ *getWatch2 ?fState clause* = *getWatch2 ?state' clause*
using ‹*clause* ∉ *set Wl'*›
using ‹*InvariantWatchesEl* (*getF ?state'*) (*getWatch1 ?state'*) (*getWatch2 ?state'*)› ‹*getF ?state'* = *getF state*›
using *Cons*(*7*)
using *notifyWatchesLoopPreservedWatches*[*of ?state' Wl' literal clause # newWl*]
by (*simp add*: *Let-def*)

moreover
have *watchCharacterizationCondition ?w1 ?w2* (*getM ?fState*) (*getF ?fState* ! *clause*) ∧
watchCharacterizationCondition ?w2 ?w1 (*getM ?fState*) (*getF ?fState* ! *clause*)
proof−
have (*getM ?fState*) = (*getM state*) ∧ (*getF ?fState* = *getF state*)
using *notifyWatchesLoopPreservedVariables*[*of ?state′ Wl′ literal clause* # *newWl*]
using ‹*InvariantWatchesEl* (*getF ?state′*) (*getWatch1 ?state′*) (*getWatch2 ?state′*)› ‹*getF ?state′* = *getF state*›
using *Cons*(*7*)
unfolding *swapWatches-def*
by (*simp add*: *Let-def*)
moreover
have ¬ *literalFalse ?w1* (*elements M*)
using ‹*literalTrue ?w1* (*elements* (*getM ?state′*))› ‹*?w1* ≠ *?w2*› ‹*?w2* = *literal*›
using *Cons*(*4*) *Cons*(*8*)
unfolding *InvariantConsistent-def*
unfolding *swapWatches-def*
by (*auto simp add*: *inconsistentCharacterization*)
moreover
have *elementLevel* (*opposite ?w2*) (*getM ?state′*) = *currentLevel* (*getM ?state′*)
using ‹*?w2* = *literal*›
using *Cons*(*5*) *Cons*(*8*)
unfolding *InvariantUniq-def*
unfolding *swapWatches-def*
by (*auto simp add*: *uniqAppendIff elementOnCurrentLevel*)
ultimately
show *?thesis*
using ‹*getWatch1 ?fState clause* = *getWatch1 ?state′ clause* ∧ *getWatch2 ?fState clause* = *getWatch2 ?state′ clause*›
using ‹*?w2* = *literal*› ‹*?w1* ≠ *?w2*›
using ‹*?w1 el* (*nth* (*getF state*) *clause*)›
using ‹*literalTrue ?w1* (*elements* (*getM ?state′*))›
unfolding *watchCharacterizationCondition-def*
using *elementLevelLeqCurrentLevel*[*of ?w1 getM ?state′*]
using *notifyWatchesLoopPreservedVariables*[*of ?state′ Wl′ literal clause* # *newWl*]
using ‹*InvariantWatchesEl* (*getF ?state′*) (*getWatch1 ?state′*) (*getWatch2 ?state′*)› ‹*getF ?state′* = *getF state*›
using *Cons*(*7*)
using *Cons*(*8*)
unfolding *swapWatches-def*
by (*auto simp add*: *Let-def*)
qed

ultimately
show *?thesis*
using *Cons(1)[of ?state′ clause # newWl]*
using *Cons(7) Cons(8)*
using ‹*uniq Wl′*›
using ‹*getWatch1 ?state′ clause = Some ?w1*›
using ‹*getWatch2 ?state′ clause = Some ?w2*›
using ‹*Some literal = getWatch1 state clause*›
using ‹*literalTrue ?w1 (elements (getM ?state′))*›
by (*simp add: Let-def*)
next
case *False*
show *?thesis*
proof (*cases getNonWatchedUnfalsifiedLiteral (nth (getF ?state′) clause) ?w1 ?w2 (getM ?state′)*)
case (*Some l′*)
hence *l′ el (nth (getF ?state′) clause) l′ ≠ ?w1 l′ ≠ ?w2 ¬ literalFalse l′ (elements (getM ?state′))*
using ‹*getWatch1 ?state′ clause = Some ?w1*›
using ‹*getWatch2 ?state′ clause = Some ?w2*›
using *getNonWatchedUnfalsifiedLiteralSomeCharacterization*
by *auto*

let *?state″ = setWatch2 clause l′ ?state′*
let *?fState = notifyWatches-loop literal Wl′ newWl ?state″*

from *Cons(2)*
have *InvariantWatchesEl (getF ?state″) (getWatch1 ?state″) (getWatch2 ?state″)*
using ‹*l′ el (nth (getF ?state′) clause)*›
unfolding *InvariantWatchesEl-def*
unfolding *swapWatches-def*
unfolding *setWatch2-def*
by *auto*
moreover
from *Cons(3)*
have *InvariantWatchesDiffer (getF ?state″) (getWatch1 ?state″) (getWatch2 ?state″)*
using ‹*l′ ≠ ?w1*›
using ‹*getWatch1 ?state′ clause = Some ?w1*›
using ‹*getWatch2 ?state′ clause = Some ?w2*›
unfolding *InvariantWatchesDiffer-def*
unfolding *swapWatches-def*
unfolding *setWatch2-def*
by *auto*
moreover
from *Cons(4)*
have *InvariantConsistent (getM ?state″)*
unfolding *InvariantConsistent-def*

unfolding *setWatch2-def*
unfolding *swapWatches-def*
by *simp*
moreover
from *Cons*(*5*)
have *InvariantUniq* (*getM ?state''*)
unfolding *InvariantUniq-def*
unfolding *setWatch2-def*
unfolding *swapWatches-def*
by *simp*
moreover
have *InvariantWatchCharacterization* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*) *M*
proof−
{
fix *c*::*nat* **and** *ww1*::*Literal* **and** *ww2*::*Literal*
assume *a*: *0* ≤ *c* ∧ *c* < *length* (*getF ?state''*) ∧ *Some ww1* = (*getWatch1 ?state'' c*) ∧ *Some ww2* = (*getWatch2 ?state'' c*)
assume *b*: *literalFalse ww1* (*elements M*)

have (∃ *l*. *l el* ((*getF ?state''*) ! *c*) ∧ *literalTrue l* (*elements M*) ∧ *elementLevel l M* ≤ *elementLevel* (*opposite ww1*) *M*) ∨
(∀ *l*. *l el* ((*getF ?state''*) ! *c*) ∧ *l* ≠ *ww1* ∧ *l* ≠ *ww2* ⟶
literalFalse l (*elements M*) ∧ *elementLevel* (*opposite l*) *M* ≤ *elementLevel* (*opposite ww1*) *M*)
proof (*cases c* = *clause*)
case *False*
thus *?thesis*
using *a* **and** *b*
using *Cons*(*6*)
unfolding *InvariantWatchCharacterization-def*
unfolding *watchCharacterizationCondition-def*
unfolding *swapWatches-def*
unfolding *setWatch2-def*
by *simp*
next
case *True*
with *a*
have *ww1* = *?w1* **and** *ww2* = *l'*
using ‹*getWatch1 ?state' clause* = *Some ?w1*›
using ‹*getWatch2 ?state' clause* = *Some ?w2*›[*THEN sym*]
unfolding *setWatch2-def*
unfolding *swapWatches-def*
by *auto*

have ¬ (∀ *l*. *l el* (*getF state* ! *clause*) ∧ *l* ≠ *?w1* ∧ *l* ≠ *?w2* ⟶ *literalFalse l* (*elements M*))
using *Cons*(*8*)

using ‹$l' \neq$ *?w1*› **and** ‹$l' \neq$ *?w2*› ‹l' *el* (*nth* (*getF ?state'*) *clause*)›
using ‹¬ *literalFalse* l' (*elements* (*getM ?state'*))›
using *a* **and** *b*
using ‹*c* = *clause*›
unfolding *swapWatches-def*
unfolding *setWatch2-def*
by *auto*
moreover
have (∃ *l*. *l el* (*getF state* ! *clause*) ∧ *literalTrue l* (*elements M*) ∧
elementLevel l M ≤ *elementLevel* (*opposite ?w1*) *M*) ∨
(∀ *l*. *l el* (*getF state* ! *clause*) ∧ *l* ≠ *?w1* ∧ *l* ≠ *?w2* ⟶ *literalFalse l* (*elements M*))
using *Cons*(*6*)
unfolding *InvariantWatchCharacterization-def*
unfolding *watchCharacterizationCondition-def*
using ‹*0* ≤ *clause* ∧ *clause* < *length* (*getF state*)›
using ‹*getWatch1 ?state' clause* = *Some ?w1*›[*THEN sym*]
using ‹*getWatch2 ?state' clause* = *Some ?w2*›[*THEN sym*]
using ‹*literalFalse ww1* (*elements M*)›
using ‹*ww1* = *?w1*›
unfolding *setWatch2-def*
unfolding *swapWatches-def*
by *auto*
ultimately
show *?thesis*
using ‹*ww1* = *?w1*›
using ‹*c* = *clause*›
unfolding *setWatch2-def*
unfolding *swapWatches-def*
by *auto*
qed
}
moreover
{
fix *c*::*nat* **and** *ww1*::*Literal* **and** *ww2*::*Literal*
assume *a*: *0* ≤ *c* ∧ *c* < *length* (*getF ?state''*) ∧ *Some ww1* = (*getWatch1 ?state'' c*) ∧ *Some ww2* = (*getWatch2 ?state'' c*)
assume *b*: *literalFalse ww2* (*elements M*)

have (∃ *l*. *l el* ((*getF ?state''*) ! *c*) ∧ *literalTrue l* (*elements M*) ∧ *elementLevel l M* ≤ *elementLevel* (*opposite ww2*) *M*) ∨
(∀ *l*. *l el* ((*getF ?state''*) ! *c*) ∧ *l* ≠ *ww1* ∧ *l* ≠ *ww2* ⟶
literalFalse l (*elements M*) ∧ *elementLevel* (*opposite l*) *M* ≤ *elementLevel* (*opposite ww2*) *M*)
proof (*cases c* = *clause*)

```
        case False
        thus ?thesis
          using a and b
          using Cons(6)
          unfolding InvariantWatchCharacterization-def
          unfolding watchCharacterizationCondition-def
          unfolding swapWatches-def
          unfolding setWatch2-def
          by auto
      next
        case True
        with a
        have ww1 = ?w1 and ww2 = l′
          using ‹getWatch1 ?state′ clause = Some ?w1›
            using ‹getWatch2 ?state′ clause = Some ?w2›[THEN
sym]
          unfolding setWatch2-def
          unfolding swapWatches-def
          by auto
        with ‹¬ literalFalse l′ (elements (getM ?state′))› b
          Cons(8)
        have False
          unfolding swapWatches-def
          by simp
        thus ?thesis
          by simp
      qed
    }
    ultimately
    show ?thesis
      unfolding InvariantWatchCharacterization-def
      unfolding watchCharacterizationCondition-def
      by blast
  qed
  moreover
   have ∀  (c::nat). c ∈ set Wl′ ⟶ Some literal = (getWatch1
?state′′ c)  ∨ Some literal = (getWatch2 ?state′′ c)
    using Cons(10)
    using ‹clause ∉ set Wl′›
    using swapWatchesEffect[of clause state]
    unfolding setWatch2-def
    by simp
  moreover
  have getM ?state′′ = getM state
    getF ?state′′ = getF state
    unfolding swapWatches-def
    unfolding setWatch2-def
    by auto
  moreover
```

have *getWatch1 ?state″ clause = Some ?w1 getWatch2 ?state″ clause = Some l′*
using ‹*getWatch1 ?state′ clause = Some ?w1*›
unfolding *swapWatches-def*
unfolding *setWatch2-def*
by *auto*
hence *getWatch1 ?fState clause = getWatch1 ?state″ clause* ∧ *getWatch2 ?fState clause = Some l′*
using ‹*clause* ∉ *set Wl′*›
using ‹*InvariantWatchesEl* (*getF ?state″*) (*getWatch1 ?state″*) (*getWatch2 ?state″*)› ‹*getF ?state″ = getF state*›
using *Cons*(*7*)
using *notifyWatchesLoopPreservedWatches*[*of ?state″ Wl′ literal newWl*]
by (*simp add*: *Let-def*)
moreover
have *watchCharacterizationCondition ?w1 l′* (*getM ?fState*) (*getF ?fState* ! *clause*) ∧
watchCharacterizationCondition l′ ?w1 (*getM ?fState*) (*getF ?fState* ! *clause*)
proof−
have (*getM ?fState*) = (*getM state*) (*getF ?fState*) = (*getF state*)
using *notifyWatchesLoopPreservedVariables*[*of ?state″ Wl′ literal newWl*]
using ‹*InvariantWatchesEl* (*getF ?state″*) (*getWatch1 ?state″*) (*getWatch2 ?state″*)› ‹*getF ?state″ = getF state*›
using *Cons*(*7*)
unfolding *setWatch2-def*
unfolding *swapWatches-def*
by (*auto simp add*: *Let-def*)

have *literalFalse ?w1* (*elements M*) ⟶
(∃ *l*. *l el* (*nth* (*getF ?state″*) *clause*) ∧ *literalTrue l* (*elements M*) ∧ *elementLevel l M* ≤ *elementLevel* (*opposite ?w1*) *M*)
proof
assume *literalFalse ?w1* (*elements M*)
show ∃ *l*. *l el* (*nth* (*getF ?state″*) *clause*) ∧ *literalTrue l* (*elements M*) ∧ *elementLevel l M* ≤ *elementLevel* (*opposite ?w1*) *M*
proof−
have ¬ (∀ *l*. *l el* (*nth* (*getF state*) *clause*) ∧ *l* ≠ *?w1* ∧ *l* ≠ *?w2* ⟶ *literalFalse l* (*elements M*))
using ‹*l′ el* (*nth* (*getF ?state′*) *clause*)› ‹*l′* ≠ *?w1*› ‹*l′* ≠ *?w2*› ‹¬ *literalFalse l′* (*elements* (*getM ?state′*))›
using *Cons*(*8*)
unfolding *swapWatches-def*
by *auto*

from ‹*literalFalse ?w1* (*elements M*)› *Cons*(*6*)

have
($\exists$ *l*. *l el* (*getF state* ! *clause*) $\wedge$ *literalTrue l* (*elements M*) $\wedge$ *elementLevel l M* $\leq$ *elementLevel* (*opposite ?w1*) *M*) $\vee$
($\forall$ *l*. *l el* (*getF state* ! *clause*) $\wedge$ *l* $\neq$ *?w1* $\wedge$ *l* $\neq$ *?w2* $\longrightarrow$
literalFalse l (*elements M*) $\wedge$ *elementLevel* (*opposite l*) *M* $\leq$ *elementLevel* (*opposite ?w1*) *M*)
using ‹*0* $\leq$ *clause* $\wedge$ *clause* < *length* (*getF state*)›
using ‹*getWatch1 ?state′ clause* = *Some ?w1*›[*THEN sym*]
using ‹*getWatch2 ?state′ clause* = *Some ?w2*›[*THEN sym*]
unfolding *InvariantWatchCharacterization-def*
unfolding *watchCharacterizationCondition-def*
unfolding *swapWatches-def*
by *simp*
with ‹$\neg$ ($\forall$ *l*. *l el* (*nth* (*getF state*) *clause*) $\wedge$ *l* $\neq$ *?w1* $\wedge$ *l* $\neq$ *?w2* $\longrightarrow$ *literalFalse l* (*elements M*))›
have $\exists$ *l*. *l el* (*getF state* ! *clause*) $\wedge$ *literalTrue l* (*elements M*) $\wedge$ *elementLevel l M* $\leq$ *elementLevel* (*opposite ?w1*) *M*
by *auto*
thus *?thesis*
unfolding *setWatch2-def*
unfolding *swapWatches-def*
by *simp*
qed
qed

have *watchCharacterizationCondition l′ ?w1* (*getM ?fState*) (*getF ?fState* ! *clause*)
using ‹$\neg$ *literalFalse l′* (*elements* (*getM ?state′*))›
using ‹*getM ?fState* = *getM state*›
unfolding *swapWatches-def*
unfolding *watchCharacterizationCondition-def*
by *simp*
moreover
have *watchCharacterizationCondition ?w1 l′* (*getM ?fState*) (*getF ?fState* ! *clause*)
proof (*cases literalFalse ?w1* (*elements* (*getM ?fState*)))
case *True*
hence *literalFalse ?w1* (*elements M*)
using *notifyWatchesLoopPreservedVariables*[*of ?state″ Wl′ literal newWl*]
using ‹*InvariantWatchesEl* (*getF ?state″*) (*getWatch1 ?state″*) (*getWatch2 ?state″*)› ‹*getF ?state″* = *getF state*›
using *Cons*(*7*) *Cons*(*8*)
using ‹*?w1* $\neq$ *?w2*› ‹*?w2* = *literal*›
unfolding *setWatch2-def*
unfolding *swapWatches-def*
by (*simp add*: *Let-def*)

```
      with ‹literalFalse ?w1 (elements M) ⟶
      (∃ l. l el (nth (getF ?state'') clause) ∧ literalTrue l (elements
M) ∧  elementLevel l M ≤ elementLevel (opposite ?w1) M)›
      obtain l::Literal
        where l el (nth (getF ?state'') clause) and
        literalTrue l (elements M) and
        elementLevel l M ≤ elementLevel (opposite ?w1) M
        by auto
      hence elementLevel l (getM state) ≤ elementLevel (opposite
?w1) (getM state)
        using Cons(8)
     using ‹literalTrue l (elements M)› ‹literalFalse ?w1 (elements
M)›
     using elementLevelAppend[of l M [(opposite literal, decision)]]
        using elementLevelAppend[of opposite ?w1 M [(opposite
literal, decision)]]
        by auto
      thus ?thesis
          using ‹l el (nth (getF ?state'') clause)› ‹literalTrue l
(elements M)›
        using ‹getM ?fState = getM state› ‹getF ?fState = getF
state› ‹getM ?state'' = getM state› ‹getF ?state'' = getF state›
        using Cons(8)
        unfolding watchCharacterizationCondition-def
        by auto
    next
      case False
      thus ?thesis
        unfolding watchCharacterizationCondition-def
        by simp
    qed
    ultimately
    show ?thesis
      by simp
  qed
  ultimately
  show ?thesis
    using Cons(1)[of ?state'' newWl]
    using Cons(7) Cons(8)
    using ‹getWatch1 ?state' clause = Some ?w1›
    using ‹getWatch2 ?state' clause = Some ?w2›
    using ‹Some literal = getWatch1 state clause›
    using ‹¬ literalTrue ?w1 (elements (getM ?state'))›
    using ‹getWatch1 ?state'' clause = Some ?w1›
    using ‹getWatch2 ?state'' clause = Some l'›
    using Some
    using ‹uniq Wl'›
    by (simp add: Let-def)
next
```

case *None*
show *?thesis*
proof (*cases literalFalse ?w1* (*elements* (*getM ?state'*)))
case *True*
let *?state''* = *?state'*⦇*getConflictFlag* := *True*, *getConflictClause* := *clause*⦈
let *?fState* = *notifyWatches-loop literal Wl'* (*clause* # *newWl*) *?state''*

from *Cons*(*2*)
have *InvariantWatchesEl* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)
unfolding *InvariantWatchesEl-def*
unfolding *swapWatches-def*
by *auto*
moreover
from *Cons*(*3*)
have *InvariantWatchesDiffer* (*getF ?state'*) (*getWatch1 ?state'*) (*getWatch2 ?state'*)
unfolding *InvariantWatchesDiffer-def*
unfolding *swapWatches-def*
by *auto*
moreover
from *Cons*(*4*)
have *InvariantConsistent* (*getM ?state'*)
unfolding *InvariantConsistent-def*
unfolding *swapWatches-def*
by *simp*
moreover
from *Cons*(*5*)
have *InvariantUniq* (*getM ?state'*)
unfolding *InvariantUniq-def*
unfolding *swapWatches-def*
by *simp*
moreover
from *Cons*(*6*)
have *InvariantWatchCharacterization* (*getF ?state'*) (*getWatch1 ?state'*) (*getWatch2 ?state'*) *M*
unfolding *swapWatches-def*
unfolding *InvariantWatchCharacterization-def*
unfolding *watchCharacterizationCondition-def*
by *simp*
moreover
have ∀ (*c*::*nat*). *c* ∈ *set Wl'* ⟶ *Some literal* = (*getWatch1 ?state'' c*) ∨ *Some literal* = (*getWatch2 ?state'' c*)
using *Cons*(*10*)
using ‹*clause* ∉ *set Wl'*›
using *swapWatchesEffect*[*of clause state*]
by *simp*

moreover
have *getM ?state'' = getM state*
getF ?state'' = getF state
unfolding *swapWatches-def*
by *auto*
moreover
have *getWatch1 ?fState clause = getWatch1 ?state'' clause* ∧ *getWatch2 ?fState clause = getWatch2 ?state'' clause*
using ‹*clause* ∉ *set Wl'*›
using ‹*InvariantWatchesEl* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)› ‹*getF ?state'' = getF state*›
using *Cons*(*7*)
using *notifyWatchesLoopPreservedWatches*[*of ?state'' Wl' literal clause # newWl*]
by (*simp add*: *Let-def*)
moreover
have *literalFalse ?w1* (*elements M*)
using ‹*literalFalse ?w1* (*elements* (*getM ?state'*))›
‹*?w1* ≠ *?w2*› ‹*?w2 = literal*› *Cons*(*8*)
unfolding *swapWatches-def*
by *auto*

have ¬ *literalTrue ?w2* (*elements M*)
using *Cons*(*4*)
using *Cons*(*8*)
using ‹*?w2 = literal*›
using *inconsistentCharacterization*[*of elements M* @ [*opposite literal*]]
unfolding *InvariantConsistent-def*
by *force*

have ∗: ∀ *l*. *l el* (*nth* (*getF state*) *clause*) ∧ *l* ≠ *?w1* ∧ *l* ≠ *?w2* ⟶
literalFalse l (*elements M*) ∧ *elementLevel* (*opposite l*) *M* ≤ *elementLevel* (*opposite ?w1*) *M*
proof−
have ¬ (∃ *l*. *l el* (*nth* (*getF state*) *clause*) ∧ *literalTrue l* (*elements M*))
proof
assume ∃ *l*. *l el* (*nth* (*getF state*) *clause*) ∧ *literalTrue l* (*elements M*)
show *False*
proof−
from ‹∃ *l*. *l el* (*nth* (*getF state*) *clause*) ∧ *literalTrue l* (*elements M*)›
obtain *l*
where *l el* (*nth* (*getF state*) *clause*) *literalTrue l* (*elements M*)
by *auto*

hence *l* ≠ *?w1 l* ≠ *?w2*
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state'*))›
using ‹¬ *literalTrue ?w2* (*elements M*)›
unfolding *swapWatches-def*
using *Cons*(*8*)
by *auto*
with ‹*l el* (*nth* (*getF state*) *clause*)›
have *literalFalse l* (*elements* (*getM ?state'*))
using ‹*getWatch1 ?state' clause* = *Some ?w1*›
using ‹*getWatch2 ?state' clause* = *Some ?w2*›
using *None*
using *getNonWatchedUnfalsifiedLiteralNoneCharacterization*[*of nth* (*getF ?state'*) *clause ?w1 ?w2 getM ?state'*]
unfolding *swapWatches-def*
by *simp*
with ‹*l* ≠ *?w2*› ‹*?w2* = *literal*› *Cons*(*8*)
have *literalFalse l* (*elements M*)
unfolding *swapWatches-def*
by *simp*
with *Cons*(*4*) ‹*literalTrue l* (*elements M*)›
show *?thesis*
unfolding *InvariantConsistent-def*
using *Cons*(*8*)
by (*auto simp add*: *inconsistentCharacterization*)
qed
qed
with ‹*InvariantWatchCharacterization* (*getF state*) (*getWatch1 state*) (*getWatch2 state*) *M*›
show *?thesis*
unfolding *InvariantWatchCharacterization-def*
using ‹*literalFalse ?w1* (*elements M*)›
using ‹*getWatch1 ?state' clause* = *Some ?w1*›[*THEN sym*]
using ‹*getWatch2 ?state' clause* = *Some ?w2*›[*THEN sym*]
using ‹*0* ≤ *clause* ∧ *clause* < *length* (*getF state*)›
unfolding *watchCharacterizationCondition-def*
unfolding *swapWatches-def*
by (*simp*) (*blast*)
qed

have ∗∗: ∀ *l*. *l el* (*nth* (*getF ?state''*) *clause*) ∧ *l* ≠ *?w1* ∧ *l* ≠ *?w2* ⟶
literalFalse l (*elements* (*getM ?state''*)) ∧
elementLevel (*opposite l*) (*getM ?state''*) ≤ *elementLevel* (*opposite ?w1*) (*getM ?state''*)
proof−
{
fix *l*::*Literal*
assume *l el* (*nth* (*getF ?state''*) *clause*) ∧ *l* ≠ *?w1* ∧ *l* ≠

?w2

have *literalFalse l* (*elements* (*getM ?state''*)) ∧
elementLevel (*opposite l*) (*getM ?state''*) ≤ *elementLevel*
(*opposite ?w1*) (*getM ?state''*)
proof−
from ∗ ‹*l el* (*nth* (*getF ?state''*) *clause*) ∧ *l* ≠ *?w1* ∧ *l*
≠ *?w2*›
have *literalFalse l* (*elements M*) *elementLevel* (*opposite*
l) *M* ≤ *elementLevel* (*opposite ?w1*) *M*
unfolding *swapWatches-def*
by *auto*
thus *?thesis*
using *elementLevelAppend*[*of opposite l M* [(*opposite*
literal, *decision*)]]
using ‹*literalFalse ?w1* (*elements M*)›
using *elementLevelAppend*[*of opposite ?w1 M* [(*opposite*
literal, *decision*)]]
using *Cons*(*8*)
unfolding *swapWatches-def*
by *simp*
qed
}
thus *?thesis*
by *simp*
qed

have (*getM ?fState*) = (*getM state*) (*getF ?fState*) = (*getF*
state)
using *notifyWatchesLoopPreservedVariables*[*of ?state'' Wl'*
literal clause # *newWl*]
using ‹*InvariantWatchesEl* (*getF ?state''*) (*getWatch1 ?state''*)
(*getWatch2 ?state''*)› ‹*getF ?state''* = *getF state*›
using *Cons*(*7*)
unfolding *swapWatches-def*
by (*auto simp add*: *Let-def*)
hence ∀ *l*. *l el* (*nth* (*getF ?fState*) *clause*) ∧ *l* ≠ *?w1* ∧ *l* ≠
?w2 ⟶
literalFalse l (*elements* (*getM ?fState*)) ∧
elementLevel (*opposite l*) (*getM ?fState*) ≤ *elementLevel*
(*opposite ?w1*) (*getM ?fState*)
using ∗∗
using ‹*getM ?state''* = *getM state*›
using ‹*getF ?state''* = *getF state*›
by *simp*
moreover
have ∀ *l*. *literalFalse l* (*elements* (*getM ?fState*)) ⟶
elementLevel (*opposite l*) (*getM ?fState*) ≤ *elementLevel*

(*opposite ?w2*) (*getM ?fState*)
proof−
have *elementLevel* (*opposite ?w2*) (*getM ?fState*) = *currentLevel* (*getM ?fState*)
using *Cons*(*8*)
using ‹(*getM ?fState*) = (*getM state*)›
using ‹¬ *literalFalse ?w2* (*elements M*)›
using ‹*?w2* = *literal*›
using *elementOnCurrentLevel*[*of opposite ?w2 M decision*]
by *simp*
thus *?thesis*
by (*simp add*: *elementLevelLeqCurrentLevel*)
qed
ultimately
show *?thesis*
using *Cons*(*1*)[*of ?state'' clause # newWl*]
using *Cons*(*7*) *Cons*(*8*)
using ‹*getWatch1 ?state' clause* = *Some ?w1*›
using ‹*getWatch2 ?state' clause* = *Some ?w2*›
using ‹*Some literal* = *getWatch1 state clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state'*))›
using *None*
using ‹*literalFalse ?w1* (*elements* (*getM ?state'*))›
using ‹*uniq Wl'*›
unfolding *watchCharacterizationCondition-def*
by (*simp add*: *Let-def*)
next
case *False*

let *?state''* = *setReason ?w1 clause* (*?state'*⦇*getQ* := (*if ?w1 el* (*getQ ?state'*) *then* (*getQ ?state'*) *else* (*getQ ?state'*) @ [*?w1*])⦈)
let *?fState* = *notifyWatches-loop literal Wl'* (*clause # newWl*) *?state''*

from *Cons*(*2*)
have *InvariantWatchesEl* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)
unfolding *InvariantWatchesEl-def*
unfolding *setReason-def*
unfolding *swapWatches-def*
by *auto*
moreover
from *Cons*(*3*)
have *InvariantWatchesDiffer* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)
unfolding *InvariantWatchesDiffer-def*
unfolding *setReason-def*
unfolding *swapWatches-def*
by *auto*

moreover
from *Cons*(*4*)
have *InvariantConsistent* (*getM ?state''*)
unfolding *InvariantConsistent-def*
unfolding *setReason-def*
unfolding *swapWatches-def*
by *simp*
moreover
from *Cons*(*5*)
have *InvariantUniq* (*getM ?state''*)
unfolding *InvariantUniq-def*
unfolding *setReason-def*
unfolding *swapWatches-def*
by *simp*
moreover
from *Cons*(*6*)
have *InvariantWatchCharacterization* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*) *M*
unfolding *swapWatches-def*
unfolding *setReason-def*
unfolding *InvariantWatchCharacterization-def*
unfolding *watchCharacterizationCondition-def*
by *simp*
moreover
have ∀ (*c::nat*). *c* ∈ *set Wl'* ⟶ *Some literal* = (*getWatch1 ?state'' c*) ∨ *Some literal* = (*getWatch2 ?state'' c*)
using *Cons*(*10*)
using ‹*clause* ∉ *set Wl'*›
using *swapWatchesEffect*[*of clause state*]
unfolding *setReason-def*
by *simp*
moreover
have *getM ?state''* = *getM state*
getF ?state'' = *getF state*
unfolding *setReason-def*
unfolding *swapWatches-def*
by *auto*
moreover
have *getWatch1 ?state'' clause* = *Some ?w1 getWatch2 ?state'' clause* = *Some ?w2*
using ‹*getWatch1 ?state' clause* = *Some ?w1*›
using ‹*getWatch2 ?state' clause* = *Some ?w2*›
unfolding *setReason-def*
unfolding *swapWatches-def*
by *auto*
moreover
have *getWatch1 ?fState clause* = *Some ?w1 getWatch2 ?fState clause* = *Some ?w2*
using ‹*getWatch1 ?state'' clause* = *Some ?w1*› ‹*getWatch2*

?state'' clause = Some ?w2›
using ‹*clause* ∉ *set Wl'*›
using ‹*InvariantWatchesEl* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)› ‹*getF ?state'' = getF state*›
using *Cons*(*7*)
using *notifyWatchesLoopPreservedWatches*[*of ?state'' Wl' literal clause # newWl*]
by (*auto simp add*: *Let-def*)
moreover
have (*getM ?fState*) = (*getM state*) (*getF ?fState*) = (*getF state*)
using *notifyWatchesLoopPreservedVariables*[*of ?state'' Wl' literal clause # newWl*]
using ‹*InvariantWatchesEl* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)› ‹*getF ?state'' = getF state*›
using *Cons*(*7*)
unfolding *setReason-def*
unfolding *swapWatches-def*
by (*auto simp add*: *Let-def*)
ultimately
have ∀ *c*. *c* ∈ *set Wl'* ⟶ (∀ *w1 w2*. *Some w1* = *getWatch1 ?fState c* ∧ *Some w2* = *getWatch2 ?fState c* ⟶
watchCharacterizationCondition w1 w2 (*getM ?fState*) (*getF ?fState* ! *c*) ∧
watchCharacterizationCondition w2 w1 (*getM ?fState*) (*getF ?fState* ! *c*)) **and**
?fState = *notifyWatches-loop literal* (*clause* # *Wl'*) *newWl state*
using *Cons*(*1*)[*of ?state'' clause # newWl*]
using *Cons*(*7*) *Cons*(*8*)
using ‹*getWatch1 ?state' clause = Some ?w1*›
using ‹*getWatch2 ?state' clause = Some ?w2*›
using ‹*Some literal = getWatch1 state clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state'*))›
using *None*
using ‹¬ *literalFalse ?w1* (*elements* (*getM ?state'*))›
using ‹*uniq Wl'*›
by (*auto simp add*: *Let-def*)
moreover
have ∗: ∀ *l*. *l el* (*nth* (*getF ?state''*) *clause*) ∧ *l* ≠ *?w1* ∧ *l* ≠ *?w2* ⟶ *literalFalse l* (*elements* (*getM ?state''*))
using *None*
using ‹*getWatch1 ?state' clause = Some ?w1*›
using ‹*getWatch2 ?state' clause = Some ?w2*›
using *getNonWatchedUnfalsifiedLiteralNoneCharacterization*[*of nth* (*getF ?state'*) *clause ?w1 ?w2 getM ?state'*]
using *Cons*(*8*)
unfolding *setReason-def*
unfolding *swapWatches-def*

by *auto*

have∗∗: ∀ *l*. *l el* (*nth* (*getF ?fState*) *clause*) ∧ *l* ≠ *?w1* ∧ *l* ≠ *?w2* ⟶ *literalFalse l* (*elements* (*getM ?fState*))
using ‹(*getM ?fState*) = (*getM state*)› ‹(*getF ?fState*) = (*getF state*)›
using ∗
using ‹*getM ?state''* = *getM state*›
using ‹*getF ?state''* = *getF state*›
unfolding *swapWatches-def*
by *auto*

have ∗∗∗: ∀ *l*. *literalFalse l* (*elements* (*getM ?fState*)) ⟶
elementLevel (*opposite l*) (*getM ?fState*) ≤ *elementLevel* (*opposite ?w2*) (*getM ?fState*)
proof−
have *elementLevel* (*opposite ?w2*) (*getM ?fState*) = *currentLevel* (*getM ?fState*)
using *Cons*(*8*)
using ‹(*getM ?fState*) = (*getM state*)›
using ‹¬ *literalFalse ?w2* (*elements M*)›
using ‹*?w2* = *literal*›
using *elementOnCurrentLevel*[*of opposite ?w2 M decision*]
by *simp*
thus *?thesis*
by (*simp add*: *elementLevelLeqCurrentLevel*)
qed

have (∀ *w1 w2*. *Some w1* = *getWatch1 ?fState clause* ∧ *Some w2* = *getWatch2 ?fState clause* ⟶
watchCharacterizationCondition w1 w2 (*getM ?fState*) (*getF ?fState* ! *clause*) ∧
watchCharacterizationCondition w2 w1 (*getM ?fState*) (*getF ?fState* ! *clause*))
proof−
{
fix *w1 w2*
assume *Some w1* = *getWatch1 ?fState clause* ∧ *Some w2* = *getWatch2 ?fState clause*
hence *w1* = *?w1 w2* = *?w2*
using ‹*getWatch1 ?fState clause* = *Some ?w1*›
using ‹*getWatch2 ?fState clause* = *Some ?w2*›
by *auto*
hence *watchCharacterizationCondition w1 w2* (*getM ?fState*) (*getF ?fState* ! *clause*) ∧
watchCharacterizationCondition w2 w1 (*getM ?fState*) (*getF ?fState* ! *clause*)
unfolding *watchCharacterizationCondition-def*
using ∗∗ ∗∗∗

unfolding *watchCharacterizationCondition-def*
using ‹(*getM ?fState*) = (*getM state*)› ‹(*getF ?fState*) = (*getF state*)›
using ‹¬ *literalFalse ?w1* (*elements* (*getM ?state'*))›
unfolding *swapWatches-def*
by *simp*
}
thus *?thesis*
by *auto*
qed
ultimately
show *?thesis*
by *simp*
qed
qed
qed
next
case *False*
let *?state'* = *state*
let *?w1* = *wa*
have *getWatch1 ?state' clause* = *Some ?w1*
using ‹*getWatch1 state clause* = *Some wa*›
by *auto*
let *?w2* = *wb*
have *getWatch2 ?state' clause* = *Some ?w2*
using ‹*getWatch2 state clause* = *Some wb*›
by *auto*

from ‹¬ *Some literal* = *getWatch1 state clause*›
‹∀ (*c::nat*). *c* ∈ *set* (*clause* # *Wl'*) ⟶ *Some literal* = (*getWatch1 state c*) ∨ *Some literal* = (*getWatch2 state c*)›
have *Some literal* = *getWatch2 state clause*
by *auto*
hence *?w2* = *literal*
using ‹*getWatch2 ?state' clause* = *Some ?w2*›
by *simp*
hence *literalFalse ?w2* (*elements* (*getM state*))
using *Cons*(*8*)
by *simp*

from ‹*InvariantWatchesEl* (*getF state*) (*getWatch1 state*) (*getWatch2 state*)›
have *?w1 el* (*nth* (*getF state*) *clause*) *?w2 el* (*nth* (*getF state*) *clause*)
using ‹*getWatch1 ?state' clause* = *Some ?w1*›
using ‹*getWatch2 ?state' clause* = *Some ?w2*›
using ‹*0* ≤ *clause* ∧ *clause* < *length* (*getF state*)›
unfolding *InvariantWatchesEl-def*
by *auto*

from ‹*InvariantWatchesDiffer* (*getF state*) (*getWatch1 state*) (*getWatch2 state*)›
have *?w1* ≠ *?w2*
using ‹*getWatch1 ?state′ clause* = *Some ?w1*›
using ‹*getWatch2 ?state′ clause* = *Some ?w2*›
using ‹*0* ≤ *clause* ∧ *clause* < *length* (*getF state*)›
unfolding *InvariantWatchesDiffer-def*
by *auto*

have ¬ *literalFalse ?w2* (*elements M*)
using ‹*?w2* = *literal*›
using *Cons*(*5*)
using *Cons*(*8*)
unfolding *InvariantUniq-def*
by (*simp add*: *uniqAppendIff*)

show *?thesis*
proof (*cases literalTrue ?w1* (*elements* (*getM ?state′*)))
case *True*

let *?fState* = *notifyWatches-loop literal Wl′* (*clause* # *newWl*) *?state′*

have *getWatch1 ?fState clause* = *getWatch1 ?state′ clause* ∧ *getWatch2 ?fState clause* = *getWatch2 ?state′ clause*
using ‹*clause* ∉ *set Wl′*›
using *Cons*(*2*)
using *Cons*(*7*)
using *notifyWatchesLoopPreservedWatches*[*of ?state′ Wl′ literal clause* # *newWl*]
by (*simp add*: *Let-def*)
moreover
have *watchCharacterizationCondition ?w1 ?w2* (*getM ?fState*) (*getF ?fState* ! *clause*) ∧
watchCharacterizationCondition ?w2 ?w1 (*getM ?fState*) (*getF ?fState* ! *clause*)
proof−
have (*getM ?fState*) = (*getM state*) ∧ (*getF ?fState* = *getF state*)
using *notifyWatchesLoopPreservedVariables*[*of ?state′ Wl′ literal clause* # *newWl*]
using *Cons*(*2*)
using *Cons*(*7*)
by (*simp add*: *Let-def*)
moreover
have ¬ *literalFalse ?w1* (*elements M*)
using ‹*literalTrue ?w1* (*elements* (*getM ?state′*))› ‹*?w1* ≠ *?w2*› ‹*?w2* = *literal*›

using *Cons*(*4*) *Cons*(*8*)
unfolding *InvariantConsistent-def*
by (*auto simp add*: *inconsistentCharacterization*)
moreover
have *elementLevel* (*opposite ?w2*) (*getM ?state′*) = *currentLevel* (*getM ?state′*)
using ‹*?w2* = *literal*›
using *Cons*(*5*) *Cons*(*8*)
unfolding *InvariantUniq-def*
by (*auto simp add*: *uniqAppendIff elementOnCurrentLevel*)
ultimately
show *?thesis*
using ‹*getWatch1 ?fState clause* = *getWatch1 ?state′ clause* ∧ *getWatch2 ?fState clause* = *getWatch2 ?state′ clause*›
using ‹*?w2* = *literal*› ‹*?w1* ≠ *?w2*›
using ‹*?w1 el* (*nth* (*getF state*) *clause*)›
using ‹*literalTrue ?w1* (*elements* (*getM ?state′*))›
unfolding *watchCharacterizationCondition-def*
using *elementLevelLeqCurrentLevel*[*of ?w1 getM ?state′*]
using *notifyWatchesLoopPreservedVariables*[*of ?state′ Wl′ literal clause # newWl*]
using ‹*InvariantWatchesEl* (*getF state*) (*getWatch1 state*) (*getWatch2 state*)›
using *Cons*(*7*)
using *Cons*(*8*)
by (*auto simp add*: *Let-def*)
qed
ultimately
show *?thesis*
using *assms*
using *Cons*(*1*)[*of ?state′ clause # newWl*]
using *Cons*(*2*) *Cons*(*3*) *Cons*(*4*) *Cons*(*5*) *Cons*(*6*) *Cons*(*7*) *Cons*(*8*) *Cons*(*9*) *Cons*(*10*)
using ‹*uniq Wl′*›
using ‹*getWatch1 ?state′ clause* = *Some ?w1*›
using ‹*getWatch2 ?state′ clause* = *Some ?w2*›
using ‹*Some literal* = *getWatch2 state clause*›
using ‹*literalTrue ?w1* (*elements* (*getM ?state′*))›
using ‹*?w1* ≠ *?w2*›
by (*simp add*:*Let-def*)
next
case *False*
show *?thesis*
proof (*cases getNonWatchedUnfalsifiedLiteral* (*nth* (*getF ?state′*) *clause*) *?w1 ?w2* (*getM ?state′*))
case (*Some l′*)
hence *l′ el* (*nth* (*getF ?state′*) *clause*) *l′* ≠ *?w1 l′* ≠ *?w2* ¬ *literalFalse l′* (*elements* (*getM ?state′*))
using ‹*getWatch1 ?state′ clause* = *Some ?w1*›

using ‹*getWatch2 ?state′ clause = Some ?w2*›
using *getNonWatchedUnfalsifiedLiteralSomeCharacterization*
by *auto*

let *?state″ = setWatch2 clause l′ ?state′*
let *?fState = notifyWatches-loop literal Wl′ newWl ?state″*

from *Cons(2)*
have *InvariantWatchesEl* (*getF ?state″*) (*getWatch1 ?state″*) (*getWatch2 ?state″*)
using ‹*l′ el* (*nth* (*getF ?state′*) *clause*)›
unfolding *InvariantWatchesEl-def*
unfolding *setWatch2-def*
by *auto*
moreover
from *Cons(3)*
have *InvariantWatchesDiffer* (*getF ?state″*) (*getWatch1 ?state″*) (*getWatch2 ?state″*)
using ‹*l′* ≠ *?w1*›
using ‹*getWatch1 ?state′ clause = Some ?w1*›
using ‹*getWatch2 ?state′ clause = Some ?w2*›
unfolding *InvariantWatchesDiffer-def*
unfolding *setWatch2-def*
by *auto*
moreover
from *Cons(4)*
have *InvariantConsistent* (*getM ?state″*)
unfolding *InvariantConsistent-def*
unfolding *setWatch2-def*
by *simp*
moreover
from *Cons(5)*
have *InvariantUniq* (*getM ?state″*)
unfolding *InvariantUniq-def*
unfolding *setWatch2-def*
by *simp*
moreover
have *InvariantWatchCharacterization* (*getF ?state″*) (*getWatch1 ?state″*) (*getWatch2 ?state″*) *M*
proof−
{
fix *c*::*nat* **and** *ww1*::*Literal* **and** *ww2*::*Literal*
assume *a*: *0* ≤ *c* ∧ *c* < *length* (*getF ?state″*) ∧ *Some ww1* = (*getWatch1 ?state″ c*) ∧ *Some ww2* = (*getWatch2 ?state″ c*)
assume *b*: *literalFalse ww1* (*elements M*)

have (∃ *l. l el* ((*getF ?state″*) ! *c*) ∧ *literalTrue l* (*elements M*) ∧ *elementLevel l M* ≤ *elementLevel* (*opposite ww1*) *M*) ∨
(∀ *l. l el* ((*getF ?state″*) ! *c*) ∧ *l* ≠ *ww1* ∧ *l* ≠ *ww2* ⟶

literalFalse l (elements M) ∧ elementLevel (opposite l) M ≤ elementLevel (opposite ww1) M)
proof (cases c = clause)
case False
thus ?thesis
using a **and** b
using Cons(6)
unfolding InvariantWatchCharacterization-def
unfolding watchCharacterizationCondition-def
unfolding setWatch2-def
by simp
next
case True
with a
have ww1 = ?w1 **and** ww2 = l′
using ‹getWatch1 ?state′ clause = Some ?w1›
using ‹getWatch2 ?state′ clause = Some ?w2›[THEN sym]
unfolding setWatch2-def
by auto

have ¬ (∀ l. l el (getF state ! clause) ∧ l ≠ ?w1 ∧ l ≠ ?w2 ⟶ literalFalse l (elements M))
using Cons(8)
using ‹l′ ≠ ?w1› **and** ‹l′ ≠ ?w2› ‹l′ el (nth (getF ?state′) clause)›
using ‹¬ literalFalse l′ (elements (getM ?state′))›
using a **and** b
using ‹c = clause›
unfolding setWatch2-def
by auto
moreover
have (∃ l. l el (getF state ! clause) ∧ literalTrue l (elements M) ∧
elementLevel l M ≤ elementLevel (opposite ?w1) M) ∨
(∀ l. l el (getF state ! clause) ∧ l ≠ ?w1 ∧ l ≠ ?w2 ⟶ literalFalse l (elements M))
using Cons(6)
unfolding InvariantWatchCharacterization-def
unfolding watchCharacterizationCondition-def
using ‹0 ≤ clause ∧ clause < length (getF state)›
using ‹getWatch1 ?state′ clause = Some ?w1›[THEN sym]
using ‹getWatch2 ?state′ clause = Some ?w2›[THEN sym]
using ‹literalFalse ww1 (elements M)›
using ‹ww1 = ?w1›
unfolding setWatch2-def
by auto

ultimately
show *?thesis*
using ‹*ww1* = *?w1*›
using ‹*c* = *clause*›
unfolding *setWatch2-def*
by *auto*
qed
}
moreover
{
fix *c*::*nat* **and** *ww1*::*Literal* **and** *ww2*::*Literal*
assume *a*: *0* ≤ *c* ∧ *c* < *length* (*getF ?state''*) ∧ *Some ww1* = (*getWatch1 ?state'' c*) ∧ *Some ww2* = (*getWatch2 ?state'' c*)
assume *b*: *literalFalse ww2* (*elements M*)

have (∃ *l*. *l el* ((*getF ?state''*) ! *c*) ∧ *literalTrue l* (*elements M*) ∧ *elementLevel l M* ≤ *elementLevel* (*opposite ww2*) *M*) ∨
(∀ *l*. *l el* ((*getF ?state''*) ! *c*) ∧ *l* ≠ *ww1* ∧ *l* ≠ *ww2* ⟶
literalFalse l (*elements M*) ∧ *elementLevel* (*opposite l*) *M* ≤ *elementLevel* (*opposite ww2*) *M*)
proof (*cases c* = *clause*)
case *False*
thus *?thesis*
using *a* **and** *b*
using *Cons*(*6*)
unfolding *InvariantWatchCharacterization-def*
unfolding *watchCharacterizationCondition-def*
unfolding *setWatch2-def*
by *auto*
next
case *True*
with *a*
have *ww1* = *?w1* **and** *ww2* = *l'*
using ‹*getWatch1 ?state' clause* = *Some ?w1*›
using ‹*getWatch2 ?state' clause* = *Some ?w2*›[*THEN sym*]
unfolding *setWatch2-def*
by *auto*
with ‹¬ *literalFalse l'* (*elements* (*getM ?state'*))› *b*
Cons(*8*)
have *False*
by *simp*
thus *?thesis*
by *simp*
qed
}
ultimately
show *?thesis*
unfolding *InvariantWatchCharacterization-def*

unfolding *watchCharacterizationCondition-def*
by *blast*
qed
moreover
have ∀ (*c*::*nat*). *c* ∈ *set Wl'* ⟶ *Some literal* = (*getWatch1 ?state'' c*) ∨ *Some literal* = (*getWatch2 ?state'' c*)
using *Cons*(*10*)
using ‹*clause* ∉ *set Wl'*›
unfolding *setWatch2-def*
by *simp*
moreover
have *getM ?state''* = *getM state*
getF ?state'' = *getF state*
unfolding *setWatch2-def*
by *auto*
moreover
have *getWatch1 ?state'' clause* = *Some ?w1 getWatch2 ?state'' clause* = *Some l'*
using ‹*getWatch1 ?state' clause* = *Some ?w1*›
unfolding *setWatch2-def*
by *auto*
hence *getWatch1 ?fState clause* = *getWatch1 ?state'' clause* ∧ *getWatch2 ?fState clause* = *Some l'*
using ‹*clause* ∉ *set Wl'*›
using ‹*InvariantWatchesEl* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)› ‹*getF ?state''* = *getF state*›
using *Cons*(*7*)
using *notifyWatchesLoopPreservedWatches*[*of ?state'' Wl' literal newWl*]
by (*simp add*: *Let-def*)
moreover
have *watchCharacterizationCondition ?w1 l'* (*getM ?fState*) (*getF ?fState* ! *clause*) ∧
watchCharacterizationCondition l' ?w1 (*getM ?fState*) (*getF ?fState* ! *clause*)
proof−
have (*getM ?fState*) = (*getM state*) (*getF ?fState*) = (*getF state*)
using *notifyWatchesLoopPreservedVariables*[*of ?state'' Wl' literal newWl*]
using ‹*InvariantWatchesEl* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)› ‹*getF ?state''* = *getF state*›
using *Cons*(*7*)
unfolding *setWatch2-def*
by (*auto simp add*: *Let-def*)

have *literalFalse ?w1* (*elements M*) ⟶
(∃ *l*. *l el* (*nth* (*getF ?state''*) *clause*) ∧ *literalTrue l* (*elements M*) ∧ *elementLevel l M* ≤ *elementLevel* (*opposite ?w1*) *M*)

proof
assume *literalFalse ?w1* (*elements M*)
show ∃ *l*. *l el* (*nth* (*getF ?state''*) *clause*) ∧ *literalTrue l* (*elements M*) ∧ *elementLevel l M* ≤ *elementLevel* (*opposite ?w1*) *M*
proof−
have ¬ (∀ *l*. *l el* (*nth* (*getF state*) *clause*) ∧ *l* ≠ *?w1* ∧ *l* ≠ *?w2* ⟶ *literalFalse l* (*elements M*))
using ‹*l' el* (*nth* (*getF ?state'*) *clause*)› ‹*l'* ≠ *?w1*› ‹*l'* ≠ *?w2*› ‹¬ *literalFalse l'* (*elements* (*getM ?state'*))›
using *Cons*(*8*)
unfolding *swapWatches-def*
by *auto*

from ‹*literalFalse ?w1* (*elements M*)› *Cons*(*6*)
have
(∃ *l*. *l el* (*getF state* ! *clause*) ∧ *literalTrue l* (*elements M*) ∧ *elementLevel l M* ≤ *elementLevel* (*opposite ?w1*) *M*) ∨
(∀ *l*. *l el* (*getF state* ! *clause*) ∧ *l* ≠ *?w1* ∧ *l* ≠ *?w2* ⟶
literalFalse l (*elements M*) ∧ *elementLevel* (*opposite l*) *M* ≤ *elementLevel* (*opposite ?w1*) *M*)
using ‹*0* ≤ *clause* ∧ *clause* < *length* (*getF state*)›
using ‹*getWatch1 ?state' clause* = *Some ?w1*›[*THEN sym*]
using ‹*getWatch2 ?state' clause* = *Some ?w2*›[*THEN sym*]
unfolding *InvariantWatchCharacterization-def*
unfolding *watchCharacterizationCondition-def*
by *simp*
with ‹¬ (∀ *l*. *l el* (*nth* (*getF state*) *clause*) ∧ *l* ≠ *?w1* ∧ *l* ≠ *?w2* ⟶ *literalFalse l* (*elements M*))›
have ∃ *l*. *l el* (*getF state* ! *clause*) ∧ *literalTrue l* (*elements M*) ∧ *elementLevel l M* ≤ *elementLevel* (*opposite ?w1*) *M*
by *auto*
thus *?thesis*
unfolding *setWatch2-def*
by *simp*
qed
qed
moreover
have *watchCharacterizationCondition l' ?w1* (*getM ?fState*) (*getF ?fState* ! *clause*)
using ‹¬ *literalFalse l'* (*elements* (*getM ?state'*))›
using ‹*getM ?fState* = *getM state*›
unfolding *watchCharacterizationCondition-def*
by *simp*
moreover
have *watchCharacterizationCondition ?w1 l'* (*getM ?fState*) (*getF ?fState* ! *clause*)
proof (*cases literalFalse ?w1* (*elements* (*getM ?fState*)))

case *True*
hence *literalFalse ?w1* (*elements M*)
using *notifyWatchesLoopPreservedVariables*[*of ?state″ Wl′ literal newWl*]
using ‹*InvariantWatchesEl* (*getF ?state″*) (*getWatch1 ?state″*) (*getWatch2 ?state″*)› ‹*getF ?state″* = *getF state*›
using *Cons*(*7*) *Cons*(*8*)
using ‹*?w1* ≠ *?w2*› ‹*?w2* = *literal*›
unfolding *setWatch2-def*
by (*simp add*: *Let-def*)
with ‹*literalFalse ?w1* (*elements M*) ⟶
(∃ *l*. *l el* (*nth* (*getF ?state″*) *clause*) ∧ *literalTrue l* (*elements M*) ∧ *elementLevel l M* ≤ *elementLevel* (*opposite ?w1*) *M*)›
obtain *l*::*Literal*
where *l el* (*nth* (*getF ?state″*) *clause*) **and**
literalTrue l (*elements M*) **and**
elementLevel l M ≤ *elementLevel* (*opposite ?w1*) *M*
by *auto*
hence *elementLevel l* (*getM state*) ≤ *elementLevel* (*opposite ?w1*) (*getM state*)
using *Cons*(*8*)
using ‹*literalTrue l* (*elements M*)› ‹*literalFalse ?w1* (*elements M*)›
using *elementLevelAppend*[*of l M* [(*opposite literal*, *decision*)]]
using *elementLevelAppend*[*of opposite ?w1 M* [(*opposite literal*, *decision*)]]
by *auto*
thus *?thesis*
using ‹*l el* (*nth* (*getF ?state″*) *clause*)› ‹*literalTrue l* (*elements M*)›
using ‹*getM ?fState* = *getM state*› ‹*getF ?fState* = *getF state*› ‹*getM ?state″* = *getM state*› ‹*getF ?state″* = *getF state*›
using *Cons*(*8*)
unfolding *watchCharacterizationCondition-def*
by *auto*
next
case *False*
thus *?thesis*
unfolding *watchCharacterizationCondition-def*
by *simp*
qed
ultimately
show *?thesis*
by *simp*
qed
ultimately
show *?thesis*
using *Cons*(*1*)[*of ?state″ newWl*]
using *Cons*(*7*) *Cons*(*8*)

using ‹*getWatch1 ?state′ clause = Some ?w1*›
using ‹*getWatch2 ?state′ clause = Some ?w2*›
using ‹*Some literal = getWatch2 state clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state′*))›
using ‹*getWatch1 ?state′′ clause = Some ?w1*›
using ‹*getWatch2 ?state′′ clause = Some l′*›
using *Some*
using ‹*uniq Wl′*›
using ‹*?w1* ≠ *?w2*›
by (*simp add*: *Let-def*)
next
case *None*
show *?thesis*
proof (*cases literalFalse ?w1* (*elements* (*getM ?state′*)))
case *True*
let *?state′′* = *?state′*(|*getConflictFlag* := *True*, *getConflictClause* := *clause*|)
let *?fState* = *notifyWatches-loop literal Wl′* (*clause* # *newWl*) *?state′′*

from *Cons*(*2*)
have *InvariantWatchesEl* (*getF ?state′′*) (*getWatch1 ?state′′*) (*getWatch2 ?state′′*)
unfolding *InvariantWatchesEl-def*
by *auto*
moreover
from *Cons*(*3*)
have *InvariantWatchesDiffer* (*getF ?state′*) (*getWatch1 ?state′*) (*getWatch2 ?state′*)
unfolding *InvariantWatchesDiffer-def*
by *auto*
moreover
from *Cons*(*4*)
have *InvariantConsistent* (*getM ?state′*)
unfolding *InvariantConsistent-def*
by *simp*
moreover
from *Cons*(*5*)
have *InvariantUniq* (*getM ?state′*)
unfolding *InvariantUniq-def*
by *simp*
moreover
from *Cons*(*6*)
have *InvariantWatchCharacterization* (*getF ?state′*) (*getWatch1 ?state′*) (*getWatch2 ?state′*) *M*
unfolding *InvariantWatchCharacterization-def*
unfolding *watchCharacterizationCondition-def*
by *simp*
moreover

have ∀ (*c*::*nat*). *c* ∈ *set Wl'* ⟶ *Some literal* = (*getWatch1 ?state'' c*) ∨ *Some literal* = (*getWatch2 ?state'' c*)
using *Cons*(*10*)
using ‹*clause* ∉ *set Wl'*›
by *simp*
moreover
have *getM ?state''* = *getM state*
getF ?state'' = *getF state*
by *auto*
moreover
have *getWatch1 ?fState clause* = *getWatch1 ?state'' clause* ∧ *getWatch2 ?fState clause* = *getWatch2 ?state'' clause*
using ‹*clause* ∉ *set Wl'*›
using ‹*InvariantWatchesEl* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)› ‹*getF ?state''* = *getF state*›
using *Cons*(*7*)
using *notifyWatchesLoopPreservedWatches*[*of ?state'' Wl' literal clause* # *newWl*]
by (*simp add*: *Let-def*)
moreover
have *literalFalse ?w1* (*elements M*)
using ‹*literalFalse ?w1* (*elements* (*getM ?state'*))›
‹*?w1* ≠ *?w2*› ‹*?w2* = *literal*› *Cons*(*8*)
by *auto*

have ¬ *literalTrue ?w2* (*elements M*)
using *Cons*(*4*)
using *Cons*(*8*)
using ‹*?w2* = *literal*›
using *inconsistentCharacterization*[*of elements M* @ [*opposite literal*]]
unfolding *InvariantConsistent-def*
by *force*

have ∗: ∀ *l*. *l el* (*nth* (*getF state*) *clause*) ∧ *l* ≠ *?w1* ∧ *l* ≠ *?w2* ⟶
literalFalse l (*elements M*) ∧ *elementLevel* (*opposite l*) *M* ≤ *elementLevel* (*opposite ?w1*) *M*
proof−
have ¬ (∃ *l*. *l el* (*nth* (*getF state*) *clause*) ∧ *literalTrue l* (*elements M*))
proof
assume ∃ *l*. *l el* (*nth* (*getF state*) *clause*) ∧ *literalTrue l* (*elements M*)
show *False*
proof−
from ‹∃ *l*. *l el* (*nth* (*getF state*) *clause*) ∧ *literalTrue l* (*elements M*)›
obtain *l*

where *l el* (*nth* (*getF state*) *clause*) *literalTrue l* (*elements M*)
by *auto*
hence *l* ≠ *?w1 l* ≠ *?w2*
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state′*))›
using ‹¬ *literalTrue ?w2* (*elements M*)›
using *Cons*(*8*)
by *auto*
with ‹*l el* (*nth* (*getF state*) *clause*)›
have *literalFalse l* (*elements* (*getM ?state′*))
using ‹*getWatch1 ?state′ clause* = *Some ?w1*›
using ‹*getWatch2 ?state′ clause* = *Some ?w2*›
using *None*
using *getNonWatchedUnfalsifiedLiteralNoneCharacterization*[*of nth* (*getF ?state′*) *clause ?w1 ?w2 getM ?state′*]
by *simp*
with ‹*l* ≠ *?w2*› ‹*?w2* = *literal*› *Cons*(*8*)
have *literalFalse l* (*elements M*)
by *simp*
with *Cons*(*4*) ‹*literalTrue l* (*elements M*)›
show *?thesis*
unfolding *InvariantConsistent-def*
using *Cons*(*8*)
by (*auto simp add*: *inconsistentCharacterization*)
qed
qed
with ‹*InvariantWatchCharacterization* (*getF state*) (*getWatch1 state*) (*getWatch2 state*) *M*›
show *?thesis*
unfolding *InvariantWatchCharacterization-def*
using ‹*literalFalse ?w1* (*elements M*)›
using ‹*getWatch1 ?state′ clause* = *Some ?w1*›[*THEN sym*]
using ‹*getWatch2 ?state′ clause* = *Some ?w2*›[*THEN sym*]
using ‹*0* ≤ *clause* ∧ *clause* < *length* (*getF state*)›
unfolding *watchCharacterizationCondition-def*
by (*simp*) (*blast*)
qed

have ∗∗: ∀ *l*. *l el* (*nth* (*getF ?state′′*) *clause*) ∧ *l* ≠ *?w1* ∧ *l* ≠ *?w2* ⟶
literalFalse l (*elements* (*getM ?state′′*)) ∧
elementLevel (*opposite l*) (*getM ?state′′*) ≤ *elementLevel* (*opposite ?w1*) (*getM ?state′′*)
proof−
{
fix *l*::*Literal*
assume *l el* (*nth* (*getF ?state′′*) *clause*) ∧ *l* ≠ *?w1* ∧ *l* ≠ *?w2*

have *literalFalse l* (*elements* (*getM ?state''*)) ∧
elementLevel (*opposite l*) (*getM ?state''*) ≤ *elementLevel* (*opposite ?w1*) (*getM ?state''*)
proof−
from ∗ ‹*l el* (*nth* (*getF ?state''*) *clause*) ∧ *l* ≠ *?w1* ∧ *l* ≠ *?w2*›
have *literalFalse l* (*elements M*) *elementLevel* (*opposite l*) *M* ≤ *elementLevel* (*opposite ?w1*) *M*
by *auto*
thus *?thesis*
using *elementLevelAppend*[*of opposite l M* [(*opposite literal*, *decision*)]]
using ‹*literalFalse ?w1* (*elements M*)›
using *elementLevelAppend*[*of opposite ?w1 M* [(*opposite literal*, *decision*)]]
using *Cons*(*8*)
by *simp*
qed
}
thus *?thesis*
by *simp*
qed

have (*getM ?fState*) = (*getM state*) (*getF ?fState*) = (*getF state*)
using *notifyWatchesLoopPreservedVariables*[*of ?state'' Wl' literal clause* # *newWl*]
using ‹*InvariantWatchesEl* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)› ‹*getF ?state''* = *getF state*›
using *Cons*(*7*)
by (*auto simp add*: *Let-def*)
hence ∀ *l*. *l el* (*nth* (*getF ?fState*) *clause*) ∧ *l* ≠ *?w1* ∧ *l* ≠ *?w2* ⟶
literalFalse l (*elements* (*getM ?fState*)) ∧
elementLevel (*opposite l*) (*getM ?fState*) ≤ *elementLevel* (*opposite ?w1*) (*getM ?fState*)
using ∗∗
using ‹*getM ?state''* = *getM state*›
using ‹*getF ?state''* = *getF state*›
by *simp*
moreover
have ∀ *l*. *literalFalse l* (*elements* (*getM ?fState*)) ⟶
elementLevel (*opposite l*) (*getM ?fState*) ≤ *elementLevel* (*opposite ?w2*) (*getM ?fState*)
proof−
have *elementLevel* (*opposite ?w2*) (*getM ?fState*) = *currentLevel* (*getM ?fState*)

using *Cons*(*8*)
using ‹(*getM ?fState*) = (*getM state*)›
using ‹¬ *literalFalse ?w2* (*elements M*)›
using ‹*?w2* = *literal*›
using *elementOnCurrentLevel*[*of opposite ?w2 M decision*]
by *simp*
thus *?thesis*
by (*simp add*: *elementLevelLeqCurrentLevel*)
qed
ultimately
show *?thesis*
using *Cons*(*1*)[*of ?state′′ clause # newWl*]
using *Cons*(*7*) *Cons*(*8*)
using ‹*getWatch1 ?state′ clause* = *Some ?w1*›
using ‹*getWatch2 ?state′ clause* = *Some ?w2*›
using ‹*Some literal* = *getWatch2 state clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state′*))›
using *None*
using ‹*literalFalse ?w1* (*elements* (*getM ?state′*))›
using ‹*uniq Wl′*›
using ‹*?w1* ≠ *?w2*›
unfolding *watchCharacterizationCondition-def*
by (*simp add*: *Let-def*)
next
case *False*

let *?state′′* = *setReason ?w1 clause* (*?state′*⦇*getQ* := (*if ?w1 el* (*getQ ?state′*) *then* (*getQ ?state′*) *else* (*getQ ?state′*) @ [*?w1*])⦈)
let *?fState* = *notifyWatches-loop literal Wl′* (*clause # newWl*) *?state′′*

from *Cons*(*2*)
have *InvariantWatchesEl* (*getF ?state′′*) (*getWatch1 ?state′′*) (*getWatch2 ?state′′*)
unfolding *InvariantWatchesEl-def*
unfolding *setReason-def*
by *auto*
moreover
from *Cons*(*3*)
have *InvariantWatchesDiffer* (*getF ?state′′*) (*getWatch1 ?state′′*) (*getWatch2 ?state′′*)
unfolding *InvariantWatchesDiffer-def*
unfolding *setReason-def*
by *auto*
moreover
from *Cons*(*4*)
have *InvariantConsistent* (*getM ?state′′*)
unfolding *InvariantConsistent-def*
unfolding *setReason-def*

by *simp*
moreover
from *Cons*(*5*)
have *InvariantUniq* (*getM ?state''*)
unfolding *InvariantUniq-def*
unfolding *setReason-def*
by *simp*
moreover
from *Cons*(*6*)
have *InvariantWatchCharacterization* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*) *M*
unfolding *setReason-def*
unfolding *InvariantWatchCharacterization-def*
unfolding *watchCharacterizationCondition-def*
by *simp*
moreover
have ∀ (*c*::*nat*). *c* ∈ *set Wl'* ⟶ *Some literal* = (*getWatch1 ?state'' c*) ∨ *Some literal* = (*getWatch2 ?state'' c*)
using *Cons*(*10*)
using ‹*clause* ∉ *set Wl'*›
unfolding *setReason-def*
by *simp*
moreover
have *getM ?state''* = *getM state*
getF ?state'' = *getF state*
unfolding *setReason-def*
by *auto*
moreover
have *getWatch1 ?state'' clause* = *Some ?w1 getWatch2 ?state'' clause* = *Some ?w2*
using ‹*getWatch1 ?state' clause* = *Some ?w1*›
using ‹*getWatch2 ?state' clause* = *Some ?w2*›
unfolding *setReason-def*
by *auto*
moreover
have *getWatch1 ?fState clause* = *Some ?w1 getWatch2 ?fState clause* = *Some ?w2*
using ‹*getWatch1 ?state'' clause* = *Some ?w1*› ‹*getWatch2 ?state'' clause* = *Some ?w2*›
using ‹*clause* ∉ *set Wl'*›
using ‹*InvariantWatchesEl* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)› ‹*getF ?state''* = *getF state*›
using *Cons*(*7*)
using *notifyWatchesLoopPreservedWatches*[*of ?state'' Wl' literal clause # newWl*]
by (*auto simp add*: *Let-def*)
moreover
have (*getM ?fState*) = (*getM state*) (*getF ?fState*) = (*getF state*)

using *notifyWatchesLoopPreservedVariables*[*of ?state'' Wl' literal clause # newWl*]
using ‹*InvariantWatchesEl* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)› ‹*getF ?state'' = getF state*›
using *Cons*(*7*)
unfolding *setReason-def*
by (*auto simp add*: *Let-def*)
ultimately
have $\forall$ *c*. *c* $\in$ *set Wl'* $\longrightarrow$ ($\forall$ *w1 w2*. *Some w1* = *getWatch1 ?fState c* $\wedge$ *Some w2* = *getWatch2 ?fState c* $\longrightarrow$
watchCharacterizationCondition w1 w2 (*getM ?fState*) (*getF ?fState* ! *c*) $\wedge$
watchCharacterizationCondition w2 w1 (*getM ?fState*) (*getF ?fState* ! *c*)) **and**
?fState = *notifyWatches-loop literal* (*clause # Wl'*) *newWl state*
using *Cons*(*1*)[*of ?state'' clause # newWl*]
using *Cons*(*7*) *Cons*(*8*)
using ‹*getWatch1 ?state' clause = Some ?w1*›
using ‹*getWatch2 ?state' clause = Some ?w2*›
using ‹*Some literal = getWatch2 state clause*›
using ‹$\neg$ *literalTrue ?w1* (*elements* (*getM ?state'*))›
using *None*
using ‹$\neg$ *literalFalse ?w1* (*elements* (*getM ?state'*))›
using ‹*uniq Wl'*›
by (*auto simp add*: *Let-def*)
moreover
have $*$: $\forall$ *l*. *l el* (*nth* (*getF ?state''*) *clause*) $\wedge$ *l* $\neq$ *?w1* $\wedge$ *l* $\neq$ *?w2* $\longrightarrow$ *literalFalse l* (*elements* (*getM ?state''*))
using *None*
using ‹*getWatch1 ?state' clause = Some ?w1*›
using ‹*getWatch2 ?state' clause = Some ?w2*›
using *getNonWatchedUnfalsifiedLiteralNoneCharacterization*[*of nth* (*getF ?state'*) *clause ?w1 ?w2 getM ?state'*]
using *Cons*(*8*)
unfolding *setReason-def*
by *auto*

have$**$: $\forall$ *l*. *l el* (*nth* (*getF ?fState*) *clause*) $\wedge$ *l* $\neq$ *?w1* $\wedge$ *l* $\neq$ *?w2* $\longrightarrow$ *literalFalse l* (*elements* (*getM ?fState*))
using ‹(*getM ?fState*) = (*getM state*)› ‹(*getF ?fState*) = (*getF state*)›
using $*$
using ‹*getM ?state'' = getM state*›
using ‹*getF ?state'' = getF state*›
by *auto*

have $***$: $\forall$ *l*. *literalFalse l* (*elements* (*getM ?fState*)) $\longrightarrow$
elementLevel (*opposite l*) (*getM ?fState*) $\leq$ *elementLevel*

(*opposite ?w2*) (*getM ?fState*)
proof−
have *elementLevel* (*opposite ?w2*) (*getM ?fState*) = *currentLevel* (*getM ?fState*)
using *Cons*(*8*)
using ‹(*getM ?fState*) = (*getM state*)›
using ‹¬ *literalFalse ?w2* (*elements M*)›
using ‹*?w2* = *literal*›
using *elementOnCurrentLevel*[*of opposite ?w2 M decision*]
by *simp*
thus *?thesis*
by (*simp add*: *elementLevelLeqCurrentLevel*)
qed

have (∀ *w1 w2*. *Some w1* = *getWatch1 ?fState clause* ∧ *Some w2* = *getWatch2 ?fState clause* ⟶
watchCharacterizationCondition w1 w2 (*getM ?fState*) (*getF ?fState* ! *clause*) ∧
watchCharacterizationCondition w2 w1 (*getM ?fState*) (*getF ?fState* ! *clause*))
proof−
{
fix *w1 w2*
assume *Some w1* = *getWatch1 ?fState clause* ∧ *Some w2* = *getWatch2 ?fState clause*
hence *w1* = *?w1 w2* = *?w2*
using ‹*getWatch1 ?fState clause* = *Some ?w1*›
using ‹*getWatch2 ?fState clause* = *Some ?w2*›
by *auto*
hence *watchCharacterizationCondition w1 w2* (*getM ?fState*) (*getF ?fState* ! *clause*) ∧
watchCharacterizationCondition w2 w1 (*getM ?fState*) (*getF ?fState* ! *clause*)
unfolding *watchCharacterizationCondition-def*
using ∗∗ ∗∗∗
unfolding *watchCharacterizationCondition-def*
using ‹(*getM ?fState*) = (*getM state*)› ‹(*getF ?fState*) = (*getF state*)›
using ‹¬ *literalFalse ?w1* (*elements* (*getM ?state′*))›
by *simp*
}
thus *?thesis*
by *auto*
qed
ultimately
show *?thesis*
by *simp*
qed
qed

  qed
 qed
qed

lemma *NotifyWatchesLoopConflictFlagEffect*:
fixes *literal* :: *Literal* **and** *Wl* :: *nat list* **and** *newWl* :: *nat list* **and** *state* :: *State*
assumes
 InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
and
 ∀ (*c*::*nat*). *c* ∈ *set Wl* ⟶ *0* ≤ *c* ∧ *c* < *length* (*getF state*) **and**
 InvariantConsistent (*getM state*)
 ∀ (*c*::*nat*). *c* ∈ *set Wl* ⟶ *Some literal* = (*getWatch1 state c*) ∨ *Some literal* = (*getWatch2 state c*)
 literalFalse literal (*elements* (*getM state*))
 uniq Wl
shows
 let state′ = *notifyWatches-loop literal Wl newWl state in*
   getConflictFlag state′ =
     (*getConflictFlag state* ∨
       (∃ *clause*. *clause* ∈ *set Wl* ∧ *clauseFalse* (*nth* (*getF state*) *clause*) (*elements* (*getM state*))))
using *assms*
proof (*induct Wl arbitrary*: *newWl state*)
 case *Nil*
 thus *?case*
  by *simp*
next
 case (*Cons clause Wl′*)

 from ‹*uniq* (*clause* # *Wl′*)›
 have *uniq Wl′* **and** *clause* ∉ *set Wl′*
  by (*auto simp add*: *uniqAppendIff*)

 from ‹∀ (*c*::*nat*). *c* ∈ *set* (*clause* # *Wl′*) ⟶ *0* ≤ *c* ∧ *c* < *length* (*getF state*)›
 have *0* ≤ *clause clause* < *length* (*getF state*)
  by *auto*
 then obtain *wa*::*Literal* **and** *wb*::*Literal*
   where *getWatch1 state clause* = *Some wa* **and** *getWatch2 state clause* = *Some wb*
  using *Cons*
  unfolding *InvariantWatchesEl-def*
  by *auto*
 show *?case*
 proof (*cases Some literal* = *getWatch1 state clause*)
  case *True*
  let *?state′* = *swapWatches clause state*

let *?w1* = *wb*
have *getWatch1 ?state′ clause* = *Some ?w1*
using ‹*getWatch2 state clause* = *Some wb*›
unfolding *swapWatches-def*
by *auto*
let *?w2* = *wa*
have *getWatch2 ?state′ clause* = *Some ?w2*
using ‹*getWatch1 state clause* = *Some wa*›
unfolding *swapWatches-def*
by *auto*

from ‹*Some literal* = *getWatch1 state clause*›
‹*getWatch2 ?state′ clause* = *Some ?w2*›
‹*literalFalse literal* (*elements* (*getM state*))›
have *literalFalse ?w2* (*elements* (*getM state*))
unfolding *swapWatches-def*
by *simp*

from ‹*InvariantWatchesEl* (*getF state*) (*getWatch1 state*) (*getWatch2 state*)›
have *?w1 el* (*nth* (*getF state*) *clause*)
using ‹*getWatch1 ?state′ clause* = *Some ?w1*›
using ‹*getWatch2 ?state′ clause* = *Some ?w2*›
using ‹*clause* < *length* (*getF state*)›
unfolding *InvariantWatchesEl-def*
unfolding *swapWatches-def*
by *auto*

show *?thesis*
proof (*cases literalTrue ?w1* (*elements* (*getM ?state′*)))
case *True*

from *Cons*(*2*)
have *InvariantWatchesEl* (*getF ?state′*) (*getWatch1 ?state′*) (*getWatch2 ?state′*)
unfolding *InvariantWatchesEl-def*
unfolding *swapWatches-def*
by *auto*
moreover
have *getF ?state′* = *getF state* ∧
getM ?state′ = *getM state* ∧
getConflictFlag ?state′ = *getConflictFlag state*

unfolding *swapWatches-def*
by *simp*
moreover
have ∀ *c*. *c* ∈ *set Wl′* ⟶ *Some literal* = *getWatch1 ?state′ c* ∨ *Some literal* = *getWatch2 ?state′ c*
using *Cons*(*5*)

unfolding *swapWatches-def*
by *auto*
moreover
have ¬ *clauseFalse* (*nth* (*getF state*) *clause*) (*elements* (*getM state*))
using ‹*?w1 el* (*nth* (*getF state*) *clause*)›
using ‹*literalTrue ?w1* (*elements* (*getM ?state′*))›
using ‹*InvariantConsistent* (*getM state*)›
unfolding *InvariantConsistent-def*
unfolding *swapWatches-def*
by (*auto simp add*: *clauseFalseIffAllLiteralsAreFalse inconsistentCharacterization*)
ultimately
show *?thesis*
using *Cons*(*1*)[*of ?state′ clause # newWl*]
using *Cons*(*3*) *Cons*(*4*) *Cons*(*6*)
using ‹*getWatch1 ?state′ clause = Some ?w1*›
using ‹*getWatch2 ?state′ clause = Some ?w2*›
using ‹*Some literal = getWatch1 state clause*›
using ‹*literalTrue ?w1* (*elements* (*getM ?state′*))›
using ‹*uniq Wl′*›
by (*auto simp add*:*Let-def*)
next
case *False*
show *?thesis*
proof (*cases getNonWatchedUnfalsifiedLiteral* (*nth* (*getF ?state′*) *clause*) *?w1 ?w2* (*getM ?state′*))
case (*Some l′*)
hence *l′ el* (*nth* (*getF ?state′*) *clause*) ¬ *literalFalse l′* (*elements* (*getM ?state′*))
using *getNonWatchedUnfalsifiedLiteralSomeCharacterization*
by *auto*

let *?state′′* = *setWatch2 clause l′ ?state′*

from *Cons*(*2*)
have *InvariantWatchesEl* (*getF ?state′′*) (*getWatch1 ?state′′*) (*getWatch2 ?state′′*)
using ‹*l′ el* (*nth* (*getF ?state′*) *clause*)›
unfolding *InvariantWatchesEl-def*
unfolding *swapWatches-def*
unfolding *setWatch2-def*
by *auto*
moreover
from *Cons*(*4*)
have *InvariantConsistent* (*getM ?state′′*)
unfolding *setWatch2-def*
unfolding *swapWatches-def*
by *simp*

moreover
have *getM ?state''* = *getM state* ∧
getF ?state'' = *getF state* ∧
getConflictFlag ?state'' = *getConflictFlag state*
unfolding *swapWatches-def*
unfolding *setWatch2-def*
by *simp*
moreover
have ∀ *c. c* ∈ *set Wl'* ⟶ *Some literal* = *getWatch1 ?state'' c* ∨ *Some literal* = *getWatch2 ?state'' c*
using *Cons*(*5*)
using ‹*clause* ∉ *set Wl'*›
unfolding *swapWatches-def*
unfolding *setWatch2-def*
by *auto*
moreover
have ¬ *clauseFalse* (*nth* (*getF state*) *clause*) (*elements* (*getM state*))
using ‹*l' el* (*nth* (*getF ?state'*) *clause*)›
using ‹¬ *literalFalse l'* (*elements* (*getM ?state'*))›
using ‹*InvariantConsistent* (*getM state*)›
unfolding *InvariantConsistent-def*
unfolding *swapWatches-def*
by (*auto simp add*: *clauseFalseIffAllLiteralsAreFalse inconsistentCharacterization*)
ultimately
show *?thesis*
using *Cons*(*1*)[*of ?state'' newWl*]
using *Cons*(*3*) *Cons*(*4*) *Cons*(*6*)
using ‹*getWatch1 ?state' clause* = *Some ?w1*›
using ‹*getWatch2 ?state' clause* = *Some ?w2*›
using ‹*Some literal* = *getWatch1 state clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state'*))›
using ‹*uniq Wl'*›
using *Some*
by (*auto simp add*: *Let-def*)
next
case *None*
hence ∀ *l. l el* (*nth* (*getF ?state'*) *clause*) ∧ *l* ≠ *?w1* ∧ *l* ≠ *?w2* ⟶ *literalFalse l* (*elements* (*getM ?state'*))
using *getNonWatchedUnfalsifiedLiteralNoneCharacterization*
by *simp*
show *?thesis*
proof (*cases literalFalse ?w1* (*elements* (*getM ?state'*)))
case *True*
let *?state''* = *?state'*⦇*getConflictFlag* := *True*, *getConflictClause* := *clause*⦈

from *Cons*(*2*)

have *InvariantWatchesEl* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)
unfolding *InvariantWatchesEl-def*
unfolding *swapWatches-def*
by *auto*
moreover
from *Cons*(*4*)
have *InvariantConsistent* (*getM ?state''*)
unfolding *setWatch2-def*
unfolding *swapWatches-def*
by *simp*
moreover
have *getM ?state'' = getM state* ∧
getF ?state'' = getF state ∧
getSATFlag ?state'' = getSATFlag state
unfolding *swapWatches-def*
by *simp*
moreover
have ∀ *c. c* ∈ *set Wl'* ⟶ *Some literal = getWatch1 ?state'' c* ∨ *Some literal = getWatch2 ?state'' c*
using *Cons*(*5*)
using ‹*clause* ∉ *set Wl'*›
unfolding *swapWatches-def*
unfolding *setWatch2-def*
by *auto*
moreover
have *clauseFalse* (*nth* (*getF state*) *clause*) (*elements* (*getM state*))
using ‹∀ *l. l el* (*nth* (*getF ?state'*) *clause*) ∧ *l* ≠ *?w1* ∧ *l* ≠ *?w2* ⟶ *literalFalse l* (*elements* (*getM ?state'*))›
using ‹*literalFalse ?w1* (*elements* (*getM ?state'*))›
using ‹*literalFalse ?w2* (*elements* (*getM state*))›
unfolding *swapWatches-def*
by (*auto simp add*: *clauseFalseIffAllLiteralsAreFalse*)
ultimately
show *?thesis*
using *Cons*(*1*)[*of ?state'' clause # newWl*]
using *Cons*(*3*) *Cons*(*4*) *Cons*(*6*)
using ‹*getWatch1 ?state' clause = Some ?w1*›
using ‹*getWatch2 ?state' clause = Some ?w2*›
using ‹*Some literal = getWatch1 state clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state'*))›
using *None*
using ‹*literalFalse ?w1* (*elements* (*getM ?state'*))›
using ‹*uniq Wl'*›
by (*auto simp add*: *Let-def*)
next
case *False*
let *?state'' = setReason ?w1 clause* (*?state'*⦇*getQ* := (*if ?w1*

el (*getQ ?state′*) *then* (*getQ ?state′*) *else* (*getQ ?state′*) @ [*?w1*])⦈)

from *Cons*(*2*)
have *InvariantWatchesEl* (*getF ?state′′*) (*getWatch1 ?state′′*) (*getWatch2 ?state′′*)
unfolding *InvariantWatchesEl-def*
unfolding *swapWatches-def*
unfolding *setReason-def*
by *auto*
moreover
from *Cons*(*4*)
have *InvariantConsistent* (*getM ?state′′*)
unfolding *swapWatches-def*
unfolding *setReason-def*
by *simp*
moreover
have *getM ?state′′* = *getM state* ∧
getF ?state′′ = *getF state* ∧
getSATFlag ?state′′ = *getSATFlag state*
unfolding *swapWatches-def*
unfolding *setReason-def*
by *simp*
moreover
have ∀ *c*. *c* ∈ *set Wl′* ⟶ *Some literal* = *getWatch1 ?state′′ c* ∨ *Some literal* = *getWatch2 ?state′′ c*
using *Cons*(*5*)
using ‹*clause* ∉ *set Wl′*›
unfolding *swapWatches-def*
unfolding *setReason-def*
by *auto*
moreover
have ¬ *clauseFalse* (*nth* (*getF state*) *clause*) (*elements* (*getM state*))
using ‹*?w1 el* (*nth* (*getF state*) *clause*)›
using ‹¬ *literalFalse ?w1* (*elements* (*getM ?state′*))›
using ‹*InvariantConsistent* (*getM state*)›
unfolding *InvariantConsistent-def*
unfolding *swapWatches-def*
by (*auto simp add*: *clauseFalseIffAllLiteralsAreFalse inconsistentCharacterization*)
ultimately
show *?thesis*
using *Cons*(*1*)[*of ?state′′ clause* # *newWl*]
using *Cons*(*3*) *Cons*(*4*) *Cons*(*6*)
using ‹*getWatch1 ?state′ clause* = *Some ?w1*›
using ‹*getWatch2 ?state′ clause* = *Some ?w2*›
using ‹*Some literal* = *getWatch1 state clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state′*))›
using *None*

using ‹¬ *literalFalse ?w1* (*elements* (*getM ?state'*))›
using ‹*uniq Wl'*›
apply (*simp add*: *Let-def*)
unfolding *setReason-def*
unfolding *swapWatches-def*
by *auto*
qed
qed
qed
next
case *False*
let *?state'* = *state*
let *?w1* = *wa*
have *getWatch1 ?state' clause* = *Some ?w1*
using ‹*getWatch1 state clause* = *Some wa*›
unfolding *swapWatches-def*
by *auto*
let *?w2* = *wb*
have *getWatch2 ?state' clause* = *Some ?w2*
using ‹*getWatch2 state clause* = *Some wb*›
unfolding *swapWatches-def*
by *auto*

from ‹¬ *Some literal* = *getWatch1 state clause*›
‹∀ (*c*::*nat*). *c* ∈ *set* (*clause* # *Wl'*) ⟶ *Some literal* = (*getWatch1 state c*) ∨ *Some literal* = (*getWatch2 state c*)›
have *Some literal* = *getWatch2 state clause*
by *auto*
hence *literalFalse ?w2* (*elements* (*getM state*))
using
‹*getWatch2 ?state' clause* = *Some ?w2*›
‹*literalFalse literal* (*elements* (*getM state*))›
by *simp*

from ‹*InvariantWatchesEl* (*getF state*) (*getWatch1 state*) (*getWatch2 state*)›
have *?w1 el* (*nth* (*getF state*) *clause*)
using ‹*getWatch1 ?state' clause* = *Some ?w1*›
using ‹*getWatch2 ?state' clause* = *Some ?w2*›
using ‹*clause* < *length* (*getF state*)›
unfolding *InvariantWatchesEl-def*
unfolding *swapWatches-def*
by *auto*

show *?thesis*
proof (*cases literalTrue ?w1* (*elements* (*getM ?state'*)))
case *True*

have ¬ *clauseFalse* (*nth* (*getF state*) *clause*) (*elements* (*getM*

state))
using ‹*?w1 el* (*nth* (*getF state*) *clause*)›
using ‹*literalTrue ?w1* (*elements* (*getM ?state'*))›
using ‹*InvariantConsistent* (*getM state*)›
unfolding *InvariantConsistent-def*
unfolding *swapWatches-def*
by (*auto simp add*: *clauseFalseIffAllLiteralsAreFalse inconsistentCharacterization*)

thus *?thesis*
using *True*
using *Cons*(*1*)[*of ?state' clause # newWl*]
using *Cons*(*2*) *Cons*(*3*) *Cons*(*4*) *Cons*(*5*) *Cons*(*6*)
using ‹¬ *Some literal* = *getWatch1 state clause*›
using ‹*getWatch1 ?state' clause* = *Some ?w1*›
using ‹*getWatch2 ?state' clause* = *Some ?w2*›
using ‹*literalTrue ?w1* (*elements* (*getM ?state'*))›
using ‹*uniq Wl'*›
by (*auto simp add*:*Let-def*)
next
case *False*
show *?thesis*
proof (*cases getNonWatchedUnfalsifiedLiteral* (*nth* (*getF ?state'*) *clause*) *?w1 ?w2* (*getM ?state'*))
case (*Some l'*)
hence *l' el* (*nth* (*getF ?state'*) *clause*) ¬ *literalFalse l'* (*elements* (*getM ?state'*))
using *getNonWatchedUnfalsifiedLiteralSomeCharacterization*
by *auto*

let *?state''* = *setWatch2 clause l' ?state'*

from *Cons*(*2*)
have *InvariantWatchesEl* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)
using ‹*l' el* (*nth* (*getF ?state'*) *clause*)›
unfolding *InvariantWatchesEl-def*
unfolding *setWatch2-def*
by *auto*
moreover
from *Cons*(*4*)
have *InvariantConsistent* (*getM ?state''*)
unfolding *setWatch2-def*
by *simp*
moreover
have *getM ?state''* = *getM state* ∧
getF ?state'' = *getF state* ∧
getConflictFlag ?state'' = *getConflictFlag state*
unfolding *setWatch2-def*

by *simp*
moreover
have ∀ *c*. *c* ∈ *set Wl'* ⟶ *Some literal* = *getWatch1 ?state'' c* ∨ *Some literal* = *getWatch2 ?state'' c*
using *Cons*(*5*)
using ‹*clause* ∉ *set Wl'*›
unfolding *setWatch2-def*
by *auto*
moreover
have ¬ *clauseFalse* (*nth* (*getF state*) *clause*) (*elements* (*getM state*))
using ‹*l' el* (*nth* (*getF ?state'*) *clause*)›
using ‹¬ *literalFalse l'* (*elements* (*getM ?state'*))›
using ‹*InvariantConsistent* (*getM state*)›
unfolding *InvariantConsistent-def*
by (*auto simp add*: *clauseFalseIffAllLiteralsAreFalse inconsistentCharacterization*)
ultimately
show *?thesis*
using *Cons*(*1*)[*of ?state'' newWl*]
using *Cons*(*3*) *Cons*(*4*) *Cons*(*6*)
using ‹*getWatch1 ?state' clause* = *Some ?w1*›
using ‹*getWatch2 ?state' clause* = *Some ?w2*›
using ‹¬ *Some literal* = *getWatch1 state clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state'*))›
using ‹*uniq Wl'*›
using *Some*
by (*auto simp add*: *Let-def*)
next
case *None*
hence ∀ *l*. *l el* (*nth* (*getF ?state'*) *clause*) ∧ *l* ≠ *?w1* ∧ *l* ≠ *?w2* ⟶ *literalFalse l* (*elements* (*getM ?state'*))
using *getNonWatchedUnfalsifiedLiteralNoneCharacterization*
by *simp*
show *?thesis*
proof (*cases literalFalse ?w1* (*elements* (*getM ?state'*)))
case *True*
let *?state''* = *?state'*⦇*getConflictFlag* := *True*, *getConflictClause* := *clause*⦈

from *Cons*(*2*)
have *InvariantWatchesEl* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)
unfolding *InvariantWatchesEl-def*
by *auto*
moreover
from *Cons*(*4*)
have *InvariantConsistent* (*getM ?state''*)
unfolding *setWatch2-def*

by *simp*
moreover
have *getM ?state'' = getM state ∧*
getF ?state'' = getF state ∧
getSATFlag ?state'' = getSATFlag state
by *simp*
moreover
have *∀ c. c ∈ set Wl' ⟶ Some literal = getWatch1 ?state'' c ∨ Some literal = getWatch2 ?state'' c*
using *Cons(5)*
using *‹clause ∉ set Wl'›*
unfolding *setWatch2-def*
by *auto*
moreover
have *clauseFalse (nth (getF state) clause) (elements (getM state))*
using *‹∀ l. l el (nth (getF ?state') clause) ∧ l ≠ ?w1 ∧ l ≠ ?w2 ⟶ literalFalse l (elements (getM ?state'))›*
using *‹literalFalse ?w1 (elements (getM ?state'))›*
using *‹literalFalse ?w2 (elements (getM state))›*
by *(auto simp add: clauseFalseIffAllLiteralsAreFalse)*
ultimately
show *?thesis*
using *Cons(1)[of ?state'' clause # newWl]*
using *Cons(3) Cons(4) Cons(6)*
using *‹getWatch1 ?state' clause = Some ?w1›*
using *‹getWatch2 ?state' clause = Some ?w2›*
using *‹¬ Some literal = getWatch1 state clause›*
using *‹¬ literalTrue ?w1 (elements (getM ?state'))›*
using *None*
using *‹literalFalse ?w1 (elements (getM ?state'))›*
using *‹uniq Wl'›*
by *(auto simp add: Let-def)*
next
case *False*
let *?state'' = setReason ?w1 clause (?state'⦇getQ := (if ?w1 el (getQ ?state') then (getQ ?state') else (getQ ?state') @ [?w1])⦈)*

from *Cons(2)*
have *InvariantWatchesEl (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'')*
unfolding *InvariantWatchesEl-def*
unfolding *setReason-def*
by *auto*
moreover
from *Cons(4)*
have *InvariantConsistent (getM ?state'')*
unfolding *setReason-def*
by *simp*

moreover
have *getM ?state''* = *getM state* ∧
getF ?state'' = *getF state* ∧
getSATFlag ?state'' = *getSATFlag state*
unfolding *setReason-def*
by *simp*
moreover
have ∀ *c*. *c* ∈ *set Wl'* ⟶ *Some literal* = *getWatch1 ?state'' c* ∨ *Some literal* = *getWatch2 ?state'' c*
using *Cons*(*5*)
using ‹*clause* ∉ *set Wl'*›
unfolding *setReason-def*
by *auto*
moreover
have ¬ *clauseFalse* (*nth* (*getF state*) *clause*) (*elements* (*getM state*))
using ‹*?w1 el* (*nth* (*getF state*) *clause*)›
using ‹¬ *literalFalse ?w1* (*elements* (*getM ?state'*))›
using ‹*InvariantConsistent* (*getM state*)›
unfolding *InvariantConsistent-def*
by (*auto simp add*: *clauseFalseIffAllLiteralsAreFalse inconsistentCharacterization*)
ultimately
show *?thesis*
using *Cons*(*1*)[*of ?state'' clause # newWl*]
using *Cons*(*3*) *Cons*(*4*) *Cons*(*6*)
using ‹*getWatch1 ?state' clause* = *Some ?w1*›
using ‹*getWatch2 ?state' clause* = *Some ?w2*›
using ‹¬ *Some literal* = *getWatch1 state clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state'*))›
using *None*
using ‹¬ *literalFalse ?w1* (*elements* (*getM ?state'*))›
using ‹*uniq Wl'*›
apply (*simp add*: *Let-def*)
unfolding *setReason-def*
by *auto*
qed
qed
qed
qed
qed

lemma *NotifyWatchesLoopQEffect*:
fixes *literal* :: *Literal* **and** *Wl* :: *nat list* **and** *newWl* :: *nat list* **and** *state* :: *State*
assumes
(*getM state*) = *M* @ [(*opposite literal*, *decision*)] **and**
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)

and
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
$\forall$ (*c*::*nat*). $c \in$ *set Wl* $\longrightarrow$ $0 \leq c \wedge c <$ *length* (*getF state*) **and**
InvariantConsistent (*getM state*) **and**
$\forall$ (*c*::*nat*). $c \in$ *set Wl* $\longrightarrow$ *Some literal* = (*getWatch1 state c*) $\vee$ *Some literal* = (*getWatch2 state c*) **and**
uniq Wl **and**
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) *M*
shows
let state' = *notifyWatches-loop literal Wl newWl state in*
((∀ *l*. $l \in$ (*set* (*getQ state'*) − *set* (*getQ state*)) ⟶
(∃ *clause*. (*clause el* (*getF state*) ∧
literal el clause ∧
(*isUnitClause clause l* (*elements* (*getM state*)))))) ∧
(∀ *clause*. *clause* ∈ *set Wl* ⟶
(∀ *l*. (*isUnitClause* (*nth* (*getF state*) *clause*) *l* (*elements* (*getM state*))) ⟶
l ∈ (*set* (*getQ state'*)))))
(**is** *let state'* = *notifyWatches-loop literal Wl newWl state in* (*?Cond1 state' state* ∧ *?Cond2 Wl state' state*))
using *assms*
proof (*induct Wl arbitrary*: *newWl state*)
case *Nil*
thus *?case*
by *simp*
next
case (*Cons clause Wl'*)

from ‹*uniq* (*clause* # *Wl'*)›
have *uniq Wl'* **and** *clause* ∉ *set Wl'*
by (*auto simp add*: *uniqAppendIff*)

from ‹∀ (*c*::*nat*). *c* ∈ *set* (*clause* # *Wl'*) ⟶ *0* ≤ *c* ∧ *c* < *length* (*getF state*)›
have *0* ≤ *clause clause* < *length* (*getF state*)
by *auto*
then obtain *wa*::*Literal* **and** *wb*::*Literal*
where *getWatch1 state clause* = *Some wa* **and** *getWatch2 state clause* = *Some wb*
using *Cons*
unfolding *InvariantWatchesEl-def*
by *auto*

from ‹*0* ≤ *clause*› ‹*clause* < *length* (*getF state*)›
have (*nth* (*getF state*) *clause*) *el* (*getF state*)
by *simp*

```
  show ?case
  proof (cases Some literal = getWatch1 state clause)
    case True
    let ?state' = swapWatches clause state
    let ?w1 = wb
    have getWatch1 ?state' clause = Some ?w1
      using ‹getWatch2 state clause = Some wb›
      unfolding swapWatches-def
      by auto
    let ?w2 = wa
    have getWatch2 ?state' clause = Some ?w2
      using ‹getWatch1 state clause = Some wa›
      unfolding swapWatches-def
      by auto

    have ?w2 = literal
      using ‹Some literal = getWatch1 state clause›
      using ‹getWatch2 ?state' clause = Some ?w2›
      unfolding swapWatches-def
      by simp

    hence literalFalse ?w2 (elements (getM state))
      using ‹(getM state) = M @ [(opposite literal, decision)]›
      by simp

   from ‹InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2
state)›
     have ?w1 el (nth (getF state) clause) ?w2 el (nth (getF state)
clause)
      using ‹getWatch1 ?state' clause = Some ?w1›
      using ‹getWatch2 ?state' clause = Some ?w2›
      using ‹clause < length (getF state)›
      unfolding InvariantWatchesEl-def
      unfolding swapWatches-def
      by auto

   from ‹InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2
state)›
    have ?w1 ≠ ?w2
      using ‹getWatch1 ?state' clause = Some ?w1›
      using ‹getWatch2 ?state' clause = Some ?w2›
      using ‹clause < length (getF state)›
      unfolding InvariantWatchesDiffer-def
      unfolding swapWatches-def
      by auto

    show ?thesis
    proof (cases literalTrue ?w1 (elements (getM ?state')))
      case True
```

from *Cons*(*3*)
have *InvariantWatchesEl* (*getF ?state'*) (*getWatch1 ?state'*) (*getWatch2 ?state'*)
unfolding *InvariantWatchesEl-def*
unfolding *swapWatches-def*
by *auto*
moreover
from *Cons*(*4*)
have *InvariantWatchesDiffer* (*getF ?state'*) (*getWatch1 ?state'*) (*getWatch2 ?state'*)
unfolding *InvariantWatchesDiffer-def*
unfolding *swapWatches-def*
by *auto*
moreover
have *getF ?state' = getF state* ∧
getM ?state' = getM state ∧
getQ ?state' = getQ state ∧
getConflictFlag ?state' = getConflictFlag state

unfolding *swapWatches-def*
by *simp*
moreover
have ∀ *c. c* ∈ *set Wl'* ⟶ *Some literal = getWatch1 ?state' c* ∨ *Some literal = getWatch2 ?state' c*
using *Cons*(*7*)
unfolding *swapWatches-def*
by *auto*
moreover
have *InvariantWatchCharacterization* (*getF ?state'*) (*getWatch1 ?state'*) (*getWatch2 ?state'*) *M*
using *Cons*(*9*)
unfolding *swapWatches-def*
unfolding *InvariantWatchCharacterization-def*
by *auto*
moreover
have ¬ (∃ *l. isUnitClause* (*nth* (*getF state*) *clause*) *l* (*elements* (*getM state*)))
using ‹*?w1 el* (*nth* (*getF state*) *clause*)›
using ‹*literalTrue ?w1* (*elements* (*getM ?state'*))›
using ‹*InvariantConsistent* (*getM state*)›
unfolding *InvariantConsistent-def*
unfolding *swapWatches-def*
by (*auto simp add*: *isUnitClause-def inconsistentCharacterization*)
ultimately
show *?thesis*
using *Cons*(*1*)[*of ?state' clause # newWl*]
using *Cons*(*2*) *Cons*(*5*) *Cons*(*6*)

using ‹*getWatch1 ?state′ clause = Some ?w1*›
using ‹*getWatch2 ?state′ clause = Some ?w2*›
using ‹*Some literal = getWatch1 state clause*›
using ‹*literalTrue ?w1* (*elements* (*getM ?state′*))›
using ‹*uniq Wl′*›
by (*simp add*:*Let-def*)
next
case *False*
show *?thesis*
proof (*cases getNonWatchedUnfalsifiedLiteral* (*nth* (*getF ?state′*) *clause*) *?w1 ?w2* (*getM ?state′*))
case (*Some l′*)
hence *l′ el* (*nth* (*getF ?state′*) *clause*) ¬ *literalFalse l′* (*elements* (*getM ?state′*)) *l′* ≠ *?w1 l′* ≠ *?w2*
using *getNonWatchedUnfalsifiedLiteralSomeCharacterization*
by *auto*

let *?state′′* = *setWatch2 clause l′ ?state′*

from *Cons*(*3*)
have *InvariantWatchesEl* (*getF ?state′′*) (*getWatch1 ?state′′*) (*getWatch2 ?state′′*)
using ‹*l′ el* (*nth* (*getF ?state′*) *clause*)›
unfolding *InvariantWatchesEl-def*
unfolding *swapWatches-def*
unfolding *setWatch2-def*
by *auto*
moreover
from *Cons*(*4*)
have *InvariantWatchesDiffer* (*getF ?state′′*) (*getWatch1 ?state′′*) (*getWatch2 ?state′′*)
using ‹*l′* ≠ *?w1*›
using ‹*getWatch1 ?state′ clause = Some ?w1*›
using ‹*getWatch2 ?state′ clause = Some ?w2*›
unfolding *InvariantWatchesDiffer-def*
unfolding *swapWatches-def*
unfolding *setWatch2-def*
by *auto*
moreover
from *Cons*(*6*)
have *InvariantConsistent* (*getM ?state′′*)
unfolding *setWatch2-def*
unfolding *swapWatches-def*
by *simp*
moreover
have *getM ?state′′ = getM state* ∧
getF ?state′′ = getF state ∧
getQ ?state′′ = getQ state ∧
getConflictFlag ?state′′ = getConflictFlag state

unfolding *swapWatches-def*
unfolding *setWatch2-def*
by *simp*
moreover
have ∀ *c*. *c* ∈ *set Wl′* ⟶ *Some literal* = *getWatch1 ?state″ c* ∨ *Some literal* = *getWatch2 ?state″ c*
using *Cons*(*7*)
using ‹*clause* ∉ *set Wl′*›
unfolding *swapWatches-def*
unfolding *setWatch2-def*
by *auto*
moreover
have *InvariantWatchCharacterization* (*getF ?state″*) (*getWatch1 ?state″*) (*getWatch2 ?state″*) *M*
proof−
{
fix *c*::*nat* **and** *ww1*::*Literal* **and** *ww2*::*Literal*
assume *a*: *0* ≤ *c* ∧ *c* < *length* (*getF ?state″*) ∧ *Some ww1* = (*getWatch1 ?state″ c*) ∧ *Some ww2* = (*getWatch2 ?state″ c*)
assume *b*: *literalFalse ww1* (*elements M*)

have (∃ *l*. *l el* ((*getF ?state″*) ! *c*) ∧ *literalTrue l* (*elements M*) ∧ *elementLevel l M* ≤ *elementLevel* (*opposite ww1*) *M*) ∨
(∀ *l*. *l el* ((*getF ?state″*) ! *c*) ∧ *l* ≠ *ww1* ∧ *l* ≠ *ww2* ⟶
literalFalse l (*elements M*) ∧ *elementLevel* (*opposite l*) *M* ≤ *elementLevel* (*opposite ww1*) *M*)
proof (*cases c* = *clause*)
case *False*
thus *?thesis*
using *a* **and** *b*
using *Cons*(*9*)
unfolding *InvariantWatchCharacterization-def*
unfolding *watchCharacterizationCondition-def*
unfolding *swapWatches-def*
unfolding *setWatch2-def*
by *simp*
next
case *True*
with *a*
have *ww1* = *?w1* **and** *ww2* = *l′*
using ‹*getWatch1 ?state′ clause* = *Some ?w1*›
using ‹*getWatch2 ?state′ clause* = *Some ?w2*›[*THEN sym*]
unfolding *setWatch2-def*
unfolding *swapWatches-def*
by *auto*

have ¬ (∀ *l*. *l el* (*getF state* ! *clause*) ∧ *l* ≠ *?w1* ∧ *l* ≠ *?w2* ⟶ *literalFalse l* (*elements M*))

using *Cons*(*2*)
using ‹*l′* ≠ *?w1*› **and** ‹*l′* ≠ *?w2*› ‹*l′* *el* (*nth* (*getF* *?state′*) *clause*)›
using ‹¬ *literalFalse* *l′* (*elements* (*getM* *?state′*))›
using *a* **and** *b*
using ‹*c* = *clause*›
unfolding *swapWatches-def*
unfolding *setWatch2-def*
by *auto*
moreover
have (∃ *l*. *l* *el* (*getF* *state* ! *clause*) ∧ *literalTrue* *l* (*elements* *M*) ∧
elementLevel *l* *M* ≤ *elementLevel* (*opposite* *?w1*) *M*) ∨
(∀ *l*. *l* *el* (*getF* *state* ! *clause*) ∧ *l* ≠ *?w1* ∧ *l* ≠ *?w2* ⟶ *literalFalse* *l* (*elements* *M*))
using *Cons*(*9*)
unfolding *InvariantWatchCharacterization-def*
unfolding *watchCharacterizationCondition-def*
using ‹*clause* < *length* (*getF* *state*)›
using ‹*getWatch1* *?state′* *clause* = *Some* *?w1*›[*THEN* *sym*]
using ‹*getWatch2* *?state′* *clause* = *Some* *?w2*›[*THEN* *sym*]
using ‹*literalFalse* *ww1* (*elements* *M*)›
using ‹*ww1* = *?w1*›
unfolding *setWatch2-def*
unfolding *swapWatches-def*
by *auto*
ultimately
show *?thesis*
using ‹*ww1* = *?w1*›
using ‹*c* = *clause*›
unfolding *setWatch2-def*
unfolding *swapWatches-def*
by *auto*
qed
}
moreover
{
fix *c*::*nat* **and** *ww1*::*Literal* **and** *ww2*::*Literal*
assume *a*: *0* ≤ *c* ∧ *c* < *length* (*getF* *?state″*) ∧ *Some* *ww1* = (*getWatch1* *?state″* *c*) ∧ *Some* *ww2* = (*getWatch2* *?state″* *c*)
assume *b*: *literalFalse* *ww2* (*elements* *M*)

have (∃ *l*. *l* *el* ((*getF* *?state″*) ! *c*) ∧ *literalTrue* *l* (*elements* *M*) ∧ *elementLevel* *l* *M* ≤ *elementLevel* (*opposite* *ww2*) *M*) ∨
(∀ *l*. *l* *el* ((*getF* *?state″*) ! *c*) ∧ *l* ≠ *ww1* ∧ *l* ≠ *ww2* ⟶
literalFalse *l* (*elements* *M*) ∧ *elementLevel* (*opposite* *l*) *M* ≤ *elementLevel* (*opposite* *ww2*) *M*)

proof (*cases c* = *clause*)
case *False*
thus *?thesis*
using *a* **and** *b*
using *Cons*(*9*)
unfolding *InvariantWatchCharacterization-def*
unfolding *watchCharacterizationCondition-def*
unfolding *swapWatches-def*
unfolding *setWatch2-def*
by *auto*
next
case *True*
with *a*
have *ww1* = *?w1* **and** *ww2* = *l′*
using ‹*getWatch1 ?state′ clause* = *Some ?w1*›
using ‹*getWatch2 ?state′ clause* = *Some ?w2*›[*THEN sym*]
unfolding *setWatch2-def*
unfolding *swapWatches-def*
by *auto*
with ‹¬ *literalFalse l′* (*elements* (*getM ?state′*))› *b* *Cons*(*2*)
have *False*
unfolding *swapWatches-def*
by *simp*
thus *?thesis*
by *simp*
qed
}
ultimately
show *?thesis*
unfolding *InvariantWatchCharacterization-def*
unfolding *watchCharacterizationCondition-def*
by *blast*
qed
moreover
have ¬ (∃ *l*. *isUnitClause* (*nth* (*getF state*) *clause*) *l* (*elements* (*getM state*)))

proof−
{
assume ¬ *?thesis*
then obtain *l*
where *isUnitClause* (*nth* (*getF state*) *clause*) *l* (*elements* (*getM state*))
by *auto*
with ‹*l′ el* (*nth* (*getF ?state′*) *clause*)› ‹¬ *literalFalse l′* (*elements* (*getM ?state′*))›
have *l* = *l′*

unfolding *isUnitClause-def*
unfolding *swapWatches-def*
by *auto*
with ‹$l' \neq$ *?w1*› **have**
literalFalse ?w1 (*elements* (*getM ?state'*))
using ‹*isUnitClause* (*nth* (*getF state*) *clause*) *l* (*elements* (*getM state*))›
using ‹*?w1 el* (*nth* (*getF state*) *clause*)›
unfolding *isUnitClause-def*
unfolding *swapWatches-def*
by *simp*
with ‹*?w1* $\neq$ *?w2*› ‹*?w2* = *literal*›
Cons(*2*)
have *literalFalse ?w1* (*elements M*)
unfolding *swapWatches-def*
by *simp*

from ‹*isUnitClause* (*nth* (*getF state*) *clause*) *l* (*elements* (*getM state*))›
Cons(*6*)
have ¬ (∃ *l*. (*l el* (*nth* (*getF state*) *clause*) ∧ *literalTrue l* (*elements* (*getM state*))))
using *containsTrueNotUnit*[*of* - (*nth* (*getF state*) *clause*) *elements* (*getM state*)]
unfolding *InvariantConsistent-def*
by *auto*

from ‹*InvariantWatchCharacterization* (*getF state*) (*getWatch1 state*) (*getWatch2 state*) *M*›
‹*clause* < *length* (*getF state*)›
‹*literalFalse ?w1* (*elements M*)›
‹*getWatch1 ?state' clause* = *Some ?w1*› [*THEN sym*]
‹*getWatch2 ?state' clause* = *Some ?w2*› [*THEN sym*]
have (∃ *l*. *l el* (*getF state* ! *clause*) ∧ *literalTrue l* (*elements M*) ∧ *elementLevel l M* ≤ *elementLevel* (*opposite ?w1*) *M*) ∨
(∀ *l*. *l el* (*getF state* ! *clause*) ∧ *l* ≠ *?w1* ∧ *l* ≠ *?w2* ⟶ *literalFalse l* (*elements M*))
unfolding *InvariantWatchCharacterization-def*
unfolding *watchCharacterizationCondition-def*
unfolding *swapWatches-def*
by *auto*
with ‹¬ (∃ *l*. (*l el* (*nth* (*getF state*) *clause*) ∧ *literalTrue l* (*elements* (*getM state*))))›
Cons(*2*)
have (∀ *l*. *l el* (*getF state* ! *clause*) ∧ *l* ≠ *?w1* ∧ *l* ≠ *?w2* ⟶ *literalFalse l* (*elements M*))
by *auto*
with ‹l' *el* (*getF ?state'* ! *clause*)› ‹$l' \neq$ *?w1*› ‹$l' \neq$ *?w2*› ‹¬ *literalFalse* l' (*elements* (*getM ?state'*))›

Cons(*2*)
have *False*
unfolding *swapWatches-def*
by *simp*
}
thus *?thesis*
by *auto*
qed
ultimately
show *?thesis*
using *Cons*(*1*)[*of ?state'' newWl*]
using *Cons*(*2*) *Cons*(*5*) *Cons*(*6*)
using ‹*getWatch1 ?state' clause = Some ?w1*›
using ‹*getWatch2 ?state' clause = Some ?w2*›
using ‹*Some literal = getWatch1 state clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state'*))›
using ‹*uniq Wl'*›
using *Some*
by (*simp add*: *Let-def*)
next
case *None*
hence ∀ *l. l el* (*nth* (*getF ?state'*) *clause*) ∧ *l* ≠ *?w1* ∧ *l* ≠ *?w2* ⟶ *literalFalse l* (*elements* (*getM ?state'*))
using *getNonWatchedUnfalsifiedLiteralNoneCharacterization*
by *simp*
show *?thesis*
proof (*cases literalFalse ?w1* (*elements* (*getM ?state'*)))
case *True*
let *?state'' = ?state'*⦇*getConflictFlag* := *True*, *getConflictClause* := *clause*⦈

from *Cons*(*3*)
have *InvariantWatchesEl* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)
unfolding *InvariantWatchesEl-def*
unfolding *swapWatches-def*
by *auto*
moreover
from *Cons*(*4*)
have *InvariantWatchesDiffer* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)
unfolding *InvariantWatchesDiffer-def*
unfolding *swapWatches-def*
by *auto*
moreover
from *Cons*(*6*)
have *InvariantConsistent* (*getM ?state''*)
unfolding *swapWatches-def*
by *simp*

moreover
have *getM ?state″ = getM state ∧*
getF ?state″ = getF state ∧
getQ ?state″ = getQ state ∧
getSATFlag ?state″ = getSATFlag state
unfolding *swapWatches-def*
by *simp*
moreover
have ∀ *c. c ∈ set Wl′ ⟶ Some literal = getWatch1 ?state″ c ∨ Some literal = getWatch2 ?state″ c*
using *Cons*(*7*)
using ‹*clause ∉ set Wl′*›
unfolding *swapWatches-def*
by *auto*
moreover
have *InvariantWatchCharacterization* (*getF ?state″*) (*getWatch1 ?state″*) (*getWatch2 ?state″*) *M*
using *Cons*(*9*)
unfolding *swapWatches-def*
unfolding *InvariantWatchCharacterization-def*
by *auto*
moreover
have *clauseFalse* (*nth* (*getF state*) *clause*) (*elements* (*getM state*))
using ‹∀ *l. l el* (*nth* (*getF ?state′*) *clause*) ∧ *l* ≠ *?w1* ∧ *l* ≠ *?w2* ⟶ *literalFalse l* (*elements* (*getM ?state′*))›
using ‹*literalFalse ?w1* (*elements* (*getM ?state′*))›
using ‹*literalFalse ?w2* (*elements* (*getM state*))›
unfolding *swapWatches-def*
by (*auto simp add*: *clauseFalseIffAllLiteralsAreFalse*)
hence ¬ (∃ *l. isUnitClause* (*nth* (*getF state*) *clause*) *l* (*elements* (*getM state*)))
unfolding *isUnitClause-def*
by (*simp add*: *clauseFalseIffAllLiteralsAreFalse*)
ultimately
show *?thesis*
using *Cons*(*1*)[*of ?state″ clause # newWl*]
using *Cons*(*2*) *Cons*(*5*) *Cons*(*6*)
using ‹*getWatch1 ?state′ clause = Some ?w1*›
using ‹*getWatch2 ?state′ clause = Some ?w2*›
using ‹*Some literal = getWatch1 state clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state′*))›
using *None*
using ‹*literalFalse ?w1* (*elements* (*getM ?state′*))›
using ‹*uniq Wl′*›
by (*simp add*: *Let-def*)
next
case *False*
let *?state″ = setReason ?w1 clause* (*?state′*⦇*getQ* := (*if ?w1*

el (*getQ ?state'*) *then* (*getQ ?state'*) *else* (*getQ ?state'*) @ [*?w1*])⦈)

from *Cons*(*3*)
have *InvariantWatchesEl* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)
unfolding *InvariantWatchesEl-def*
unfolding *swapWatches-def*
unfolding *setReason-def*
by *auto*
moreover
from *Cons*(*4*)
have *InvariantWatchesDiffer* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)
unfolding *InvariantWatchesDiffer-def*
unfolding *swapWatches-def*
unfolding *setReason-def*
by *auto*
moreover
from *Cons*(*6*)
have *InvariantConsistent* (*getM ?state''*)
unfolding *swapWatches-def*
unfolding *setReason-def*
by *simp*
moreover
have *getM ?state''* = *getM state* ∧
getF ?state'' = *getF state* ∧
getSATFlag ?state'' = *getSATFlag state* ∧
getQ ?state'' = (*if ?w1 el* (*getQ state*) *then* (*getQ state*) *else* (*getQ state* @ [*?w1*]))
unfolding *swapWatches-def*
unfolding *setReason-def*
by *simp*
moreover
have ∀ *c*. *c* ∈ *set Wl'* ⟶ *Some literal* = *getWatch1 ?state'' c* ∨ *Some literal* = *getWatch2 ?state'' c*
using *Cons*(*7*)
using ‹*clause* ∉ *set Wl'*›
unfolding *swapWatches-def*
unfolding *setReason-def*
by *auto*
moreover
have *InvariantWatchCharacterization* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*) *M*
using *Cons*(*9*)
unfolding *swapWatches-def*
unfolding *setReason-def*
unfolding *InvariantWatchCharacterization-def*
by *auto*
ultimately

have *let state′ = notifyWatches-loop literal Wl′ (clause # newWl) ?state′′ in*
?Cond1 state′ ?state′′ ∧ ?Cond2 Wl′ state′ ?state′′
using *Cons(1)[of ?state′′ clause # newWl]*
using *Cons(2) Cons(5)*
using ‹*uniq Wl′*›
by (*simp add: Let-def*)
moreover
have *notifyWatches-loop literal Wl′ (clause # newWl) ?state′′ = notifyWatches-loop literal (clause # Wl′) newWl state*
using ‹*getWatch1 ?state′ clause = Some ?w1*›
using ‹*getWatch2 ?state′ clause = Some ?w2*›
using ‹*Some literal = getWatch1 state clause*›
using ‹¬ *literalTrue ?w1 (elements (getM ?state′))*›
using *None*
using ‹¬ *literalFalse ?w1 (elements (getM ?state′))*›
by (*simp add: Let-def*)
ultimately
have *let state′ = notifyWatches-loop literal (clause # Wl′) newWl state in*
?Cond1 state′ ?state′′ ∧ ?Cond2 Wl′ state′ ?state′′
by *simp*

have *isUnitClause (nth (getF state) clause) ?w1 (elements (getM state))*
using ‹∀ *l. l el (nth (getF ?state′) clause) ∧ l ≠ ?w1 ∧ l ≠ ?w2 ⟶ literalFalse l (elements (getM ?state′))*›
using ‹*?w1 el (nth (getF state) clause)*›
using ‹*?w2 el (nth (getF state) clause)*›
using ‹*literalFalse ?w2 (elements (getM state))*›
using ‹¬ *literalFalse ?w1 (elements (getM ?state′))*›
using ‹¬ *literalTrue ?w1 (elements (getM ?state′))*›
unfolding *swapWatches-def*
unfolding *isUnitClause-def*
by *auto*

show *?thesis*
proof−
{
fix *l::Literal*
assume *let state′ = notifyWatches-loop literal (clause # Wl′) newWl state in*
l ∈ set (getQ state′) − set (getQ state)
have ∃ *clause. clause el (getF state) ∧ literal el clause ∧ isUnitClause clause l (elements (getM state))*
proof (*cases l ≠ ?w1*)
case *True*
hence *let state′ = notifyWatches-loop literal (clause # Wl′) newWl state in*

l ∈ *set* (*getQ state′*) − *set* (*getQ ?state″*)
using ‹*let state′* = *notifyWatches-loop literal* (*clause* # *Wl′*) *newWl state in*
l ∈ *set* (*getQ state′*) − *set* (*getQ state*)›
unfolding *setReason-def*
unfolding *swapWatches-def*
by (*simp add*:*Let-def*)
with ‹*let state′* = *notifyWatches-loop literal* (*clause* # *Wl′*) *newWl state in*
?Cond1 state′ ?state″ ∧ *?Cond2 Wl′ state′ ?state″*›
show *?thesis*
unfolding *setReason-def*
unfolding *swapWatches-def*
by (*simp add*:*Let-def del*: *notifyWatches-loop.simps*)
next
case *False*
thus *?thesis*
using ‹(*nth* (*getF state*) *clause*) *el* (*getF state*)›
‹*?w2* = *literal*›
‹*?w2 el* (*nth* (*getF state*) *clause*)›
‹*isUnitClause* (*nth* (*getF state*) *clause*) *?w1* (*elements* (*getM state*))›
by (*auto simp add*:*Let-def*)
qed
}
hence *let state′* = *notifyWatches-loop literal* (*clause* # *Wl′*) *newWl state in*
?Cond1 state′ state
by *simp*
moreover
{
fix *c*
assume *c* ∈ *set* (*clause* # *Wl′*)
have *let state′* = *notifyWatches-loop literal* (*clause* # *Wl′*) *newWl state in*
∀ *l*. *isUnitClause* (*nth* (*getF state*) *c*) *l* (*elements* (*getM state*)) ⟶ *l* ∈ *set* (*getQ state′*)
proof (*cases c* = *clause*)
case *True*
{
fix *l*::*Literal*
assume *isUnitClause* (*nth* (*getF state*) *c*) *l* (*elements* (*getM state*))
with ‹*isUnitClause* (*nth* (*getF state*) *clause*) *?w1* (*elements* (*getM state*))› ‹*c* = *clause*›
have *l* = *?w1*
unfolding *isUnitClause-def*
by *auto*
have *isPrefix* (*getQ ?state″*) (*getQ* (*notifyWatches-loop*

literal Wl′ (*clause* # *newWl*) *?state′′*))
using ‹*InvariantWatchesEl* (*getF ?state′′*) (*getWatch1 ?state′′*) (*getWatch2 ?state′′*)›
using *notifyWatchesLoopPreservedVariables*[*of ?state′′ Wl′ literal clause # newWl*]
using *Cons*(*5*)
unfolding *swapWatches-def*
unfolding *setReason-def*
by (*simp add*: *Let-def*)
hence *set* (*getQ ?state′′*) ⊆ *set* (*getQ* (*notifyWatches-loop literal Wl′* (*clause* # *newWl*) *?state′′*))
using *prefixIsSubset*[*of getQ ?state′′ getQ* (*notifyWatches-loop literal Wl′* (*clause* # *newWl*) *?state′′*)]
by *auto*
hence *l* ∈ *set* (*getQ* (*notifyWatches-loop literal Wl′* (*clause* # *newWl*) *?state′′*))
using ‹*l* = *?w1*›
unfolding *swapWatches-def*
unfolding *setReason-def*
by *auto*
}
thus *?thesis*
using ‹*notifyWatches-loop literal Wl′* (*clause* # *newWl*) *?state′′* = *notifyWatches-loop literal* (*clause* # *Wl′*) *newWl state*›
by (*simp add*:*Let-def*)
next
case *False*
hence *c* ∈ *set Wl′*
using ‹*c* ∈ *set* (*clause* # *Wl′*)›
by *simp*
{
fix *l*::*Literal*
assume *isUnitClause* (*nth* (*getF state*) *c*) *l* (*elements* (*getM state*))
hence *isUnitClause* (*nth* (*getF ?state′′*) *c*) *l* (*elements* (*getM ?state′′*))
unfolding *setReason-def*
unfolding *swapWatches-def*
by *simp*
with ‹*let state′* = *notifyWatches-loop literal* (*clause* # *Wl′*) *newWl state in*
?Cond1 state′ ?state′′ ∧ *?Cond2 Wl′ state′ ?state′′*›
‹*c* ∈ *set Wl′*›
have *let state′* = *notifyWatches-loop literal* (*clause* # *Wl′*) *newWl state in l* ∈ *set* (*getQ state′*)
by (*simp add*:*Let-def*)
}
thus *?thesis*
by (*simp add*:*Let-def*)

```
          qed
        }
          hence ?Cond2 (clause # Wl') (notifyWatches-loop literal
(clause # Wl') newWl state) state
          by (simp add: Let-def)
        ultimately
        show ?thesis
          by (simp add:Let-def)
      qed
    qed
  qed
 qed
next
  case False
  let ?state' = state
  let ?w1 = wa
  have getWatch1 ?state' clause = Some ?w1
    using ‹getWatch1 state clause = Some wa›
    unfolding swapWatches-def
    by auto
  let ?w2 = wb
  have getWatch2 ?state' clause = Some ?w2
    using ‹getWatch2 state clause = Some wb›
    unfolding swapWatches-def
    by auto


  from ‹¬ Some literal = getWatch1 state clause›
    ‹∀ (c::nat). c ∈ set (clause # Wl') ⟶ Some literal = (getWatch1
state c)  ∨ Some literal = (getWatch2 state c)›
  have Some literal = getWatch2 state clause
    by auto
  hence ?w2 = literal
    using ‹getWatch2 ?state' clause = Some ?w2›
    by simp
  hence literalFalse ?w2 (elements (getM state))
    using Cons(2)
    by simp

 from ‹InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2
state)›
   have ?w1 el (nth (getF state) clause) ?w2 el (nth (getF state)
clause)
    using ‹getWatch1 ?state' clause = Some ?w1›
    using ‹getWatch2 ?state' clause = Some ?w2›
    using ‹clause < length (getF state)›
    unfolding InvariantWatchesEl-def
    unfolding swapWatches-def
    by auto
```

from ‹*InvariantWatchesDiffer* (*getF state*) (*getWatch1 state*) (*getWatch2 state*)›
have *?w1* ≠ *?w2*
using ‹*getWatch1 ?state′ clause* = *Some ?w1*›
using ‹*getWatch2 ?state′ clause* = *Some ?w2*›
using ‹*clause* < *length* (*getF state*)›
unfolding *InvariantWatchesDiffer-def*
unfolding *swapWatches-def*
by *auto*

show *?thesis*
proof (*cases literalTrue ?w1* (*elements* (*getM ?state′*)))
case *True*
have ¬ (∃ *l*. *isUnitClause* (*nth* (*getF state*) *clause*) *l* (*elements* (*getM state*)))
using ‹*?w1 el* (*nth* (*getF state*) *clause*)›
using ‹*literalTrue ?w1* (*elements* (*getM ?state′*))›
using ‹*InvariantConsistent* (*getM state*)›
unfolding *InvariantConsistent-def*
by (*auto simp add*: *isUnitClause-def inconsistentCharacterization*)
thus *?thesis*
using *True*
using *Cons*(*1*)[*of ?state′ clause* # *newWl*]
using *Cons*(*2*) *Cons*(*3*) *Cons*(*4*) *Cons*(*5*) *Cons*(*6*) *Cons*(*7*) *Cons*(*8*) *Cons*(*9*)
using ‹¬ *Some literal* = *getWatch1 state clause*›
using ‹*getWatch1 ?state′ clause* = *Some ?w1*›
using ‹*getWatch2 ?state′ clause* = *Some ?w2*›
using ‹*literalTrue ?w1* (*elements* (*getM ?state′*))›
using ‹*uniq Wl′*›
by (*simp add*:*Let-def*)
next
case *False*
show *?thesis*
proof (*cases getNonWatchedUnfalsifiedLiteral* (*nth* (*getF ?state′*) *clause*) *?w1 ?w2* (*getM ?state′*))
case (*Some l′*)
hence *l′ el* (*nth* (*getF ?state′*) *clause*) ¬ *literalFalse l′* (*elements* (*getM ?state′*)) *l′* ≠ *?w1 l′* ≠ *?w2*
using *getNonWatchedUnfalsifiedLiteralSomeCharacterization*
by *auto*

let *?state′′* = *setWatch2 clause l′ ?state′*

from *Cons*(*3*)
have *InvariantWatchesEl* (*getF ?state′′*) (*getWatch1 ?state′′*) (*getWatch2 ?state′′*)

using ‹*l′ el* (*nth* (*getF ?state′*) *clause*)›
unfolding *InvariantWatchesEl-def*
unfolding *setWatch2-def*
by *auto*
moreover
from *Cons*(*4*)
have *InvariantWatchesDiffer* (*getF ?state″*) (*getWatch1 ?state″*) (*getWatch2 ?state″*)
using ‹$l' \neq$ *?w1*›
using ‹*getWatch1 ?state′ clause* = *Some ?w1*›
using ‹*getWatch2 ?state′ clause* = *Some ?w2*›
unfolding *InvariantWatchesDiffer-def*
unfolding *setWatch2-def*
by *auto*
moreover
from *Cons*(*6*)
have *InvariantConsistent* (*getM ?state″*)
unfolding *setWatch2-def*
by *simp*
moreover
have *getM ?state″* = *getM state* ∧
getF ?state″ = *getF state* ∧
getQ ?state″ = *getQ state* ∧
getConflictFlag ?state″ = *getConflictFlag state*
unfolding *setWatch2-def*
by *simp*
moreover
have ∀ *c*. *c* ∈ *set Wl′* ⟶ *Some literal* = *getWatch1 ?state″ c* ∨ *Some literal* = *getWatch2 ?state″ c*
using *Cons*(*7*)
using ‹*clause* ∉ *set Wl′*›
unfolding *setWatch2-def*
by *auto*
moreover
have *InvariantWatchCharacterization* (*getF ?state″*) (*getWatch1 ?state″*) (*getWatch2 ?state″*) *M*
proof−
{
fix *c*::*nat* **and** *ww1*::*Literal* **and** *ww2*::*Literal*
assume *a*: *0* ≤ *c* ∧ *c* < *length* (*getF ?state″*) ∧ *Some ww1* = (*getWatch1 ?state″ c*) ∧ *Some ww2* = (*getWatch2 ?state″ c*)
assume *b*: *literalFalse ww1* (*elements M*)

have (∃ *l*. *l el* ((*getF ?state″*) ! *c*) ∧ *literalTrue l* (*elements M*) ∧ *elementLevel l M* ≤ *elementLevel* (*opposite ww1*) *M*) ∨
(∀ *l*. *l el* (*getF ?state″* ! *c*) ∧ *l* ≠ *ww1* ∧ *l* ≠ *ww2* ⟶
literalFalse l (*elements M*) ∧ *elementLevel* (*opposite l*) *M* ≤ *elementLevel* (*opposite ww1*) *M*)
proof (*cases c* = *clause*)

case *False*
thus *?thesis*
using *a* **and** *b*
using *Cons*(*9*)
unfolding *InvariantWatchCharacterization-def*
unfolding *watchCharacterizationCondition-def*
unfolding *setWatch2-def*
by *auto*
next
case *True*
with *a*
have *ww1* = *?w1* **and** *ww2* = *l′*
using ‹*getWatch1 ?state′ clause* = *Some ?w1*›
using ‹*getWatch2 ?state′ clause* = *Some ?w2*›[*THEN sym*]
unfolding *setWatch2-def*
by *auto*

have ¬ (∀ *l*. *l el* (*getF state* ! *clause*) ∧ *l* ≠ *?w1* ∧ *l* ≠ *?w2* ⟶ *literalFalse l* (*elements M*))
using ‹*l′* ≠ *?w1*› **and** ‹*l′* ≠ *?w2*› ‹*l′ el* (*nth* (*getF ?state′*) *clause*)›
using ‹¬ *literalFalse l′* (*elements* (*getM ?state′*))›
using *Cons*(*2*)
using *a* **and** *b*
using ‹*c* = *clause*›
unfolding *setWatch2-def*
by *auto*
moreover
have (∃ *l*. *l el* (*getF state* ! *clause*) ∧ *literalTrue l* (*elements M*) ∧ *elementLevel l M* ≤ *elementLevel* (*opposite ?w1*) *M*) ∨
(∀ *l*. *l el* (*getF state* ! *clause*) ∧ *l* ≠ *?w1* ∧ *l* ≠ *?w2* ⟶ *literalFalse l* (*elements M*))
using *Cons*(*9*)
unfolding *InvariantWatchCharacterization-def*
unfolding *watchCharacterizationCondition-def*
using ‹*clause* < *length* (*getF state*)›
using ‹*getWatch1 ?state′ clause* = *Some ?w1*›[*THEN sym*]
using ‹*getWatch2 ?state′ clause* = *Some ?w2*›[*THEN sym*]
using ‹*literalFalse ww1* (*elements M*)›
using ‹*ww1* = *?w1*›
unfolding *setWatch2-def*
by *auto*
ultimately
show *?thesis*
using ‹*ww1* = *?w1*›
using ‹*c* = *clause*›

```
        unfolding setWatch2-def
        by auto
      qed
    }
    moreover
    {
      fix c::nat and ww1::Literal and ww2::Literal
      assume a: 0 ≤ c ∧ c < length (getF ?state'') ∧ Some ww1
= (getWatch1 ?state'' c) ∧ Some ww2 = (getWatch2 ?state'' c)
      assume b: literalFalse ww2 (elements M)

      have (∃ l.  l el ((getF ?state'') ! c) ∧ literalTrue l (elements
M) ∧ elementLevel l M ≤ elementLevel (opposite ww2) M) ∨
          (∀ l. l el (getF ?state'' ! c) ∧ l ≠ ww1 ∧ l ≠ ww2 ⟶
              literalFalse l (elements M) ∧ elementLevel (opposite l)
M ≤ elementLevel (opposite ww2) M)
      proof (cases c = clause)
        case False
        thus ?thesis
          using a and b
          using Cons(9)
          unfolding InvariantWatchCharacterization-def
          unfolding watchCharacterizationCondition-def
          unfolding setWatch2-def
          by auto
      next
        case True
        with a
        have ww1 = ?w1 and ww2 = l'
          using ‹getWatch1 ?state' clause = Some ?w1›
            using ‹getWatch2 ?state' clause = Some ?w2›[THEN
sym]
          unfolding setWatch2-def
          by auto
        with ‹¬ literalFalse l' (elements (getM ?state'))› b
        Cons(2)
        have False
          unfolding setWatch2-def
          by simp
        thus ?thesis
          by simp
      qed
    }
    ultimately
    show ?thesis
      unfolding InvariantWatchCharacterization-def
      unfolding watchCharacterizationCondition-def
      by blast
  qed
```

moreover
have ¬ (∃ *l*. *isUnitClause* (*nth* (*getF state*) *clause*) *l* (*elements* (*getM state*)))

proof−
{
assume ¬ *?thesis*
then obtain *l*
where *isUnitClause* (*nth* (*getF state*) *clause*) *l* (*elements* (*getM state*))
by *auto*
with ‹*l′ el* (*nth* (*getF ?state′*) *clause*)› ‹¬ *literalFalse l′* (*elements* (*getM ?state′*))›
have *l* = *l′*
unfolding *isUnitClause-def*
by *auto*
with ‹*l′* ≠ *?w1*› **have**
literalFalse ?w1 (*elements* (*getM ?state′*))
using ‹*isUnitClause* (*nth* (*getF state*) *clause*) *l* (*elements* (*getM state*))›
using ‹*?w1 el* (*nth* (*getF state*) *clause*)›
unfolding *isUnitClause-def*
by *simp*
with ‹*?w1* ≠ *?w2*› ‹*?w2* = *literal*›
Cons(*2*)
have *literalFalse ?w1* (*elements M*)
by *simp*

from ‹*isUnitClause* (*nth* (*getF state*) *clause*) *l* (*elements* (*getM state*))›
Cons(*6*)
have ¬ (∃ *l*. (*l el* (*nth* (*getF state*) *clause*) ∧ *literalTrue l* (*elements* (*getM state*))))
using *containsTrueNotUnit*[*of* - (*nth* (*getF state*) *clause*) *elements* (*getM state*)]
unfolding *InvariantConsistent-def*
by *auto*

from ‹*InvariantWatchCharacterization* (*getF state*) (*getWatch1 state*) (*getWatch2 state*) *M*›
‹*clause* < *length* (*getF state*)›
‹*literalFalse ?w1* (*elements M*)›
‹*getWatch1 ?state′ clause* = *Some ?w1*› [*THEN sym*]
‹*getWatch2 ?state′ clause* = *Some ?w2*› [*THEN sym*]
have (∃ *l*. *l el* (*getF state* ! *clause*) ∧ *literalTrue l* (*elements M*) ∧ *elementLevel l M* ≤ *elementLevel* (*opposite ?w1*) *M*) ∨
(∀ *l*. *l el* (*getF state* ! *clause*) ∧ *l* ≠ *?w1* ∧ *l* ≠ *?w2* ⟶ *literalFalse l* (*elements M*))
unfolding *InvariantWatchCharacterization-def*

unfolding *watchCharacterizationCondition-def*
unfolding *swapWatches-def*
by *auto*
with ‹¬ (∃ *l*. (*l el* (*nth* (*getF state*) *clause*) ∧ *literalTrue l* (*elements* (*getM state*))))›
Cons(*2*)
have (∀ *l*. *l el* (*getF state* ! *clause*) ∧ *l* ≠ *?w1* ∧ *l* ≠ *?w2* ⟶ *literalFalse l* (*elements M*))
by *auto*
with ‹*l′ el* (*getF ?state′* ! *clause*)› ‹*l′* ≠ *?w1*› ‹*l′* ≠ *?w2*› ‹¬ *literalFalse l′* (*elements* (*getM ?state′*))›
Cons(*2*)
have *False*
unfolding *swapWatches-def*
by *simp*
}
thus *?thesis*
by *auto*
qed
ultimately
show *?thesis*
using *Cons*(*1*)[*of ?state″ newWl*]
using *Cons*(*2*) *Cons*(*5*) *Cons*(*7*)
using ‹*getWatch1 ?state′ clause* = *Some ?w1*›
using ‹*getWatch2 ?state′ clause* = *Some ?w2*›
using ‹¬ *Some literal* = *getWatch1 state clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state′*))›
using ‹*uniq Wl′*›
using *Some*
by (*simp add*: *Let-def*)
next
case *None*
hence ∀ *l*. *l el* (*nth* (*getF ?state′*) *clause*) ∧ *l* ≠ *?w1* ∧ *l* ≠ *?w2* ⟶ *literalFalse l* (*elements* (*getM ?state′*))
using *getNonWatchedUnfalsifiedLiteralNoneCharacterization*
by *simp*
show *?thesis*
proof (*cases literalFalse ?w1* (*elements* (*getM ?state′*)))
case *True*
let *?state″* = *?state′*⦇*getConflictFlag* := *True*, *getConflictClause* := *clause*⦈

from *Cons*(*3*)
have *InvariantWatchesEl* (*getF ?state″*) (*getWatch1 ?state″*) (*getWatch2 ?state″*)
unfolding *InvariantWatchesEl-def*
by *auto*
moreover
from *Cons*(*4*)

have *InvariantWatchesDiffer* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)
unfolding *InvariantWatchesDiffer-def*
by *auto*
moreover
from *Cons*(*6*)
have *InvariantConsistent* (*getM ?state''*)
unfolding *setWatch2-def*
by *simp*
moreover
have *getM ?state''* = *getM state* ∧
getF ?state'' = *getF state* ∧
getSATFlag ?state'' = *getSATFlag state*
by *simp*
moreover
have ∀ *c*. *c* ∈ *set Wl'* ⟶ *Some literal* = *getWatch1 ?state'' c* ∨ *Some literal* = *getWatch2 ?state'' c*
using *Cons*(*7*)
using ‹*clause* ∉ *set Wl'*›
unfolding *setWatch2-def*
by *auto*
moreover
have *InvariantWatchCharacterization* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*) *M*
using *Cons*(*9*)
unfolding *InvariantWatchCharacterization-def*
by *auto*
moreover
have *clauseFalse* (*nth* (*getF state*) *clause*) (*elements* (*getM state*))
using ‹∀ *l*. *l el* (*nth* (*getF ?state'*) *clause*) ∧ *l* ≠ *?w1* ∧ *l* ≠ *?w2* ⟶ *literalFalse l* (*elements* (*getM ?state'*))›
using ‹*literalFalse ?w1* (*elements* (*getM ?state'*))›
using ‹*literalFalse ?w2* (*elements* (*getM state*))›
unfolding *swapWatches-def*
by (*auto simp add*: *clauseFalseIffAllLiteralsAreFalse*)
hence ¬ (∃ *l*. *isUnitClause* (*nth* (*getF state*) *clause*) *l* (*elements* (*getM state*)))
unfolding *isUnitClause-def*
by (*simp add*: *clauseFalseIffAllLiteralsAreFalse*)
ultimately
show *?thesis*
using *Cons*(*1*)[*of ?state'' clause # newWl*]
using *Cons*(*2*) *Cons*(*5*) *Cons*(*7*)
using ‹*getWatch1 ?state' clause* = *Some ?w1*›
using ‹*getWatch2 ?state' clause* = *Some ?w2*›
using ‹¬ *Some literal* = *getWatch1 state clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state'*))›
using *None*

using ‹*literalFalse ?w1* (*elements* (*getM ?state′*))›
using ‹*uniq Wl′*›
by (*simp add*: *Let-def*)
next
case *False*
let *?state′′* = *setReason ?w1 clause* (*?state′*⦇*getQ* := (*if ?w1 el* (*getQ ?state′*) *then* (*getQ ?state′*) *else* (*getQ ?state′*) @ [*?w1*])⦈)

from *Cons*(*3*)
have *InvariantWatchesEl* (*getF ?state′′*) (*getWatch1 ?state′′*) (*getWatch2 ?state′′*)
unfolding *InvariantWatchesEl-def*
unfolding *setReason-def*
by *auto*
moreover
from *Cons*(*4*)
have *InvariantWatchesDiffer* (*getF ?state′′*) (*getWatch1 ?state′′*) (*getWatch2 ?state′′*)
unfolding *InvariantWatchesDiffer-def*
unfolding *setReason-def*
by *auto*
moreover
from *Cons*(*6*)
have *InvariantConsistent* (*getM ?state′′*)
unfolding *setReason-def*
by *simp*
moreover
have *getM ?state′′* = *getM state* ∧
getF ?state′′ = *getF state* ∧
getSATFlag ?state′′ = *getSATFlag state*
unfolding *setReason-def*
by *simp*
moreover
have ∀ *c. c* ∈ *set Wl′* ⟶ *Some literal* = *getWatch1 ?state′′ c* ∨ *Some literal* = *getWatch2 ?state′′ c*
using *Cons*(*7*)
using ‹*clause* ∉ *set Wl′*›
unfolding *setReason-def*
by *auto*
moreover
have *InvariantWatchCharacterization* (*getF ?state′′*) (*getWatch1 ?state′′*) (*getWatch2 ?state′′*) *M*
using *Cons*(*9*)
unfolding *InvariantWatchCharacterization-def*
unfolding *setReason-def*
by *auto*
ultimately
have *let state′* = *notifyWatches-loop literal Wl′* (*clause* # *newWl*) *?state′′ in*

?Cond1 state′ ?state′′ ∧ *?Cond2 Wl′ state′ ?state′′*
using *Cons*(*1*)[*of ?state′′ clause # newWl*]
using *Cons*(*2*) *Cons*(*5*) *Cons*(*6*) *Cons*(*7*)
using ‹*uniq Wl′*›
by (*simp add*: *Let-def*)
moreover
have *notifyWatches-loop literal Wl′* (*clause # newWl*) *?state′′* = *notifyWatches-loop literal* (*clause # Wl′*) *newWl state*
using ‹*getWatch1 ?state′ clause* = *Some ?w1*›
using ‹*getWatch2 ?state′ clause* = *Some ?w2*›
using ‹¬ *Some literal* = *getWatch1 state clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state′*))›
using *None*
using ‹¬ *literalFalse ?w1* (*elements* (*getM ?state′*))›
by (*simp add*: *Let-def*)
ultimately
have *let state′* = *notifyWatches-loop literal* (*clause # Wl′*) *newWl state in*
?Cond1 state′ ?state′′ ∧ *?Cond2 Wl′ state′ ?state′′*
by *simp*

have *isUnitClause* (*nth* (*getF state*) *clause*) *?w1* (*elements* (*getM state*))
using ‹∀ *l*. *l el* (*nth* (*getF ?state′*) *clause*) ∧ *l* ≠ *?w1* ∧ *l* ≠ *?w2* ⟶ *literalFalse l* (*elements* (*getM ?state′*))›
using ‹*?w1 el* (*nth* (*getF state*) *clause*)›
using ‹*?w2 el* (*nth* (*getF state*) *clause*)›
using ‹*literalFalse ?w2* (*elements* (*getM state*))›
using ‹¬ *literalFalse ?w1* (*elements* (*getM ?state′*))›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state′*))›
unfolding *swapWatches-def*
unfolding *isUnitClause-def*
by *auto*

show *?thesis*
proof−
{
fix *l*::*Literal*
assume *let state′* = *notifyWatches-loop literal* (*clause # Wl′*) *newWl state in*
l ∈ *set* (*getQ state′*) − *set* (*getQ state*)
have ∃ *clause*. *clause el* (*getF state*) ∧ *literal el clause* ∧ *isUnitClause clause l* (*elements* (*getM state*))
proof (*cases l* ≠ *?w1*)
case *True*
hence *let state′* = *notifyWatches-loop literal* (*clause # Wl′*) *newWl state in*
l ∈ *set* (*getQ state′*) − *set* (*getQ ?state′′*)
using ‹*let state′* = *notifyWatches-loop literal* (*clause #*

```
Wl′) newWl state in
              l ∈ set (getQ state′) − set (getQ state)›
             unfolding setReason-def
             unfolding swapWatches-def
             by (simp add:Let-def)
             with ‹let state′ = notifyWatches-loop literal (clause #
Wl′) newWl state in
              ?Cond1 state′ ?state′′ ∧ ?Cond2 Wl′ state′ ?state′′›
           show ?thesis
             unfolding setReason-def
             unfolding swapWatches-def
             by (simp add:Let-def del: notifyWatches-loop.simps)
         next
           case False
           thus ?thesis
                using ‹(nth (getF state) clause) el (getF state)›
‹isUnitClause (nth (getF state) clause) ?w1 (elements (getM state))›
                ‹?w2 = literal›
                ‹?w2 el (nth (getF state) clause)›
             by (auto simp add:Let-def)
         qed
       }
       hence let state′ = notifyWatches-loop literal (clause # Wl′)
newWl state in
           ?Cond1 state′ state
         by simp
       moreover
       {
         fix c
         assume c ∈ set (clause # Wl′)
        have let state′ = notifyWatches-loop literal (clause # Wl′)
newWl state in
           ∀ l. isUnitClause (nth (getF state) c) l (elements (getM
state)) ⟶ l ∈ set (getQ state′)
         proof (cases c = clause)
           case True
           {
             fix l::Literal
              assume isUnitClause (nth (getF state) c) l (elements
(getM state))
                  with ‹isUnitClause (nth (getF state) clause) ?w1
(elements (getM state))› ‹c = clause›
             have l = ?w1
               unfolding isUnitClause-def
               by auto
             have isPrefix (getQ ?state′′) (getQ (notifyWatches-loop
literal Wl′ (clause # newWl) ?state′′))
              using ‹InvariantWatchesEl (getF ?state′′) (getWatch1
?state′′) (getWatch2 ?state′′)›
```

using *notifyWatchesLoopPreservedVariables*[*of ?state''*
Wl' literal clause # newWl]
using *Cons*(*5*)
unfolding *swapWatches-def*
unfolding *setReason-def*
by (*simp add*: *Let-def*)
hence *set* (*getQ ?state''*) ⊆ *set* (*getQ* (*notifyWatches-loop*
literal Wl' (*clause # newWl*) *?state''*))
using *prefixIsSubset*[*of getQ ?state'' getQ* (*notifyWatches-loop*
literal Wl' (*clause # newWl*) *?state''*)]
by *auto*
hence *l* ∈ *set* (*getQ* (*notifyWatches-loop literal Wl'*
(*clause # newWl*) *?state''*))
using ‹*l* = *?w1*›
unfolding *swapWatches-def*
unfolding *setReason-def*
by *auto*
}
thus *?thesis*
using ‹*notifyWatches-loop literal Wl'* (*clause # newWl*)
?state'' = *notifyWatches-loop literal* (*clause # Wl'*) *newWl state*›
by (*simp add*:*Let-def*)
next
case *False*
hence *c* ∈ *set Wl'*
using ‹*c* ∈ *set* (*clause # Wl'*)›
by *simp*
{
fix *l*::*Literal*
assume *isUnitClause* (*nth* (*getF state*) *c*) *l* (*elements*
(*getM state*))
hence *isUnitClause* (*nth* (*getF ?state''*) *c*) *l* (*elements*
(*getM ?state''*))
unfolding *setReason-def*
unfolding *swapWatches-def*
by *simp*
with ‹*let state'* = *notifyWatches-loop literal* (*clause #*
Wl') *newWl state in*
?Cond1 state' ?state'' ∧ *?Cond2 Wl' state' ?state''*›
‹*c* ∈ *set Wl'*›
have *let state'* = *notifyWatches-loop literal* (*clause #*
Wl') *newWl state in l* ∈ *set* (*getQ state'*)
by (*simp add*:*Let-def*)
}
thus *?thesis*
by (*simp add*:*Let-def*)
qed
}
hence *?Cond2* (*clause # Wl'*) (*notifyWatches-loop literal*

(*clause* # *Wl'*) *newWl state*) *state*
by (*simp add*: *Let-def*)
ultimately
show *?thesis*
by (*simp add*:*Let-def*)
qed
qed
qed
qed
qed
qed

lemma *InvariantUniqQAfterNotifyWatchesLoop*:
fixes *literal* :: *Literal* **and** *Wl* :: *nat list* **and** *newWl* :: *nat list* **and** *state* :: *State*
assumes
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
and
$\forall$ (*c*::*nat*). $c \in$ *set Wl* $\longrightarrow 0 \leq c \wedge c <$ *length* (*getF state*) **and**
InvariantUniqQ (*getQ state*)
shows
let state' = *notifyWatches-loop literal Wl newWl state in*
InvariantUniqQ (*getQ state'*)

using *assms*
proof (*induct Wl arbitrary*: *newWl state*)
case *Nil*
thus *?case*
by *simp*
next
case (*Cons clause Wl'*)
from ‹$\forall$ (*c*::*nat*). $c \in$ *set* (*clause* # *Wl'*) $\longrightarrow 0 \leq c \wedge c <$ *length* (*getF state*)›
have $0 \leq$ *clause* $\wedge$ *clause* $<$ *length* (*getF state*)
by *auto*
then obtain *wa*::*Literal* **and** *wb*::*Literal*
where *getWatch1 state clause* = *Some wa* **and** *getWatch2 state clause* = *Some wb*
using *Cons*
unfolding *InvariantWatchesEl-def*
by *auto*
show *?case*
proof (*cases Some literal* = *getWatch1 state clause*)
case *True*
let *?state'* = *swapWatches clause state*
let *?w1* = *wb*
have *getWatch1 ?state' clause* = *Some ?w1*
using ‹*getWatch2 state clause* = *Some wb*›
unfolding *swapWatches-def*

by *auto*
let *?w2* = *wa*
have *getWatch2 ?state′ clause* = *Some ?w2*
using ‹*getWatch1 state clause* = *Some wa*›
unfolding *swapWatches-def*
by *auto*
show *?thesis*
proof (*cases literalTrue ?w1* (*elements* (*getM ?state′*)))
case *True*

from *Cons*(*2*)
have *InvariantWatchesEl* (*getF ?state′*) (*getWatch1 ?state′*) (*getWatch2 ?state′*)
unfolding *InvariantWatchesEl-def*
unfolding *swapWatches-def*
by *auto*
moreover
have *getM ?state′* = *getM state* ∧
getF ?state′ = *getF state* ∧
getQ ?state′ = *getQ state*

unfolding *swapWatches-def*
by *simp*
ultimately
show *?thesis*
using *Cons*(*1*)[*of ?state′ clause # newWl*]
using *Cons*(*3*) *Cons*(*4*)
using ‹*getWatch1 ?state′ clause* = *Some ?w1*›
using ‹*getWatch2 ?state′ clause* = *Some ?w2*›
using ‹*Some literal* = *getWatch1 state clause*›
using ‹*literalTrue ?w1* (*elements* (*getM ?state′*))›
by (*simp add*:*Let-def*)
next
case *False*
show *?thesis*
proof (*cases getNonWatchedUnfalsifiedLiteral* (*nth* (*getF ?state′*) *clause*) *?w1 ?w2* (*getM ?state′*))
case (*Some l′*)
hence *l′ el* (*nth* (*getF ?state′*) *clause*)
using *getNonWatchedUnfalsifiedLiteralSomeCharacterization*
by *simp*

let *?state″* = *setWatch2 clause l′ ?state′*

from *Cons*(*2*)
have *InvariantWatchesEl* (*getF ?state″*) (*getWatch1 ?state″*) (*getWatch2 ?state″*)
using ‹*l′ el* (*nth* (*getF ?state′*) *clause*)›
unfolding *InvariantWatchesEl-def*

unfolding *swapWatches-def*
unfolding *setWatch2-def*
by *auto*
moreover
have *getM ?state'' = getM state ∧*
getF ?state'' = getF state ∧
getQ ?state'' = getQ state
unfolding *swapWatches-def*
unfolding *setWatch2-def*
by *simp*
ultimately
show *?thesis*
using *Cons(1)[of ?state'' newWl]*
using *Cons(3) Cons(4)*
using ‹*getWatch1 ?state' clause = Some ?w1*›
using ‹*getWatch2 ?state' clause = Some ?w2*›
using ‹*Some literal = getWatch1 state clause*›
using ‹¬ *literalTrue ?w1 (elements (getM ?state'))*›
using *Some*
by (*simp add: Let-def*)
next
case *None*
show *?thesis*
proof (*cases literalFalse ?w1 (elements (getM ?state'))*)
case *True*
let *?state'' = ?state'⦇getConflictFlag := True, getConflictClause := clause⦈*

from *Cons(2)*
have *InvariantWatchesEl (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'')*
unfolding *InvariantWatchesEl-def*
unfolding *swapWatches-def*
by *auto*
moreover
have *getM ?state'' = getM state ∧*
getF ?state'' = getF state ∧
getQ ?state'' = getQ state
unfolding *swapWatches-def*
by *simp*
ultimately
show *?thesis*
using *Cons(1)[of ?state'' clause # newWl]*
using *Cons(3) Cons(4)*
using ‹*getWatch1 ?state' clause = Some ?w1*›
using ‹*getWatch2 ?state' clause = Some ?w2*›
using ‹*Some literal = getWatch1 state clause*›
using ‹¬ *literalTrue ?w1 (elements (getM ?state'))*›
using *None*

using ‹*literalFalse ?w1* (*elements* (*getM ?state′*))›
by (*simp add*: *Let-def*)
next
case *False*
let *?state′′* = *setReason ?w1 clause* (*?state′*⦇*getQ* := (*if ?w1 el* (*getQ ?state′*) *then* (*getQ ?state′*) *else* (*getQ ?state′*) @ [*?w1*])⦈)
from *Cons*(*2*)
have *InvariantWatchesEl* (*getF ?state′′*) (*getWatch1 ?state′′*) (*getWatch2 ?state′′*)
unfolding *InvariantWatchesEl-def*
unfolding *swapWatches-def*
unfolding *setReason-def*
by *auto*
moreover
have *getM ?state′′* = *getM state*
getF ?state′′ = *getF state*
getQ ?state′′ = (*if ?w1 el* (*getQ state*) *then* (*getQ state*) *else* (*getQ state*) @ [*?w1*])
unfolding *swapWatches-def*
unfolding *setReason-def*
by *auto*
moreover
have *uniq* (*getQ ?state′′*)
using *Cons*(*4*)
using ‹*getQ ?state′′* = (*if ?w1 el* (*getQ state*) *then* (*getQ state*) *else* (*getQ state*) @ [*?w1*])›
unfolding *InvariantUniqQ-def*
by (*simp add*: *uniqAppendIff*)
ultimately
show *?thesis*
using *Cons*(*1*)[*of ?state′′ clause # newWl*]
using *Cons*(*3*)
using ‹*getWatch1 ?state′ clause* = *Some ?w1*›
using ‹*getWatch2 ?state′ clause* = *Some ?w2*›
using ‹*Some literal* = *getWatch1 state clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state′*))›
using *None*
using ‹¬ *literalFalse ?w1* (*elements* (*getM ?state′*))›
unfolding *isPrefix-def*
unfolding *InvariantUniqQ-def*
by (*simp add*: *Let-def split*: *if-split-asm*)
qed
qed
qed
next
case *False*
let *?state′* = *state*
let *?w1* = *wa*
have *getWatch1 ?state′ clause* = *Some ?w1*

using ‹*getWatch1 state clause = Some wa*›
by *auto*
let *?w2 = wb*
have *getWatch2 ?state′ clause = Some ?w2*
using ‹*getWatch2 state clause = Some wb*›
by *auto*
show *?thesis*
proof (*cases literalTrue ?w1* (*elements* (*getM ?state′*)))
case *True*
thus *?thesis*
using *Cons*
using ‹¬ *Some literal = getWatch1 state clause*›
using ‹*getWatch1 ?state′ clause = Some ?w1*›
using ‹*getWatch2 ?state′ clause = Some ?w2*›
using ‹*literalTrue ?w1* (*elements* (*getM ?state′*))›
by (*simp add*:*Let-def*)
next
case *False*
show *?thesis*
proof (*cases getNonWatchedUnfalsifiedLiteral* (*nth* (*getF ?state′*) *clause*) *?w1 ?w2* (*getM ?state′*))
case (*Some l′*)
hence *l′ el* (*nth* (*getF ?state′*)) *clause*
using *getNonWatchedUnfalsifiedLiteralSomeCharacterization*
by *simp*

let *?state′′ = setWatch2 clause l′ ?state′*

from *Cons*(*2*)
have *InvariantWatchesEl* (*getF ?state′′*) (*getWatch1 ?state′′*) (*getWatch2 ?state′′*)
using ‹*l′ el* (*nth* (*getF ?state′*)) *clause*›
unfolding *InvariantWatchesEl-def*
unfolding *setWatch2-def*
by *auto*
moreover
have *getM ?state′′ = getM state* ∧
getF ?state′′ = getF state ∧
getQ ?state′′ = getQ state
unfolding *setWatch2-def*
by *simp*
ultimately
show *?thesis*
using *Cons*(*1*)[*of ?state′′*]
using *Cons*(*3*) *Cons*(*4*)
using ‹*getWatch1 ?state′ clause = Some ?w1*›
using ‹*getWatch2 ?state′ clause = Some ?w2*›
using ‹¬ *Some literal = getWatch1 state clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state′*))›

using *Some*
by (*simp add*: *Let-def*)
next
case *None*
show *?thesis*
proof (*cases literalFalse ?w1* (*elements* (*getM ?state'*)))
case *True*
let *?state''* = *?state'*(|*getConflictFlag* := *True*, *getConflictClause* := *clause*|)

from *Cons*(*2*)
have *InvariantWatchesEl* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)
unfolding *InvariantWatchesEl-def*
by *auto*
moreover
have *getM ?state''* = *getM state* ∧
getF ?state'' = *getF state* ∧
getQ ?state'' = *getQ state*
by *simp*
ultimately
show *?thesis*
using *Cons*(*1*)[*of ?state''*]
using *Cons*(*3*) *Cons*(*4*)
using ‹*getWatch1 ?state' clause* = *Some ?w1*›
using ‹*getWatch2 ?state' clause* = *Some ?w2*›
using ‹¬ *Some literal* = *getWatch1 state clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state'*))›
using *None*
using ‹*literalFalse ?w1* (*elements* (*getM ?state'*))›
by (*simp add*: *Let-def*)
next
case *False*
let *?state''* = *setReason ?w1 clause* (*?state'*(|*getQ* := (*if ?w1 el* (*getQ ?state'*) *then* (*getQ ?state'*) *else* (*getQ ?state'*) @ [*?w1*])|))
from *Cons*(*2*)
have *InvariantWatchesEl* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)
unfolding *InvariantWatchesEl-def*
unfolding *setReason-def*
by *auto*
moreover
have *getM ?state''* = *getM state*
getF ?state'' = *getF state*
getQ ?state'' = (*if ?w1 el* (*getQ state*) *then* (*getQ state*) *else* (*getQ state*) @ [*?w1*])
unfolding *setReason-def*
by *auto*
moreover

have *uniq* (*getQ ?state''*)
using *Cons*(*4*)
using ‹*getQ ?state''* = (*if ?w1 el* (*getQ state*) *then* (*getQ state*) *else* (*getQ state*) @ [*?w1*])›
unfolding *InvariantUniqQ-def*
by (*simp add*: *uniqAppendIff*)
ultimately
show *?thesis*
using *Cons*(*1*)[*of ?state''*]
using *Cons*(*3*)
using ‹*getWatch1 ?state' clause* = *Some ?w1*›
using ‹*getWatch2 ?state' clause* = *Some ?w2*›
using ‹¬ *Some literal* = *getWatch1 state clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state'*))›
using *None*
using ‹¬ *literalFalse ?w1* (*elements* (*getM ?state'*))›
unfolding *isPrefix-def*
unfolding *InvariantUniqQ-def*
by (*simp add*: *Let-def split*: *if-split-asm*)
qed
qed
qed
qed
qed

lemma *InvariantConflictClauseCharacterizationAfterNotifyWatches*:
assumes
(*getM state*) = *M* @ [(*opposite literal*, *decision*)] **and**
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
∀ (*c*::*nat*). *c* ∈ *set Wl* ⟶ *0* ≤ *c* ∧ *c* < *length* (*getF state*) **and**
∀ (*c*::*nat*). *c* ∈ *set Wl* ⟶ *Some literal* = (*getWatch1 state c*) ∨ *Some literal* = (*getWatch2 state c*) **and**
InvariantConflictClauseCharacterization (*getConflictFlag state*) (*getConflictClause state*) (*getF state*) (*getM state*)
uniq Wl
shows
let state' = (*notifyWatches-loop literal Wl newWl state*) *in*
InvariantConflictClauseCharacterization (*getConflictFlag state'*) (*getConflictClause state'*) (*getF state'*) (*getM state'*)
using *assms*
proof (*induct Wl arbitrary*: *newWl state*)
case *Nil*
thus *?case*
by *simp*
next
case (*Cons clause Wl'*)

from ‹*uniq* (*clause* # *Wl'*)›

have *clause* ∉ *set Wl′ uniq Wl′*
by (*auto simp add*:*uniqAppendIff*)

from ‹∀ (*c*::*nat*). *c* ∈ *set* (*clause* # *Wl′*) ⟶ *0* ≤ *c* ∧ *c* < *length* (*getF state*)›
have *0* ≤ *clause* ∧ *clause* < *length* (*getF state*)
by *auto*
then obtain *wa*::*Literal* **and** *wb*::*Literal*
where *getWatch1 state clause* = *Some wa* **and** *getWatch2 state clause* = *Some wb*
using *Cons*
unfolding *InvariantWatchesEl-def*
by *auto*
show *?case*
proof (*cases Some literal* = *getWatch1 state clause*)
case *True*
let *?state′* = *swapWatches clause state*
let *?w1* = *wb*
have *getWatch1 ?state′ clause* = *Some ?w1*
using ‹*getWatch2 state clause* = *Some wb*›
unfolding *swapWatches-def*
by *auto*
let *?w2* = *wa*
have *getWatch2 ?state′ clause* = *Some ?w2*
using ‹*getWatch1 state clause* = *Some wa*›
unfolding *swapWatches-def*
by *auto*

with *True* **have**
?w2 = *literal*
unfolding *swapWatches-def*
by *simp*
hence *literalFalse ?w2* (*elements* (*getM state*))
using *Cons*(*2*)
by *simp*

show *?thesis*
proof (*cases literalTrue ?w1* (*elements* (*getM ?state′*)))
case *True*

from *Cons*(*3*)
have *InvariantWatchesEl* (*getF ?state′*) (*getWatch1 ?state′*) (*getWatch2 ?state′*)
unfolding *InvariantWatchesEl-def*
unfolding *swapWatches-def*
by *auto*
moreover
have ∀ *c*. *c* ∈ *set Wl′* ⟶ *Some literal* = *getWatch1 ?state′ c* ∨ *Some literal* = *getWatch2 ?state′ c*

using *Cons*(*5*)
unfolding *swapWatches-def*
by *auto*
moreover
have *getM ?state′ = getM state ∧*
getF ?state′ = getF state ∧
getConflictFlag ?state′ = getConflictFlag state ∧
getConflictClause ?state′ = getConflictClause state

unfolding *swapWatches-def*
by *simp*
ultimately
show *?thesis*
using *Cons*(*1*)[*of ?state′ clause # newWl*]
using *Cons*(*2*) *Cons*(*4*) *Cons*(*6*) *Cons*(*7*)
using ‹*getWatch1 ?state′ clause = Some ?w1*›
using ‹*getWatch2 ?state′ clause = Some ?w2*›
using ‹*Some literal = getWatch1 state clause*›
using ‹*literalTrue ?w1* (*elements* (*getM ?state′*))›
using ‹*uniq Wl′*›
by (*simp add*:*Let-def*)
next
case *False*
show *?thesis*
proof (*cases getNonWatchedUnfalsifiedLiteral* (*nth* (*getF ?state′*) *clause*) *?w1 ?w2* (*getM ?state′*))
case (*Some l′*)
hence *l′ el* (*nth* (*getF ?state′*) *clause*)
using *getNonWatchedUnfalsifiedLiteralSomeCharacterization*
by *simp*

let *?state′′ = setWatch2 clause l′ ?state′*

from *Cons*(*3*)
have *InvariantWatchesEl* (*getF ?state′′*) (*getWatch1 ?state′′*) (*getWatch2 ?state′′*)
using ‹*l′ el* (*nth* (*getF ?state′*) *clause*)›
unfolding *InvariantWatchesEl-def*
unfolding *swapWatches-def*
unfolding *setWatch2-def*
by *auto*
moreover
have ∀ (*c*::*nat*). *c ∈ set Wl′ ⟶ Some literal =* (*getWatch1 ?state′′ c*) ∨ *Some literal =* (*getWatch2 ?state′′ c*)
using *Cons*(*5*)
using ‹*clause ∉ set Wl′*›
using *swapWatchesEffect*[*of clause state*]
unfolding *setWatch2-def*
by *simp*

moreover
have *getM ?state''* = *getM state* ∧
getF ?state'' = *getF state* ∧
getConflictFlag ?state'' = *getConflictFlag state* ∧
getConflictClause ?state'' = *getConflictClause state*
unfolding *swapWatches-def*
unfolding *setWatch2-def*
by *simp*
ultimately
show *?thesis*
using *Cons*(*1*)[*of ?state'' newWl*]
using *Cons*(*2*) *Cons*(*4*) *Cons*(*6*) *Cons*(*7*)
using ‹*getWatch1 ?state' clause* = *Some ?w1*›
using ‹*getWatch2 ?state' clause* = *Some ?w2*›
using ‹*Some literal* = *getWatch1 state clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state'*))›
using *Some*
using ‹*uniq Wl'*›
by (*simp add*: *Let-def*)
next
case *None*
show *?thesis*
proof (*cases literalFalse ?w1* (*elements* (*getM ?state'*)))
case *True*
let *?state''* = *?state'*⦇*getConflictFlag* := *True*, *getConflictClause* := *clause*⦈

from *Cons*(*3*)
have *InvariantWatchesEl* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)
unfolding *InvariantWatchesEl-def*
unfolding *swapWatches-def*
by *auto*
moreover
have *getM ?state''* = *getM state* ∧
getF ?state'' = *getF state* ∧
getConflictFlag ?state'' ∧
getConflictClause ?state'' = *clause*
unfolding *swapWatches-def*
by *simp*
moreover
have ∀ (*c*::*nat*). *c* ∈ *set Wl'* ⟶ *Some literal* = (*getWatch1 ?state'' c*) ∨ *Some literal* = (*getWatch2 ?state'' c*)
using *Cons*(*5*)
using ‹*clause* ∉ *set Wl'*›
using *swapWatchesEffect*[*of clause state*]
by *simp*
moreover
have ∀ *l*. *l el* (*nth* (*getF ?state''*) *clause*) ∧ *l* ≠ *?w1* ∧ *l* ≠

?w2 ⟶ *literalFalse l* (*elements* (*getM ?state''*))
    using *None*
    using ‹*getWatch1 ?state' clause = Some ?w1*›
    using ‹*getWatch2 ?state' clause = Some ?w2*›
    using *getNonWatchedUnfalsifiedLiteralNoneCharacterization*[*of nth* (*getF ?state'*) *clause ?w1 ?w2 getM ?state'*]
    unfolding *setReason-def*
    unfolding *swapWatches-def*
    by *auto*

  hence *clauseFalse* (*nth* (*getF state*) *clause*) (*elements* (*getM state*))
    using ‹*literalFalse ?w1* (*elements* (*getM ?state'*))›
    using ‹*literalFalse ?w2* (*elements* (*getM state*))›
    unfolding *swapWatches-def*
    by (*auto simp add*: *clauseFalseIffAllLiteralsAreFalse*)
  moreover
  have (*nth* (*getF state*) *clause*) *el* (*getF state*)
    using ‹*0* ≤ *clause* ∧ *clause* < *length* (*getF state*)›
    using *nth-mem*[*of clause getF state*]
    by *simp*
  ultimately
  show *?thesis*
    using *Cons*(*1*)[*of ?state'' clause # newWl*]
    using *Cons*(*2*) *Cons*(*4*) *Cons*(*6*) *Cons*(*7*)
    using ‹*getWatch1 ?state' clause = Some ?w1*›
    using ‹*getWatch2 ?state' clause = Some ?w2*›
    using ‹*Some literal = getWatch1 state clause*›
    using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state'*))›
    using *None*
    using ‹*literalFalse ?w1* (*elements* (*getM ?state'*))›
    using ‹*uniq Wl'*›
    using ‹*0* ≤ *clause* ∧ *clause* < *length* (*getF state*)›
    unfolding *InvariantConflictClauseCharacterization-def*
    by (*simp add*: *Let-def*)
next
  case *False*
  let *?state''* = *setReason ?w1 clause* (*?state'*⦇*getQ* := (*if ?w1 el* (*getQ ?state'*) *then* (*getQ ?state'*) *else* (*getQ ?state'*) @ [*?w1*])⦈)
  from *Cons*(*3*)
  have *InvariantWatchesEl* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)
    unfolding *InvariantWatchesEl-def*
    unfolding *swapWatches-def*
    unfolding *setReason-def*
    by *auto*
  moreover
  have *getM ?state''* = *getM state*
    getF ?state'' = *getF state*

```
          getConflictFlag ?state'' = getConflictFlag state
          getConflictClause ?state'' = getConflictClause state
          unfolding swapWatches-def
          unfolding setReason-def
          by auto
        moreover
        have ∀ (c::nat). c ∈ set Wl' ⟶ Some literal = (getWatch1
?state'' c) ∨ Some literal = (getWatch2 ?state'' c)
          using Cons(5)
          using ‹clause ∉ set Wl'›
          using swapWatchesEffect[of clause state]
          unfolding setReason-def
          by simp
        ultimately
        show ?thesis
          using Cons(1)[of ?state'' clause # newWl]
          using Cons(2) Cons(4) Cons(6) Cons(7)
          using ‹getWatch1 ?state' clause = Some ?w1›
          using ‹getWatch2 ?state' clause = Some ?w2›
          using ‹Some literal = getWatch1 state clause›
          using ‹¬ literalTrue ?w1 (elements (getM ?state'))›
          using None
          using ‹¬ literalFalse ?w1 (elements (getM ?state'))›
          using ‹uniq Wl'›
          by (simp add: Let-def)
      qed
    qed
  qed
next
  case False
  let ?state' = state
  let ?w1 = wa
  have getWatch1 ?state' clause = Some ?w1
    using ‹getWatch1 state clause = Some wa›
    by auto
  let ?w2 = wb
  have getWatch2 ?state' clause = Some ?w2
    using ‹getWatch2 state clause = Some wb›
    by auto

  from ‹¬ Some literal = getWatch1 state clause›
    ‹∀ (c::nat). c ∈ set (clause # Wl') ⟶ Some literal = (getWatch1
state c) ∨ Some literal = (getWatch2 state c)›
  have Some literal = getWatch2 state clause
    by auto
  hence ?w2 = literal
    using ‹getWatch2 ?state' clause = Some ?w2›
    by simp
  hence literalFalse ?w2 (elements (getM state))
```

using *Cons*(*2*)
by *simp*

show *?thesis*
proof (*cases literalTrue ?w1* (*elements* (*getM ?state′*)))
case *True*
thus *?thesis*
using *Cons*(*1*)[*of ?state′ clause # newWl*]
using *Cons*(*2*) *Cons*(*3*) *Cons*(*4*) *Cons*(*5*) *Cons*(*6*) *Cons*(*7*)
using ‹¬ *Some literal* = *getWatch1 state clause*›
using ‹*getWatch1 ?state′ clause* = *Some ?w1*›
using ‹*getWatch2 ?state′ clause* = *Some ?w2*›
using ‹*literalTrue ?w1* (*elements* (*getM ?state′*))›
using ‹*uniq Wl′*›
by (*simp add*:*Let-def*)
next
case *False*
show *?thesis*
proof (*cases getNonWatchedUnfalsifiedLiteral* (*nth* (*getF ?state′*) *clause*) *?w1 ?w2* (*getM ?state′*))
case (*Some l′*)
hence *l′ el* (*nth* (*getF ?state′*)) *clause*
using *getNonWatchedUnfalsifiedLiteralSomeCharacterization*
by *simp*

let *?state′′* = *setWatch2 clause l′ ?state′*

from *Cons*(*3*)
have *InvariantWatchesEl* (*getF ?state′′*) (*getWatch1 ?state′′*) (*getWatch2 ?state′′*)
using ‹*l′ el* (*nth* (*getF ?state′*)) *clause*›
unfolding *InvariantWatchesEl-def*
unfolding *setWatch2-def*
by *auto*
moreover
have *getM ?state′′* = *getM state* ∧
getF ?state′′ = *getF state* ∧
getQ ?state′′ = *getQ state* ∧
getConflictFlag ?state′′ = *getConflictFlag state* ∧
getConflictClause ?state′′ = *getConflictClause state*
unfolding *setWatch2-def*
by *simp*
moreover
have ∀ (*c*::*nat*). *c* ∈ *set Wl′* ⟶ *Some literal* = (*getWatch1 ?state′′ c*) ∨ *Some literal* = (*getWatch2 ?state′′ c*)
using *Cons*(*5*)
using ‹*clause* ∉ *set Wl′*›
unfolding *setWatch2-def*
by *simp*

ultimately
show *?thesis*
using *Cons*(*1*)[*of ?state'' newWl*]
using *Cons*(*2*) *Cons*(*4*) *Cons*(*6*) *Cons*(*7*)
using ‹*getWatch1 ?state' clause* = *Some ?w1*›
using ‹*getWatch2 ?state' clause* = *Some ?w2*›
using ‹¬ *Some literal* = *getWatch1 state clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state'*))›
using *Some*
using ‹*uniq Wl'*›
by (*simp add*: *Let-def*)
next
case *None*
show *?thesis*
proof (*cases literalFalse ?w1* (*elements* (*getM ?state'*)))
case *True*
let *?state''* = *?state'*⦇*getConflictFlag* := *True*, *getConflictClause* := *clause*⦈

from *Cons*(*3*)
have *InvariantWatchesEl* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)
unfolding *InvariantWatchesEl-def*
by *auto*
moreover
have *getM ?state''* = *getM state* ∧
getF ?state'' = *getF state* ∧
getQ ?state'' = *getQ state* ∧
getConflictFlag ?state'' ∧
getConflictClause ?state'' = *clause*
by *simp*
moreover
have ∀ (*c*::*nat*). *c* ∈ *set Wl'* ⟶ *Some literal* = (*getWatch1 ?state'' c*) ∨ *Some literal* = (*getWatch2 ?state'' c*)
using *Cons*(*5*)
using ‹*clause* ∉ *set Wl'*›
by *simp*
moreover
have ∀ *l*. *l el* (*nth* (*getF ?state''*) *clause*) ∧ *l* ≠ *?w1* ∧ *l* ≠ *?w2* ⟶ *literalFalse l* (*elements* (*getM ?state''*))
using *None*
using ‹*getWatch1 ?state' clause* = *Some ?w1*›
using ‹*getWatch2 ?state' clause* = *Some ?w2*›
using *getNonWatchedUnfalsifiedLiteralNoneCharacterization*[*of nth* (*getF ?state'*) *clause ?w1 ?w2 getM ?state'*]
unfolding *setReason-def*
by *auto*
hence *clauseFalse* (*nth* (*getF state*) *clause*) (*elements* (*getM state*))

using ‹*literalFalse ?w1* (*elements* (*getM ?state′*))›
using ‹*literalFalse ?w2* (*elements* (*getM state*))›
by (*auto simp add*: *clauseFalseIffAllLiteralsAreFalse*)
moreover
have (*nth* (*getF state*) *clause*) *el* (*getF state*)
using ‹*0* ≤ *clause* ∧ *clause* < *length* (*getF state*)›
using *nth-mem*[*of clause getF state*]
by *simp*
ultimately
show *?thesis*
using *Cons*(*1*)[*of ?state′′*]
using *Cons*(*2*) *Cons*(*4*) *Cons*(*6*) *Cons*(*7*)
using ‹*getWatch1 ?state′ clause* = *Some ?w1*›
using ‹*getWatch2 ?state′ clause* = *Some ?w2*›
using ‹¬ *Some literal* = *getWatch1 state clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state′*))›
using *None*
using ‹*literalFalse ?w1* (*elements* (*getM ?state′*))›
using ‹*uniq Wl′*›
using ‹*0* ≤ *clause* ∧ *clause* < *length* (*getF state*)›
unfolding *InvariantConflictClauseCharacterization-def*
by (*simp add*: *Let-def*)
next
case *False*
let *?state′′* = *setReason ?w1 clause* (*?state′*(|*getQ* := (*if ?w1 el* (*getQ ?state′*) *then* (*getQ ?state′*) *else* (*getQ ?state′*) @ [*?w1*])|))
from *Cons*(*3*)
have *InvariantWatchesEl* (*getF ?state′′*) (*getWatch1 ?state′′*) (*getWatch2 ?state′′*)
unfolding *InvariantWatchesEl-def*
unfolding *setReason-def*
by *auto*
moreover
have *getM ?state′′* = *getM state*
getF ?state′′ = *getF state*
getConflictFlag ?state′′ = *getConflictFlag state*
getConflictClause ?state′′ = *getConflictClause state*
unfolding *setReason-def*
by *auto*
moreover
have ∀ (*c*::*nat*). *c* ∈ *set Wl′* ⟶ *Some literal* = (*getWatch1 ?state′′ c*) ∨ *Some literal* = (*getWatch2 ?state′′ c*)
using *Cons*(*5*)
using ‹*clause* ∉ *set Wl′*›
unfolding *setReason-def*
by *simp*
ultimately
show *?thesis*
using *Cons*(*1*)[*of ?state′′*]

```
          using Cons(2) Cons(4) Cons(6) Cons(7)
          using ‹getWatch1 ?state' clause = Some ?w1›
          using ‹getWatch2 ?state' clause = Some ?w2›
          using ‹¬ Some literal = getWatch1 state clause›
          using ‹¬ literalTrue ?w1 (elements (getM ?state'))›
          using None
          using ‹¬ literalFalse ?w1 (elements (getM ?state'))›
          using ‹uniq Wl'›
          by (simp add: Let-def)
        qed
      qed
    qed
  qed
qed

lemma InvariantGetReasonIsReasonQSubset:
  assumes Q ⊆ Q' and
  InvariantGetReasonIsReason GetReason F M Q'
  shows
  InvariantGetReasonIsReason GetReason F M Q
using assms
unfolding InvariantGetReasonIsReason-def
by auto

lemma InvariantGetReasonIsReasonAfterNotifyWatches:
assumes
  InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
and
  ∀ (c::nat). c ∈ set Wl ⟶ 0 ≤ c ∧ c < length (getF state) and
  ∀ (c::nat). c ∈ set Wl ⟶ Some literal = (getWatch1 state c) ∨
Some literal = (getWatch2 state c) and
  uniq Wl
  getM state = M @ [(opposite literal, decision)]
  InvariantGetReasonIsReason (getReason state) (getF state) (getM
state) Q
shows
  let state' = notifyWatches-loop literal Wl newWl state in
   let Q' = Q ∪ (set (getQ state') − set (getQ state)) in
    InvariantGetReasonIsReason (getReason state') (getF state') (getM
state') Q'
using assms
proof (induct Wl arbitrary: newWl state Q)
  case Nil
  thus ?case
    by simp
next
  case (Cons clause Wl')

  from ‹uniq (clause # Wl')›
```

have *clause* ∉ *set Wl′ uniq Wl′*
by (*auto simp add*:*uniqAppendIff*)

from ‹∀ (*c*::*nat*). *c* ∈ *set* (*clause* # *Wl′*) ⟶ *0* ≤ *c* ∧ *c* < *length* (*getF state*)›
have *0* ≤ *clause* ∧ *clause* < *length* (*getF state*)
by *auto*
then obtain *wa*::*Literal* **and** *wb*::*Literal*
where *getWatch1 state clause* = *Some wa* **and** *getWatch2 state clause* = *Some wb*
using *Cons*
unfolding *InvariantWatchesEl-def*
by *auto*
show *?case*
proof (*cases Some literal* = *getWatch1 state clause*)
case *True*
let *?state′* = *swapWatches clause state*
let *?w1* = *wb*
have *getWatch1 ?state′ clause* = *Some ?w1*
using ‹*getWatch2 state clause* = *Some wb*›
unfolding *swapWatches-def*
by *auto*
let *?w2* = *wa*
have *getWatch2 ?state′ clause* = *Some ?w2*
using ‹*getWatch1 state clause* = *Some wa*›
unfolding *swapWatches-def*
by *auto*
with *True* **have**
?w2 = *literal*
unfolding *swapWatches-def*
by *simp*
hence *literalFalse ?w2* (*elements* (*getM state*))
using *Cons*(*6*)
by *simp*

from ‹*InvariantWatchesEl* (*getF state*) (*getWatch1 state*) (*getWatch2 state*)›
have *?w1 el* (*nth* (*getF state*) *clause*) *?w2 el* (*nth* (*getF state*) *clause*)
using ‹*getWatch1 ?state′ clause* = *Some ?w1*›
using ‹*getWatch2 ?state′ clause* = *Some ?w2*›
using ‹*0* ≤ *clause* ∧ *clause* < *length* (*getF state*)›
unfolding *InvariantWatchesEl-def*
unfolding *swapWatches-def*
by *auto*

show *?thesis*
proof (*cases literalTrue ?w1* (*elements* (*getM ?state′*)))
case *True*

from *Cons*(*2*)
have *InvariantWatchesEl* (*getF ?state'*) (*getWatch1 ?state'*) (*getWatch2 ?state'*)
unfolding *InvariantWatchesEl-def*
unfolding *swapWatches-def*
by *auto*
moreover
have $\forall$ *c. c* $\in$ *set Wl'* $\longrightarrow$ *Some literal* = *getWatch1 ?state' c* $\vee$ *Some literal* = *getWatch2 ?state' c*
using *Cons*(*4*)
unfolding *swapWatches-def*
by *auto*
moreover
have *getM ?state'* = *getM state* $\wedge$
getF ?state' = *getF state* $\wedge$
getQ ?state' = *getQ state* $\wedge$
getReason ?state' = *getReason state*

unfolding *swapWatches-def*
by *simp*
ultimately
show *?thesis*
using *Cons*(*1*)[*of ?state' Q clause # newWl*]
using *Cons*(*3*) *Cons*(*6*) *Cons*(*7*)
using ‹*getWatch1 ?state' clause* = *Some ?w1*›
using ‹*getWatch2 ?state' clause* = *Some ?w2*›
using ‹*Some literal* = *getWatch1 state clause*›
using ‹*literalTrue ?w1* (*elements* (*getM ?state'*))›
using ‹*uniq Wl'*›
by (*simp add*:*Let-def*)
next
case *False*
show *?thesis*
proof (*cases getNonWatchedUnfalsifiedLiteral* (*nth* (*getF ?state'*) *clause*) *?w1 ?w2* (*getM ?state'*))
case (*Some l'*)
hence *l' el* (*nth* (*getF ?state'*) *clause*)
using *getNonWatchedUnfalsifiedLiteralSomeCharacterization*
by *simp*

let *?state''* = *setWatch2 clause l' ?state'*

from *Cons*(*2*)
have *InvariantWatchesEl* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)
using ‹*l' el* (*nth* (*getF ?state'*) *clause*)›
unfolding *InvariantWatchesEl-def*
unfolding *swapWatches-def*

unfolding *setWatch2-def*
by *auto*
moreover
have ∀ (*c*::*nat*). *c* ∈ *set Wl'* ⟶ *Some literal* = (*getWatch1 ?state'' c*) ∨ *Some literal* = (*getWatch2 ?state'' c*)
using *Cons*(*4*)
using ‹*clause* ∉ *set Wl'*›
using *swapWatchesEffect*[*of clause state*]
unfolding *setWatch2-def*
by *simp*
moreover
have *getM ?state''* = *getM state* ∧
getF ?state'' = *getF state* ∧
getQ ?state'' = *getQ state* ∧
getReason ?state'' = *getReason state*
unfolding *swapWatches-def*
unfolding *setWatch2-def*
by *simp*
ultimately
show *?thesis*
using *Cons*(*1*)[*of ?state'' Q newWl*]
using *Cons*(*3*) *Cons*(*6*) *Cons*(*7*)
using ‹*getWatch1 ?state' clause* = *Some ?w1*›
using ‹*getWatch2 ?state' clause* = *Some ?w2*›
using ‹*Some literal* = *getWatch1 state clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state'*))›
using *Some*
using ‹*uniq Wl'*›
by (*simp add*: *Let-def*)
next
case *None*
hence ∀ *l*. *l el* (*nth* (*getF ?state'*) *clause*) ∧ *l* ≠ *?w1* ∧ *l* ≠ *?w2* ⟶ *literalFalse l* (*elements* (*getM ?state'*))
using *getNonWatchedUnfalsifiedLiteralNoneCharacterization*
by *simp*
show *?thesis*
proof (*cases literalFalse ?w1* (*elements* (*getM ?state'*)))
case *True*
let *?state''* = *?state'*⦇*getConflictFlag* := *True*, *getConflictClause* := *clause*⦈

from *Cons*(*2*)
have *InvariantWatchesEl* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)
unfolding *InvariantWatchesEl-def*
unfolding *swapWatches-def*
by *auto*
moreover
have ∀ *c*. *c* ∈ *set Wl'* ⟶ *Some literal* = *getWatch1 ?state''*

c ∨ Some literal = getWatch2 ?state'' c
using Cons(4)
unfolding swapWatches-def
by auto
moreover
have getM ?state'' = getM state ∧
getF ?state'' = getF state ∧
getQ ?state'' = getQ state ∧
getReason ?state'' = getReason state
unfolding swapWatches-def
by simp
ultimately
show ?thesis
using Cons(1)[of ?state'' Qclause # newWl]
using Cons(3) Cons(6) Cons(7)
using ‹getWatch1 ?state' clause = Some ?w1›
using ‹getWatch2 ?state' clause = Some ?w2›
using ‹Some literal = getWatch1 state clause›
using ‹¬ literalTrue ?w1 (elements (getM ?state'))›
using None
using ‹literalFalse ?w1 (elements (getM ?state'))›
using ‹uniq Wl'›
by (simp add: Let-def)
next
case False
let ?state'' = setReason ?w1 clause (?state'⦇getQ := (if ?w1 el (getQ ?state') then (getQ ?state') else (getQ ?state') @ [?w1])⦈)
let ?state0 = notifyWatches-loop literal Wl' (clause # newWl) ?state''

from Cons(2)
have InvariantWatchesEl (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'')
unfolding InvariantWatchesEl-def
unfolding swapWatches-def
unfolding setReason-def
by auto
moreover
have getM ?state'' = getM state
getF ?state'' = getF state
getQ ?state'' = (if ?w1 el (getQ state) then (getQ state) else (getQ state) @ [?w1])
getReason ?state'' = (getReason state)(?w1 := Some clause)
unfolding swapWatches-def
unfolding setReason-def
by auto
moreover
hence ∀ (c::nat). c ∈ set Wl' ⟶ Some literal = (getWatch1

?state'' c) ∨ Some literal = (getWatch2 ?state'' c)
using *Cons(4)*
using ‹*clause* $\notin$ *set Wl'*›
using *swapWatchesEffect[of clause state]*
unfolding *setReason-def*
by *simp*
moreover
have *isUnitClause (nth (getF state) clause) ?w1 (elements (getM state))*
using ‹∀ *l. l el (nth (getF ?state') clause)* ∧ *l* ≠ *?w1* ∧ *l* ≠ *?w2* ⟶ *literalFalse l (elements (getM ?state'))*›
using ‹*?w1 el (nth (getF state) clause)*›
using ‹*?w2 el (nth (getF state) clause)*›
using ‹¬ *literalTrue ?w1 (elements (getM ?state'))*›
using ‹¬ *literalFalse ?w1 (elements (getM ?state'))*›
using ‹*literalFalse ?w2 (elements (getM state))*›
unfolding *swapWatches-def*
unfolding *isUnitClause-def*
by *auto*
hence *InvariantGetReasonIsReason (getReason ?state'') (getF ?state'') (getM ?state'') (Q* ∪ *{?w1})*
using *Cons(7)*
using ‹*getM ?state'' = getM state*›
using ‹*getF ?state'' = getF state*›
using ‹*getQ ?state'' = (if ?w1 el (getQ state) then (getQ state) else (getQ state) @ [?w1])*›
using ‹*getReason ?state'' = (getReason state)(?w1 := Some clause)*›
using ‹*0* ≤ *clause* ∧ *clause < length (getF state)*›
using ‹¬ *literalTrue ?w1 (elements (getM ?state'))*›
using ‹*isUnitClause (nth (getF state) clause) ?w1 (elements (getM state))*›
unfolding *swapWatches-def*
unfolding *InvariantGetReasonIsReason-def*
by *auto*
moreover
have *(λa. if a = ?w1 then Some clause else getReason state a) = getReason ?state''*
unfolding *setReason-def*
unfolding *swapWatches-def*
by *(auto simp add: fun-upd-def)*
ultimately
have *InvariantGetReasonIsReason (getReason ?state0) (getF ?state0) (getM ?state0)*
(Q ∪ *(set (getQ ?state0)* − *set (getQ ?state''))* ∪ *{?w1})*
using *Cons(1)[of ?state'' Q* ∪ *{?w1} clause # newWl]*
using *Cons(3) Cons(6) Cons(7)*
using ‹*uniq Wl'*›
by *(simp add: Let-def split: if-split-asm)*

moreover
have $(Q \cup (set\ (getQ\ ?state0) - set\ (getQ\ state))) \subseteq (Q \cup (set\ (getQ\ ?state0) - set\ (getQ\ ?state'')) \cup \{?w1\})$
using ‹$getQ\ ?state'' = (if\ ?w1\ el\ (getQ\ state)\ then\ (getQ\ state)\ else\ (getQ\ state)\ @\ [?w1])$›
unfolding *swapWatches-def*
by *auto*
ultimately
have *InvariantGetReasonIsReason* (*getReason ?state0*) (*getF ?state0*) (*getM ?state0*)
$(Q \cup (set\ (getQ\ ?state0) - set\ (getQ\ state)))$
using *InvariantGetReasonIsReasonQSubset*[*of* $Q \cup (set\ (getQ\ ?state0) - set\ (getQ\ state))$
$Q \cup (set\ (getQ\ ?state0) - set\ (getQ\ ?state'')) \cup \{?w1\}$ *getReason ?state0 getF ?state0 getM ?state0*]
by *simp*
moreover
have *notifyWatches-loop literal* (*clause # Wl'*) *newWl state* = *?state0*
using ‹*getWatch1 ?state' clause = Some ?w1*›
using ‹*getWatch2 ?state' clause = Some ?w2*›
using ‹*Some literal = getWatch1 state clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state'*))›
using *None*
using ‹¬ *literalFalse ?w1* (*elements* (*getM ?state'*))›
using ‹*uniq Wl'*›
by (*simp add*: *Let-def*)
ultimately
show *?thesis*
by *simp*
qed
qed
qed
next
case *False*
let *?state' = state*
let *?w1 = wa*
have *getWatch1 ?state' clause = Some ?w1*
using ‹*getWatch1 state clause = Some wa*›
by *auto*
let *?w2 = wb*
have *getWatch2 ?state' clause = Some ?w2*
using ‹*getWatch2 state clause = Some wb*›
by *auto*

have *?w2 = literal*
using ‹$0 \leq clause \wedge clause < length\ (getF\ state)$›
using ‹*getWatch1 ?state' clause = Some ?w1*›
using ‹*getWatch2 ?state' clause = Some ?w2*›

```
      using Cons(4)
      using False
      by simp

    hence literalFalse ?w2 (elements (getM state))
      using Cons(6)
      by simp

  from ‹InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2
state)›
    have ?w1 el (nth (getF state) clause) ?w2 el (nth (getF state)
clause)
      using ‹getWatch1 ?state' clause = Some ?w1›
      using ‹getWatch2 ?state' clause = Some ?w2›
      using ‹0 ≤ clause ∧ clause < length (getF state)›
      unfolding InvariantWatchesEl-def
      unfolding swapWatches-def
      by auto

    show ?thesis
    proof (cases literalTrue ?w1 (elements (getM ?state')))
      case True
      thus ?thesis
        using Cons(1)[of state Q clause # newWl]
        using Cons(2) Cons(3) Cons(4) Cons(5) Cons(6) Cons(7)
        using ‹¬ Some literal = getWatch1 state clause›
        using ‹getWatch1 ?state' clause = Some ?w1›
        using ‹getWatch2 ?state' clause = Some ?w2›
        using ‹literalTrue ?w1 (elements (getM ?state'))›
        using ‹uniq Wl'›
        by (simp add:Let-def)
    next
      case False
      show ?thesis
      proof (cases getNonWatchedUnfalsifiedLiteral (nth (getF ?state')
clause) ?w1 ?w2 (getM ?state'))
        case (Some l')
        hence l' el (nth (getF ?state')) clause
          using getNonWatchedUnfalsifiedLiteralSomeCharacterization
          by simp

        let ?state'' = setWatch2 clause l' ?state'

        from Cons(2)
        have InvariantWatchesEl (getF ?state'') (getWatch1 ?state'')
(getWatch2 ?state'')
          using ‹l' el (nth (getF ?state')) clause›
          unfolding InvariantWatchesEl-def
          unfolding setWatch2-def
```

by *auto*
moreover
have ∀ *c*. *c* ∈ *set Wl'* ⟶ *Some literal* = *getWatch1 ?state'' c* ∨ *Some literal* = *getWatch2 ?state'' c*
using *Cons*(*4*)
using ‹*clause* ∉ *set Wl'*›
unfolding *setWatch2-def*
by *simp*
moreover
have *getM ?state''* = *getM state* ∧
getF ?state'' = *getF state* ∧
getQ ?state'' = *getQ state* ∧
getReason ?state'' = *getReason state*
unfolding *setWatch2-def*
by *simp*
ultimately
show *?thesis*
using *Cons*(*1*)[*of ?state''*]
using *Cons*(*3*) *Cons*(*6*) *Cons*(*7*)
using ‹*getWatch1 ?state' clause* = *Some ?w1*›
using ‹*getWatch2 ?state' clause* = *Some ?w2*›
using ‹¬ *Some literal* = *getWatch1 state clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state'*))›
using ‹*uniq Wl'*›
using *Some*
by (*simp add*: *Let-def*)
next
case *None*
hence ∀ *l*. *l el* (*nth* (*getF ?state'*) *clause*) ∧ *l* ≠ *?w1* ∧ *l* ≠ *?w2* ⟶ *literalFalse l* (*elements* (*getM ?state'*))
using *getNonWatchedUnfalsifiedLiteralNoneCharacterization*
by *simp*

show *?thesis*
proof (*cases literalFalse ?w1* (*elements* (*getM ?state'*)))
case *True*
let *?state''* = *?state'*⦇*getConflictFlag* := *True*, *getConflictClause* := *clause*⦈

from *Cons*(*2*)
have *InvariantWatchesEl* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)
unfolding *InvariantWatchesEl-def*
by *auto*
moreover
have ∀ *c*. *c* ∈ *set Wl'* ⟶ *Some literal* = *getWatch1 ?state'' c* ∨ *Some literal* = *getWatch2 ?state'' c*
using *Cons*(*4*)
using ‹*clause* ∉ *set Wl'*›

unfolding *setWatch2-def*
by *simp*
moreover
have *getM ?state'' = getM state ∧*
getF ?state'' = getF state ∧
getQ ?state'' = getQ state ∧
getReason ?state'' = getReason state
by *simp*
ultimately
show *?thesis*
using *Cons(1)[of ?state'']*
using *Cons(3) Cons(6) Cons(7)*
using ‹*getWatch1 ?state' clause = Some ?w1*›
using ‹*getWatch2 ?state' clause = Some ?w2*›
using ‹¬ *Some literal = getWatch1 state clause*›
using ‹¬ *literalTrue ?w1 (elements (getM ?state'))*›
using *None*
using ‹*literalFalse ?w1 (elements (getM ?state'))*›
using ‹*uniq Wl'*›
by (*simp add*: *Let-def*)
next
case *False*
let *?state'' = setReason ?w1 clause (?state'⦇getQ := (if ?w1 el (getQ ?state') then (getQ ?state') else (getQ ?state') @ [?w1])⦈)*
let *?state0 = notifyWatches-loop literal Wl' (clause # newWl) ?state''*

from *Cons(2)*
have *InvariantWatchesEl (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'')*
unfolding *InvariantWatchesEl-def*
unfolding *setReason-def*
by *auto*
moreover
have *getM ?state'' = getM state*
getF ?state'' = getF state
getQ ?state'' = (if ?w1 el (getQ state) then (getQ state) else (getQ state) @ [?w1])
getReason ?state'' = (getReason state)(?w1 := Some clause)
unfolding *setReason-def*
by *auto*
moreover
hence ∀ (*c::nat*). *c ∈ set Wl' ⟶ Some literal = (getWatch1 ?state'' c) ∨ Some literal = (getWatch2 ?state'' c)*
using *Cons(4)*
using ‹*clause ∉ set Wl'*›
unfolding *setReason-def*
by *simp*

moreover
have *isUnitClause* (*nth* (*getF state*) *clause*) *?w1* (*elements* (*getM state*))
using ‹∀ *l*. *l el* (*nth* (*getF ?state′*) *clause*) ∧ *l* ≠ *?w1* ∧ *l* ≠ *?w2* ⟶ *literalFalse l* (*elements* (*getM ?state′*))›
using ‹*?w1 el* (*nth* (*getF state*) *clause*)›
using ‹*?w2 el* (*nth* (*getF state*) *clause*)›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state′*))›
using ‹¬ *literalFalse ?w1* (*elements* (*getM ?state′*))›
using ‹*literalFalse ?w2* (*elements* (*getM state*))›
unfolding *isUnitClause-def*
by *auto*
hence *InvariantGetReasonIsReason* (*getReason ?state″*) (*getF ?state″*) (*getM ?state″*) (*Q* ∪ {*?w1*})
using *Cons*(*7*)
using ‹*getM ?state″* = *getM state*›
using ‹*getF ?state″* = *getF state*›
using ‹*getQ ?state″* = (*if ?w1 el* (*getQ state*) *then* (*getQ state*) *else* (*getQ state*) @ [*?w1*])›
using ‹*getReason ?state″* = (*getReason state*)(*?w1* := *Some clause*)›
using ‹*0* ≤ *clause* ∧ *clause* < *length* (*getF state*)›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state′*))›
using ‹*isUnitClause* (*nth* (*getF state*) *clause*) *?w1* (*elements* (*getM state*))›
unfolding *InvariantGetReasonIsReason-def*
by *auto*
moreover
have (λ*a*. *if a* = *?w1 then Some clause else getReason state a*) = *getReason ?state″*
unfolding *setReason-def*
by (*auto simp add*: *fun-upd-def*)
ultimately
have *InvariantGetReasonIsReason* (*getReason ?state0*) (*getF ?state0*) (*getM ?state0*)
(*Q* ∪ (*set* (*getQ ?state0*) − *set* (*getQ ?state″*)) ∪ {*?w1*})
using *Cons*(*1*)[*of ?state″ Q* ∪ {*?w1*} *clause* # *newWl*]
using *Cons*(*3*) *Cons*(*6*) *Cons*(*7*)
using ‹*uniq Wl′*›
by (*simp add*: *Let-def split*: *if-split-asm*)
moreover
have (*Q* ∪ (*set* (*getQ ?state0*) − *set* (*getQ state*))) ⊆ (*Q* ∪ (*set* (*getQ ?state0*) − *set* (*getQ ?state″*)) ∪ {*?w1*})
using ‹*getQ ?state″* = (*if ?w1 el* (*getQ state*) *then* (*getQ state*) *else* (*getQ state*) @ [*?w1*])›
by *auto*
ultimately
have *InvariantGetReasonIsReason* (*getReason ?state0*) (*getF ?state0*) (*getM ?state0*)

(*Q* ∪ (*set* (*getQ ?state0*) − *set* (*getQ state*)))
using *InvariantGetReasonIsReasonQSubset*[*of Q* ∪ (*set* (*getQ ?state0*) − *set* (*getQ state*))
Q ∪ (*set* (*getQ ?state0*) − *set* (*getQ ?state''*)) ∪ {*?w1*} *getReason ?state0 getF ?state0 getM ?state0*]
by *simp*
moreover
have *notifyWatches-loop literal* (*clause* # *Wl'*) *newWl state* = *?state0*
using ‹*getWatch1 ?state' clause* = *Some ?w1*›
using ‹*getWatch2 ?state' clause* = *Some ?w2*›
using ‹¬ *Some literal* = *getWatch1 state clause*›
using ‹¬ *literalTrue ?w1* (*elements* (*getM ?state'*))›
using *None*
using ‹¬ *literalFalse ?w1* (*elements* (*getM ?state'*))›
using ‹*uniq Wl'*›
by (*simp add*: *Let-def*)
ultimately
show *?thesis*
by *simp*
qed
qed
qed
qed
qed

lemma *assertLiteralEffect*:
fixes *state*::*State* **and** *l*::*Literal* **and** *d*::*bool*
assumes
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
shows
(*getM* (*assertLiteral l d state*)) = (*getM state*) @ [(*l*, *d*)] **and**
(*getF* (*assertLiteral l d state*)) = (*getF state*) **and**
(*getSATFlag* (*assertLiteral l d state*)) = (*getSATFlag state*) **and**
isPrefix (*getQ state*) (*getQ* (*assertLiteral l d state*))
using *assms*
unfolding *assertLiteral-def*
unfolding *notifyWatches-def*
unfolding *InvariantWatchListsContainOnlyClausesFromF-def*
using *notifyWatchesLoopPreservedVariables*[*of* (*state*⦇*getM* := *getM state* @ [(*l*, *d*)]⦈) *getWatchList* (*state*⦇*getM* := *getM state* @ [(*l*, *d*)]⦈) (*opposite l*)]

by (*auto simp add*: *Let-def*)

lemma *WatchInvariantsAfterAssertLiteral*:
assumes
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
shows
let state' = (*assertLiteral literal decision state*) *in*
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state'*) (*getF state'*) ∧
InvariantWatchListsUniq (*getWatchList state'*) ∧
InvariantWatchListsCharacterization (*getWatchList state'*) (*getWatch1 state'*) (*getWatch2 state'*) ∧
InvariantWatchesEl (*getF state'*) (*getWatch1 state'*) (*getWatch2 state'*) ∧
InvariantWatchesDiffer (*getF state'*) (*getWatch1 state'*) (*getWatch2 state'*)

using *assms*
unfolding *assertLiteral-def*
unfolding *notifyWatches-def*
using *InvariantWatchesElNotifyWatchesLoop*[*of state*(|*getM* := *getM state* @ [(*literal*, *decision*)]|) *getWatchList state* (*opposite literal*) *opposite literal* []]
using *InvariantWatchesDifferNotifyWatchesLoop*[*of state*(|*getM* := *getM state* @ [(*literal*, *decision*)]|) *getWatchList state* (*opposite literal*) *opposite literal* []]
using *InvariantWatchListsContainOnlyClausesFromFNotifyWatchesLoop*[*of state*(|*getM* := *getM state* @ [(*literal*, *decision*)]|) *getWatchList state* (*opposite literal*) [] *opposite literal*]
using *InvariantWatchListsCharacterizationNotifyWatchesLoop*[*of state*(|*getM* := *getM state* @ [(*literal*, *decision*)]|) (*getWatchList* (*state*(|*getM* := *getM state* @ [(*literal*, *decision*)]|)) (*opposite literal*)) *opposite literal* []]
unfolding *InvariantWatchListsContainOnlyClausesFromF-def*
unfolding *InvariantWatchListsCharacterization-def*
unfolding *InvariantWatchListsUniq-def*
by (*auto simp add*: *Let-def*)

lemma *InvariantWatchCharacterizationAfterAssertLiteral*:
assumes
InvariantConsistent ((*getM state*) @ [(*literal*, *decision*)]) **and**

InvariantUniq ((*getM state*) @ [(*literal*, *decision*)]) **and**
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*)
shows
let state′ = (*assertLiteral literal decision state*) *in*
InvariantWatchCharacterization (*getF state′*) (*getWatch1 state′*) (*getWatch2 state′*) (*getM state′*)
proof−
let *?state* = *state*⦇*getM* := *getM state* @ [(*literal*, *decision*)]⦈
let *?state′* = *assertLiteral literal decision state*
have ∗: ∀ *c*. *c* ∈ *set* (*getWatchList ?state* (*opposite literal*)) ⟶
(∀ *w1 w2*. *Some w1* = *getWatch1 ?state′ c* ∧ *Some w2* = *getWatch2 ?state′ c* ⟶
watchCharacterizationCondition w1 w2 (*getM ?state′*) (*getF ?state′* ! *c*) ∧
watchCharacterizationCondition w2 w1 (*getM ?state′*) (*getF ?state′* ! *c*))
using *assms*
using *NotifyWatchesLoopWatchCharacterizationEffect*[*of ?state getM state getWatchList ?state* (*opposite literal*) *opposite literal decision* []]
unfolding *InvariantWatchListsContainOnlyClausesFromF-def*
unfolding *InvariantWatchListsUniq-def*
unfolding *InvariantWatchListsCharacterization-def*
unfolding *assertLiteral-def*
unfolding *notifyWatches-def*
by (*simp add*: *Let-def*)
{
fix *c*
assume *0* ≤ *c* **and** *c* < *length* (*getF ?state′*)
fix *w1*::*Literal* **and** *w2*::*Literal*
assume *Some w1* = *getWatch1 ?state′ c Some w2* = *getWatch2 ?state′ c*
have *watchCharacterizationCondition w1 w2* (*getM ?state′*) (*getF ?state′* ! *c*) ∧
watchCharacterizationCondition w2 w1 (*getM ?state′*) (*getF ?state′* ! *c*)
proof (*cases c* ∈ *set* (*getWatchList ?state* (*opposite literal*)))
case *True*
thus *?thesis*
using ∗

using ‹*Some w1 = getWatch1 ?state′ c*› ‹*Some w2 = getWatch2 ?state′ c*›
by *auto*
next
case *False*
hence *Some* (*opposite literal*) ≠ *getWatch1 state c* **and** *Some* (*opposite literal*) ≠ *getWatch2 state c*
using ‹*InvariantWatchListsCharacterization* (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)›
unfolding *InvariantWatchListsCharacterization-def*
by *auto*
moreover
from *assms False*
have *getWatch1 ?state′ c = getWatch1 state c* **and** *getWatch2 ?state′ c = getWatch2 state c*
using *notifyWatchesLoopPreservedWatches*[*of ?state getWatchList ?state* (*opposite literal*) *opposite literal* []]
using *False*
unfolding *assertLiteral-def*
unfolding *notifyWatches-def*
unfolding *InvariantWatchListsContainOnlyClausesFromF-def*
by (*auto simp add*: *Let-def*)
ultimately
have *w1* ≠ *opposite literal w2* ≠ *opposite literal*
using ‹*Some w1 = getWatch1 ?state′ c*› **and** ‹*Some w2 = getWatch2 ?state′ c*›
by *auto*

have *watchCharacterizationCondition w1 w2* (*getM state*) (*getF state* ! *c*) **and**
watchCharacterizationCondition w2 w1 (*getM state*) (*getF state* ! *c*)
using ‹*InvariantWatchCharacterization* (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*)›
using ‹*Some w1 = getWatch1 ?state′ c*› **and** ‹*Some w2 = getWatch2 ?state′ c*›
using ‹*getWatch1 ?state′ c = getWatch1 state c*› **and** ‹*getWatch2 ?state′ c = getWatch2 state c*›
unfolding *InvariantWatchCharacterization-def*
using ‹*c* < *length* (*getF ?state′*)›
using *assms*
using *assertLiteralEffect*[*of state literal decision*]
by *auto*

have *watchCharacterizationCondition w1 w2* (*getM ?state′*) ((*getF ?state′*) ! *c*)
proof−
{
assume *literalFalse w1* (*elements* (*getM ?state′*))

with ‹*w1* ≠ *opposite literal*›
have *literalFalse w1* (*elements* (*getM state*))
using *assms*
using *assertLiteralEffect*[*of state literal decision*]
by *simp*
with ‹*watchCharacterizationCondition w1 w2* (*getM state*) (*getF state* ! *c*)›
have (∃ *l*. *l el* ((*getF state*) ! *c*) ∧ *literalTrue l* (*elements* (*getM state*))
∧ *elementLevel l* (*getM state*) ≤ *elementLevel* (*opposite w1*) (*getM state*)) ∨
(∀ *l*. *l el* (*getF state* ! *c*) ∧ *l* ≠ *w1* ∧ *l* ≠ *w2* ⟶
literalFalse l (*elements* (*getM state*)) ∧
elementLevel (*opposite l*) (*getM state*) ≤ *elementLevel* (*opposite w1*) (*getM state*)) (**is** *?a state* ∨ *?b state*)
unfolding *watchCharacterizationCondition-def*
using *assms*
using *assertLiteralEffect*[*of state literal decision*]
using ‹*w1* ≠ *opposite literal*›
by *simp*
have *?a ?state′* ∨ *?b ?state′*
proof (*cases ?b state*)
case *True*
show *?thesis*
proof−
{
fix *l*
assume *l el* (*nth* (*getF ?state′*) *c*) *l* ≠ *w1 l* ≠ *w2*
have *literalFalse l* (*elements* (*getM ?state′*)) ∧
elementLevel (*opposite l*) (*getM ?state′*) ≤ *elementLevel* (*opposite w1*) (*getM ?state′*)
proof−
from *True* ‹*l el* (*nth* (*getF ?state′*) *c*)› ‹*l* ≠ *w1*› ‹*l* ≠ *w2*›
have *literalFalse l* (*elements* (*getM state*))
elementLevel (*opposite l*) (*getM state*) ≤ *elementLevel* (*opposite w1*) (*getM state*)
using *assms*
using *assertLiteralEffect*[*of state literal decision*]
by *auto*
thus *?thesis*
using ‹*literalFalse w1* (*elements* (*getM state*))›
using *elementLevelAppend*[*of opposite w1 getM state* [(*literal*, *decision*)]]
using *elementLevelAppend*[*of opposite l getM state* [(*literal*, *decision*)]]
using *assms*
using *assertLiteralEffect*[*of state literal decision*]
by *auto*

```
            qed
          }
          thus ?thesis
            by simp
        qed
      next
        case False
        with ‹?a state ∨ ?b state›
        obtain l::Literal
            where l el (getF state ! c) literalTrue l (elements (getM state))
          elementLevel l (getM state) ≤ elementLevel (opposite w1) (getM state)
          by auto

        from ‹w1 ≠ opposite literal›
          ‹literalFalse w1 (elements (getM ?state'))›
         have elementLevel (opposite w1) ((getM state) @ [(literal, decision)]) = elementLevel (opposite w1) (getM state)
          using assms
          using assertLiteralEffect[of state literal decision]
          unfolding elementLevel-def
          by (simp add: markedElementsToAppend)
        moreover
        from ‹literalTrue l (elements (getM state))›
        have elementLevel l ((getM state) @ [(literal, decision)]) = elementLevel l (getM state)
          unfolding elementLevel-def
          by (simp add: markedElementsToAppend)
        ultimately
        have elementLevel l ((getM state) @ [(literal, decision)]) ≤ elementLevel (opposite w1) ((getM state) @ [(literal, decision)])
        using ‹elementLevel l (getM state) ≤ elementLevel (opposite w1) (getM state)›
          by simp
        thus ?thesis
          using ‹l el (getF state ! c)› ‹literalTrue l (elements (getM state))›
          using assms
          using assertLiteralEffect[of state literal decision]
          by auto
      qed
    }
    thus ?thesis
      unfolding watchCharacterizationCondition-def
      by auto
  qed
  moreover
  have watchCharacterizationCondition w2 w1 (getM ?state') ((getF
```

?state') ! *c*)
proof −
{
assume *literalFalse w2* (*elements* (*getM ?state'*))
with ‹*w2* ≠ *opposite literal*›
have *literalFalse w2* (*elements* (*getM state*))
using *assms*
using *assertLiteralEffect*[*of state literal decision*]
by *simp*
with ‹*watchCharacterizationCondition w2 w1* (*getM state*) (*getF state* ! *c*)›
have (∃ *l*. *l el* ((*getF state*) ! *c*) ∧ *literalTrue l* (*elements* (*getM state*))
∧ *elementLevel l* (*getM state*) ≤ *elementLevel* (*opposite w2*) (*getM state*)) ∨
(∀ *l*. *l el* (*getF state* ! *c*) ∧ *l* ≠ *w2* ∧ *l* ≠ *w1* ⟶
literalFalse l (*elements* (*getM state*)) ∧
elementLevel (*opposite l*) (*getM state*) ≤ *elementLevel* (*opposite w2*) (*getM state*)) (**is** *?a state* ∨ *?b state*)
unfolding *watchCharacterizationCondition-def*
using *assms*
using *assertLiteralEffect*[*of state literal decision*]
using ‹*w2* ≠ *opposite literal*›
by *simp*
have *?a ?state'* ∨ *?b ?state'*
proof (*cases ?b state*)
case *True*
show *?thesis*
proof −
{
fix *l*
assume *l el* (*nth* (*getF ?state'*) *c*) *l* ≠ *w1* *l* ≠ *w2*
have *literalFalse l* (*elements* (*getM ?state'*)) ∧
elementLevel (*opposite l*) (*getM ?state'*) ≤ *elementLevel* (*opposite w2*) (*getM ?state'*)
proof −
from *True* ‹*l el* (*nth* (*getF ?state'*) *c*)› ‹*l* ≠ *w1*› ‹*l* ≠ *w2*›
have *literalFalse l* (*elements* (*getM state*))
elementLevel (*opposite l*) (*getM state*) ≤ *elementLevel* (*opposite w2*) (*getM state*)
using *assms*
using *assertLiteralEffect*[*of state literal decision*]
by *auto*
thus *?thesis*
using ‹*literalFalse w2* (*elements* (*getM state*))›
using *elementLevelAppend*[*of opposite w2 getM state* [(*literal*, *decision*)]]
using *elementLevelAppend*[*of opposite l getM state*

```
[(literal, decision)]]
                using assms
                using assertLiteralEffect[of state literal decision]
                by auto
            qed
          }
          thus ?thesis
            by simp
        qed
      next
        case False
        with ‹?a state ∨ ?b state›
        obtain l::Literal
            where l el (getF state ! c) literalTrue l (elements (getM
state))
          elementLevel l (getM state) ≤ elementLevel (opposite w2)
(getM state)
          by auto

        from ‹w2 ≠ opposite literal›
          ‹literalFalse w2 (elements (getM ?state'))›
         have elementLevel (opposite w2) ((getM state) @ [(literal,
decision)]) = elementLevel (opposite w2) (getM state)
          using assms
          using assertLiteralEffect[of state literal decision]
          unfolding elementLevel-def
          by (simp add: markedElementsToAppend)
        moreover
        from ‹literalTrue l (elements (getM state))›
        have elementLevel l ((getM state) @ [(literal, decision)]) =
elementLevel l (getM state)
          unfolding elementLevel-def
          by (simp add: markedElementsToAppend)
        ultimately
        have elementLevel l ((getM state) @ [(literal, decision)]) ≤
elementLevel (opposite w2) ((getM state) @ [(literal, decision)])
        using ‹elementLevel l (getM state) ≤ elementLevel (opposite
w2) (getM state)›
          by simp
        thus ?thesis
          using ‹l el (getF state ! c)› ‹literalTrue l (elements (getM
state))›
          using assms
          using assertLiteralEffect[of state literal decision]
          by auto
      qed
    }
    thus ?thesis
      unfolding watchCharacterizationCondition-def
```

by *auto*
qed
ultimately
show *?thesis*
by *simp*
qed
}
thus *?thesis*
unfolding *InvariantWatchCharacterization-def*
by (*simp add*: *Let-def*)
qed

lemma *assertLiteralConflictFlagEffect*:
assumes
InvariantConsistent ((*getM state*) @ [(*literal*, *decision*)])
InvariantUniq ((*getM state*) @ [(*literal*, *decision*)])
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*)
InvariantWatchListsUniq (*getWatchList state*)
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*)
shows
let state′ = assertLiteral literal decision state in
getConflictFlag state′ = (*getConflictFlag state* ∨
(∃ *clause*. *clause el* (*getF state*) ∧
opposite literal el clause ∧
clauseFalse clause ((*elements* (*getM state*)) @ [*literal*])))
proof−
let *?state* = *state*⦇*getM* := *getM state* @ [(*literal*, *decision*)]⦈
let *?state′* = *assertLiteral literal decision state*

have *getConflictFlag ?state′* = (*getConflictFlag state* ∨
(∃ *clause*. *clause* ∈ *set* (*getWatchList ?state* (*opposite literal*)) ∧
clauseFalse (*nth* (*getF ?state*) *clause*) (*elements* (*getM ?state*))))
using *NotifyWatchesLoopConflictFlagEffect*[*of ?state*
getWatchList ?state (*opposite literal*)
opposite literal []]
using ‹*InvariantConsistent* ((*getM state*) @ [(*literal*, *decision*)])›
using ‹*InvariantWatchListsContainOnlyClausesFromF* (*getWatchList state*) (*getF state*)›
using ‹*InvariantWatchListsUniq* (*getWatchList state*)›
using ‹*InvariantWatchListsCharacterization* (*getWatchList state*)

(*getWatch1 state*) (*getWatch2 state*)›
  using ‹*InvariantWatchesEl* (*getF state*) (*getWatch1 state*) (*getWatch2 state*)›
    unfolding *InvariantWatchListsUniq-def*
    unfolding *InvariantWatchListsCharacterization-def*
    unfolding *InvariantWatchListsContainOnlyClausesFromF-def*
    unfolding *assertLiteral-def*
    unfolding *notifyWatches-def*
    by (*simp add*: *Let-def*)
  moreover
  have (∃ *clause*. *clause* ∈ *set* (*getWatchList ?state* (*opposite literal*)) ∧
    clauseFalse (*nth* (*getF ?state*) *clause*) (*elements* (*getM ?state*))) =
    (∃ *clause*. *clause el* (*getF state*) ∧
      opposite literal el clause ∧
      clauseFalse clause ((*elements* (*getM state*)) @ [*literal*]))
(**is** *?lhs* = *?rhs*)
  proof
    assume *?lhs*
    then obtain *clause*
      where *clause* ∈ *set* (*getWatchList ?state* (*opposite literal*))
      clauseFalse (*nth* (*getF ?state*) *clause*) (*elements* (*getM ?state*))
      by *auto*

  have *getWatch1 ?state clause* = *Some* (*opposite literal*) ∨ *getWatch2 ?state clause* = *Some* (*opposite literal*)
      clause < *length* (*getF ?state*)
      ∃ *w1 w2*. *getWatch1 ?state clause* = *Some w1* ∧ *getWatch2 ?state clause* = *Some w2* ∧
      w1 el (*nth* (*getF ?state*) *clause*) ∧ *w2 el* (*nth* (*getF ?state*) *clause*)
      using ‹*clause* ∈ *set* (*getWatchList ?state* (*opposite literal*))›
      using *assms*
      unfolding *InvariantWatchListsContainOnlyClausesFromF-def*
      unfolding *InvariantWatchesEl-def*
      unfolding *InvariantWatchListsCharacterization-def*
      by *auto*
    hence (*nth* (*getF ?state*) *clause*) *el* (*getF ?state*)
      opposite literal el (*nth* (*getF ?state*) *clause*)
      using *nth-mem*[*of clause getF ?state*]
      by *auto*
    thus *?rhs*
      using ‹*clauseFalse* (*nth* (*getF ?state*) *clause*) (*elements* (*getM ?state*))›
      by *auto*
  next
    assume *?rhs*
    then obtain *clause*
      where *clause el* (*getF ?state*)

opposite literal el clause
clauseFalse clause ((*elements* (*getM state*)) @ [*literal*])
by *auto*
then obtain *ci*
where *clause* = (*nth* (*getF ?state*) *ci*) *ci* < *length* (*getF ?state*)
by (*auto simp add*: *in-set-conv-nth*)
moreover
from ‹*ci* < *length* (*getF ?state*)›
obtain *w1 w2*
where *getWatch1 state ci* = *Some w1 getWatch2 state ci* = *Some w2*
w1 el (*nth* (*getF state*) *ci*) *w2 el* (*nth* (*getF state*) *ci*)
using *assms*
unfolding *InvariantWatchesEl-def*
by *auto*
have *getWatch1 state ci* = *Some* (*opposite literal*) ∨ *getWatch2 state ci* = *Some* (*opposite literal*)
proof−
{
assume ¬ *?thesis*
with ‹*clauseFalse clause* ((*elements* (*getM state*)) @ [*literal*])›
‹*clause* = (*nth* (*getF ?state*) *ci*)›
‹*getWatch1 state ci* = *Some w1*› ‹*getWatch2 state ci* = *Some w2*›
‹*w1 el* (*nth* (*getF state*) *ci*)› ‹*w2 el* (*nth* (*getF state*) *ci*)›
have *literalFalse w1* (*elements* (*getM state*)) *literalFalse w2* (*elements* (*getM state*))
by (*auto simp add*: *clauseFalseIffAllLiteralsAreFalse*)

from ‹*InvariantConsistent* ((*getM state*) @ [(*literal*, *decision*)])›
‹*clauseFalse clause* ((*elements* (*getM state*)) @ [*literal*])›
have ¬ (∃ *l*. *l el clause* ∧ *literalTrue l* (*elements* (*getM state*)))
unfolding *InvariantConsistent-def*
by (*auto simp add*: *inconsistentCharacterization clauseFalseIffAllLiteralsAreFalse*)

from ‹*InvariantUniq* ((*getM state*) @ [(*literal*, *decision*)])›
have ¬ *literalTrue literal* (*elements* (*getM state*))
unfolding *InvariantUniq-def*
by (*auto simp add*: *uniqAppendIff*)

from ‹*InvariantWatchCharacterization* (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*)›
‹*literalFalse w1* (*elements* (*getM state*))› ‹*literalFalse w2* (*elements* (*getM state*))›
‹¬ (∃ *l*. *l el clause* ∧ *literalTrue l* (*elements* (*getM state*)))›

‹*getWatch1 state ci* = *Some w1*›[*THEN sym*]
‹*getWatch2 state ci* = *Some w2*›[*THEN sym*]
‹*ci* < *length* (*getF ?state*)›
‹*clause* = (*nth* (*getF ?state*) *ci*)›
have ∀ *l*. *l el clause* ∧ *l* ≠ *w1* ∧ *l* ≠ *w2* ⟶ *literalFalse l* (*elements* (*getM state*))
unfolding *InvariantWatchCharacterization-def*
unfolding *watchCharacterizationCondition-def*
by *auto*
hence *literalTrue literal* (*elements* (*getM state*))
using ‹¬ (*getWatch1 state ci* = *Some* (*opposite literal*) ∨ *getWatch2 state ci* = *Some* (*opposite literal*))›
using ‹*opposite literal el clause*›
using ‹*getWatch1 state ci* = *Some w1*›
using ‹*getWatch2 state ci* = *Some w2*›
by *auto*
with ‹¬ *literalTrue literal* (*elements* (*getM state*))›
have *False*
by *simp*
}
thus *?thesis*
by *auto*
qed
ultimately
show *?lhs*
using *assms*
using ‹*clauseFalse clause* ((*elements* (*getM state*)) @ [*literal*])›
unfolding *InvariantWatchListsCharacterization-def*
by *force*
qed
ultimately
show *?thesis*
by *auto*
qed

lemma *InvariantConflictFlagCharacterizationAfterAssertLiteral*:
assumes
InvariantConsistent ((*getM state*) @ [(*literal*, *decision*)])
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**

InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*)
InvariantConflictFlagCharacterization (*getConflictFlag state*) (*getF state*) (*getM state*)
shows
let state′ = (*assertLiteral literal decision state*) *in*
InvariantConflictFlagCharacterization (*getConflictFlag state′*) (*getF state′*) (*getM state′*)
proof−
let *?state* = *state*⦇*getM* := *getM state* @ [(*literal*, *decision*)]⦈
let *?state′* = *assertLiteral literal decision state*

have ∗:*getConflictFlag ?state′* = (*getConflictFlag state* ∨
(∃ *clause*. *clause* ∈ *set* (*getWatchList ?state* (*opposite literal*)) ∧
clauseFalse (*nth* (*getF ?state*) *clause*) (*elements* (*getM ?state*))))
using *NotifyWatchesLoopConflictFlagEffect*[*of ?state*
getWatchList ?state (*opposite literal*)
opposite literal []]
using ‹*InvariantWatchesEl* (*getF state*) (*getWatch1 state*) (*getWatch2 state*)›
using ‹*InvariantConsistent* ((*getM state*) @ [(*literal*, *decision*)])›
using ‹*InvariantWatchListsContainOnlyClausesFromF* (*getWatchList state*) (*getF state*)›
using ‹*InvariantWatchListsUniq* (*getWatchList state*)›
using ‹*InvariantWatchListsCharacterization* (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)›
unfolding *InvariantWatchListsUniq-def*
unfolding *InvariantWatchListsCharacterization-def*
unfolding *InvariantWatchListsContainOnlyClausesFromF-def*
unfolding *assertLiteral-def*
unfolding *notifyWatches-def*
by (*simp add*: *Let-def*)

hence *getConflictFlag state* ⟶ *getConflictFlag ?state′*
by *simp*

show *?thesis*
proof (*cases getConflictFlag state*)
case *True*
thus *?thesis*
using ‹*InvariantConflictFlagCharacterization* (*getConflictFlag state*) (*getF state*) (*getM state*)›
using *assertLiteralEffect*[*of state literal decision*]
using ‹*getConflictFlag state* ⟶ *getConflictFlag ?state′*›
using *assms*
unfolding *InvariantConflictFlagCharacterization-def*
by (*auto simp add*: *Let-def formulaFalseAppendValuation*)

next
case *False*

hence ¬ *formulaFalse* (*getF state*) (*elements* (*getM state*))
using ‹*InvariantConflictFlagCharacterization* (*getConflictFlag state*) (*getF state*) (*getM state*)›
unfolding *InvariantConflictFlagCharacterization-def*
by *simp*

have ∗∗: ∀ *clause. clause* ∉ *set* (*getWatchList ?state* (*opposite literal*)) ∧
0 ≤ *clause* ∧ *clause* < *length* (*getF ?state*) ⟶
¬ *clauseFalse* (*nth* (*getF ?state*) *clause*) (*elements* (*getM ?state*))
proof−
{
fix *clause*
assume *clause* ∉ *set* (*getWatchList ?state* (*opposite literal*)) **and**
0 ≤ *clause* ∧ *clause* < *length* (*getF ?state*)

from ‹*0* ≤ *clause* ∧ *clause* < *length* (*getF ?state*)›
obtain *w1*::*Literal* **and** *w2*::*Literal*
where *getWatch1 ?state clause* = *Some w1* **and**
getWatch2 ?state clause = *Some w2* **and**
w1 el (*nth* (*getF ?state*) *clause*) **and**
w2 el (*nth* (*getF ?state*) *clause*)
using ‹*InvariantWatchesEl* (*getF state*) (*getWatch1 state*) (*getWatch2 state*)›
unfolding *InvariantWatchesEl-def*
by *auto*

have ¬ *clauseFalse* (*nth* (*getF ?state*) *clause*) (*elements* (*getM ?state*))
proof−
from ‹*clause* ∉ *set* (*getWatchList ?state* (*opposite literal*))›
have *w1* ≠ *opposite literal* **and**
w2 ≠ *opposite literal*
using ‹*InvariantWatchListsCharacterization* (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)›
using ‹*getWatch1 ?state clause* = *Some w1*› **and** ‹*getWatch2 ?state clause* = *Some w2*›
unfolding *InvariantWatchListsCharacterization-def*
by *auto*

from ‹¬ *formulaFalse* (*getF state*) (*elements* (*getM state*))›
have ¬ *clauseFalse* (*nth* (*getF ?state*) *clause*) (*elements* (*getM state*))
using ‹*0* ≤ *clause* ∧ *clause* < *length* (*getF ?state*)›

by (*simp add*: *formulaFalseIffContainsFalseClause*)

show *?thesis*
proof (*cases literalFalse w1* (*elements* (*getM state*)) ∨ *literalFalse w2* (*elements* (*getM state*)))
case *True*

with ‹*InvariantWatchCharacterization* (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*)›
have $: (∃ *l*. *l el* (*nth* (*getF state*) *clause*) ∧ *literalTrue l* (*elements* (*getM state*))) ∨
(∀ *l*. *l el* (*nth* (*getF state*) *clause*) ∧
l ≠ *w1* ∧ *l* ≠ *w2* ⟶ *literalFalse l* (*elements* (*getM state*)))

using ‹*getWatch1 ?state clause* = *Some w1*›[*THEN sym*]
using ‹*getWatch2 ?state clause* = *Some w2*›[*THEN sym*]
using ‹*0* ≤ *clause* ∧ *clause* < *length* (*getF ?state*)›
unfolding *InvariantWatchCharacterization-def*
unfolding *watchCharacterizationCondition-def*
by *auto*

thus *?thesis*
proof (*cases* ∀ *l*. *l el* (*nth* (*getF state*) *clause*) ∧
l ≠ *w1* ∧ *l* ≠ *w2* ⟶ *literalFalse l* (*elements* (*getM state*)))
case *True*
have ¬ *literalFalse w1* (*elements* (*getM state*)) ∨ ¬ *literalFalse w2* (*elements* (*getM state*))
proof−
from ‹¬ *clauseFalse* (*nth* (*getF ?state*) *clause*) (*elements* (*getM state*))›
obtain *l*::*Literal*
where *l el* (*nth* (*getF ?state*) *clause*) **and** ¬ *literalFalse l* (*elements* (*getM state*))
by (*auto simp add*: *clauseFalseIffAllLiteralsAreFalse*)
with *True*
show *?thesis*
by *auto*
qed
hence ¬ *literalFalse w1* (*elements* (*getM ?state*)) ∨ ¬ *literalFalse w2* (*elements* (*getM ?state*))
using ‹*w1* ≠ *opposite literal*› **and** ‹*w2* ≠ *opposite literal*›
by *auto*
thus *?thesis*
using ‹*w1 el* (*nth* (*getF ?state*) *clause*)› ‹*w2 el* (*nth* (*getF ?state*) *clause*)›
by (*auto simp add*: *clauseFalseIffAllLiteralsAreFalse*)

```
          next
            case False
            then obtain l::Literal
                where l el (nth (getF state) clause) and literalTrue l
(elements (getM state))
              using $
              by auto
            thus ?thesis
                using ‹InvariantConsistent ((getM state) @ [(literal,
decision)])›
              unfolding InvariantConsistent-def
                by (auto simp add: clauseFalseIffAllLiteralsAreFalse
inconsistentCharacterization)
          qed
        next
          case False
          thus ?thesis
            using ‹w1 el (nth (getF ?state) clause)› and
              ‹w1 ≠ opposite literal›
            by (auto simp add: clauseFalseIffAllLiteralsAreFalse)
        qed
      qed
    } thus ?thesis
      by simp
  qed

  show ?thesis
  proof (cases getConflictFlag ?state')
    case True
    from ‹¬ getConflictFlag state› ‹getConflictFlag ?state'›
    obtain clause::nat
      where
      clause ∈ set (getWatchList ?state (opposite literal)) and
       clauseFalse (nth (getF ?state) clause) (elements (getM ?state))
      using ∗
      by auto
   from ‹InvariantWatchListsContainOnlyClausesFromF (getWatchList
state) (getF state)›
      ‹clause ∈ set (getWatchList ?state (opposite literal))›
    have (nth (getF ?state) clause) el (getF ?state)
      unfolding InvariantWatchListsContainOnlyClausesFromF-def
      using nth-mem
      by simp
      with ‹clauseFalse (nth (getF ?state) clause) (elements (getM
?state))›
    have formulaFalse (getF ?state) (elements (getM ?state))
     by (auto simp add: Let-def formulaFalseIffContainsFalseClause)

    thus ?thesis
```

using ‹¬ *getConflictFlag state*› ‹*getConflictFlag ?state'*›
unfolding *InvariantConflictFlagCharacterization-def*
using *assms*
using *assertLiteralEffect*[*of state literal decision*]
by (*simp add*: *Let-def*)
next
case *False*
hence ∀ *clause*::*nat*. *clause* ∈ *set* (*getWatchList ?state* (*opposite literal*)) ⟶
¬ *clauseFalse* (*nth* (*getF ?state*) *clause*) (*elements* (*getM ?state*))
using ∗
by *auto*
with ∗∗
have ∀ *clause*. *0* ≤ *clause* ∧ *clause* < *length* (*getF ?state*) ⟶
¬ *clauseFalse* (*nth* (*getF ?state*) *clause*) (*elements* (*getM ?state*))
by *auto*
hence ¬ *formulaFalse* (*getF ?state*) (*elements* (*getM ?state*))
by (*auto simp add*:*set-conv-nth formulaFalseIffContainsFalseClause*)
thus *?thesis*
using ‹¬ *getConflictFlag state*› ‹¬ *getConflictFlag ?state'*›
using *assms*
unfolding *InvariantConflictFlagCharacterization-def*
by (*auto simp add*: *Let-def assertLiteralEffect*)
qed
qed
qed

lemma *InvariantConflictClauseCharacterizationAfterAssertLiteral*:
assumes
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
and
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*)
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchListsUniq (*getWatchList state*)
InvariantConflictClauseCharacterization (*getConflictFlag state*) (*getConflictClause state*) (*getF state*) (*getM state*)
shows
let state' = *assertLiteral literal decision state in*
InvariantConflictClauseCharacterization (*getConflictFlag state'*) (*getConflictClause state'*) (*getF state'*) (*getM state'*)
proof−
let *?state0* = *state*⦇ *getM* := *getM state* @ [(*literal*, *decision*)]⦈
show *?thesis*
using *assms*
using *InvariantConflictClauseCharacterizationAfterNotifyWatches*[*of*

?state0 getM state opposite literal decision
getWatchList ?state0 (*opposite literal*) []]
unfolding *assertLiteral-def*
unfolding *notifyWatches-def*
unfolding *InvariantWatchListsUniq-def*
unfolding *InvariantWatchListsContainOnlyClausesFromF-def*
unfolding *InvariantWatchListsCharacterization-def*
unfolding *InvariantConflictClauseCharacterization-def*
by (*simp add*: *Let-def clauseFalseAppendValuation*)
qed

lemma *assertLiteralQEffect*:
assumes
InvariantConsistent ((*getM state*) @ [(*literal*, *decision*)])
InvariantUniq ((*getM state*) @ [(*literal*, *decision*)])
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*)
InvariantWatchListsUniq (*getWatchList state*)
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*)
shows
let state′ = assertLiteral literal decision state in
set (*getQ state′*) = *set* (*getQ state*) ∪
{ *ul.* (∃ *uc. uc el* (*getF state*) ∧
opposite literal el uc ∧
isUnitClause uc ul ((*elements* (*getM state*)) @ [*literal*])) }
(**is** *let state′ = assertLiteral literal decision state in*
set (*getQ state′*) = *set* (*getQ state*) ∪ *?ulSet*)
proof−
let *?state′* = *state*⦇*getM* := *getM state* @ [(*literal*, *decision*)]⦈
let *?state″* = *assertLiteral literal decision state*

have *set* (*getQ ?state″*) − *set* (*getQ state*) ⊆ *?ulSet*
unfolding *assertLiteral-def*
unfolding *notifyWatches-def*
using *assms*
using *NotifyWatchesLoopQEffect*[*of ?state′ getM state opposite literal decision getWatchList ?state′* (*opposite literal*) []]
unfolding *InvariantWatchListsCharacterization-def*
unfolding *InvariantWatchListsUniq-def*
unfolding *InvariantWatchListsContainOnlyClausesFromF-def*
using *set-conv-nth*[*of getF state*]
by (*auto simp add*: *Let-def*)

moreover
have *?ulSet* ⊆ *set* (*getQ ?state''*)
proof
fix *ul*
assume *ul* ∈ *?ulSet*
then obtain *uc*
where *uc el* (*getF state*) *opposite literal el uc isUnitClause uc ul* ((*elements* (*getM state*)) @ [*literal*])
by *auto*
then obtain *uci*
where *uc* = (*nth* (*getF state*) *uci*) *uci* < *length* (*getF state*)
using *set-conv-nth*[*of getF state*]
by *auto*
let *?w1* = *getWatch1 state uci*
let *?w2* = *getWatch2 state uci*

have *?w1* = *Some* (*opposite literal*) ∨ *?w2* = *Some* (*opposite literal*)
proof−
{
assume ¬ *?thesis*

from ‹*InvariantWatchesEl* (*getF state*) (*getWatch1 state*) (*getWatch2 state*)›
obtain *wl1 wl2*
where *?w1* = *Some wl1 ?w2* = *Some wl2 wl1 el* (*getF state* ! *uci*) *wl2 el* (*getF state* ! *uci*)
unfolding *InvariantWatchesEl-def*
using ‹*uci* < *length* (*getF state*)›
by *force*

with ‹*InvariantWatchCharacterization* (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*)›
have *watchCharacterizationCondition wl1 wl2* (*getM state*) (*getF state* ! *uci*)
watchCharacterizationCondition wl2 wl1 (*getM state*) (*getF state* ! *uci*)
using ‹*uci* < *length* (*getF state*)›
unfolding *InvariantWatchCharacterization-def*
by *auto*

from ‹*isUnitClause uc ul* ((*elements* (*getM state*)) @ [*literal*])›
have ¬ (∃ *l*. *l el uc* ∧ (*literalTrue l* ((*elements* (*getM state*)) @ [*literal*])))
using *containsTrueNotUnit*
using ‹*InvariantConsistent* ((*getM state*) @ [(*literal*, *decision*)])›
unfolding *InvariantConsistent-def*
by *auto*

from ‹*InvariantUniq* ((*getM state*) @ [(*literal*, *decision*)])›
have ¬ *literal el* (*elements* (*getM state*))
  unfolding *InvariantUniq-def*
  by (*simp add*: *uniqAppendIff*)

from ‹¬ *?thesis*›
  ‹*?w1* = *Some wl1*› ‹*?w2* = *Some wl2*›
have *wl1* ≠ *opposite literal wl2* ≠ *opposite literal*
  by *auto*

from ‹*InvariantWatchesDiffer* (*getF state*) (*getWatch1 state*) (*getWatch2 state*)›
have *wl1* ≠ *wl2*
  using ‹*?w1* = *Some wl1*› ‹*?w2* = *Some wl2*›
  unfolding *InvariantWatchesDiffer-def*
  using ‹*uci* < *length* (*getF state*)›
  by *auto*

have *literalFalse wl1* (*elements* (*getM state*)) ∨ *literalFalse wl2* (*elements* (*getM state*))
proof (*cases ul* = *wl1*)
  case *True*
  with ‹*wl1* ≠ *wl2*›
  have *ul* ≠ *wl2*
    by *simp*
with ‹*isUnitClause uc ul* ((*elements* (*getM state*)) @ [*literal*])›
    ‹*wl2* ≠ *opposite literal*› ‹*wl2 el* (*getF state* ! *uci*)›
    ‹*uc* = (*getF state* ! *uci*)›
  show *?thesis*
    unfolding *isUnitClause-def*
    by *auto*
next
  case *False*
with ‹*isUnitClause uc ul* ((*elements* (*getM state*)) @ [*literal*])›
    ‹*wl1* ≠ *opposite literal*› ‹*wl1 el* (*getF state* ! *uci*)›
    ‹*uc* = (*getF state* ! *uci*)›
  show *?thesis*
    unfolding *isUnitClause-def*
    by *auto*
qed

with ‹*watchCharacterizationCondition wl1 wl2* (*getM state*) (*getF state* ! *uci*)›
  ‹*watchCharacterizationCondition wl2 wl1* (*getM state*) (*getF state* ! *uci*)›
  ‹¬ (∃ *l*. *l el uc* ∧ (*literalTrue l* ((*elements* (*getM state*)) @ [*literal*])))›
  ‹*uc* = (*getF state* ! *uci*)›

‹?w1 = Some wl1› ‹?w2 = Some wl2›
have ∀ *l. l el uc* ∧ *l* ≠ *wl1* ∧ *l* ≠ *wl2* ⟶ *literalFalse l* (*elements* (*getM state*))
unfolding *watchCharacterizationCondition-def*
by *auto*
with ‹*wl1* ≠ *opposite literal*› ‹*wl2* ≠ *opposite literal*› ‹*opposite literal el uc*›
have *literalTrue literal* (*elements* (*getM state*))
by *auto*
with ‹¬ *literal el* (*elements* (*getM state*))›
have *False*
by *simp*
} **thus** *?thesis*
by *auto*
qed
with ‹*InvariantWatchListsCharacterization* (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)›
have *uci* ∈ *set* (*getWatchList state* (*opposite literal*))
unfolding *InvariantWatchListsCharacterization-def*
by *auto*

thus *ul* ∈ *set* (*getQ ?state''*)
using ‹*uc el* (*getF state*)›
using ‹*isUnitClause uc ul* ((*elements* (*getM state*)) @ [*literal*])›
using ‹*uc* = (*getF state* ! *uci*)›
unfolding *assertLiteral-def*
unfolding *notifyWatches-def*
using *assms*
using *NotifyWatchesLoopQEffect*[*of ?state' getM state opposite literal decision getWatchList ?state'* (*opposite literal*) []]
unfolding *InvariantWatchListsCharacterization-def*
unfolding *InvariantWatchListsUniq-def*
unfolding *InvariantWatchListsContainOnlyClausesFromF-def*
by (*auto simp add*: *Let-def*)
qed
moreover
have *set* (*getQ state*) ⊆ *set* (*getQ ?state''*)
using *assms*
using *assertLiteralEffect*[*of state literal decision*]
using *prefixIsSubset*[*of getQ state getQ ?state''*]
by *simp*
ultimately
show *?thesis*
by (*auto simp add*: *Let-def*)
qed

lemma *InvariantQCharacterizationAfterAssertLiteral*:
assumes

InvariantConsistent ((*getM state*) @ [(*literal*, *decision*)])
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*)
InvariantConflictFlagCharacterization (*getConflictFlag state*) (*getF state*) (*getM state*)
InvariantQCharacterization (*getConflictFlag state*) (*getQ state*) (*getF state*) (*getM state*)
shows
let state' = (*assertLiteral literal decision state*) *in*
InvariantQCharacterization (*getConflictFlag state'*) (*removeAll literal* (*getQ state'*)) (*getF state'*) (*getM state'*)
proof−
let *?state* = *state*⦇*getM* := *getM state* @ [(*literal*, *decision*)]⦈
let *?state'* = *assertLiteral literal decision state*

have ∗:∀ *l*. *l* ∈ *set* (*getQ ?state'*) − *set* (*getQ ?state*) ⟶
(∃ *clause*. *clause el* (*getF ?state*) ∧ *isUnitClause clause l* (*elements* (*getM ?state*)))
using *NotifyWatchesLoopQEffect*[*of ?state getM state opposite literal decision getWatchList ?state* (*opposite literal*) []]
using *assms*
unfolding *InvariantWatchListsUniq-def*
unfolding *InvariantWatchListsCharacterization-def*
unfolding *InvariantWatchListsContainOnlyClausesFromF-def*
unfolding *InvariantWatchCharacterization-def*
unfolding *assertLiteral-def*
unfolding *notifyWatches-def*
by (*auto simp add*: *Let-def*)

have ∗∗: ∀ *clause*. *clause* ∈ *set* (*getWatchList ?state* (*opposite literal*)) ⟶
(∀ *l*. (*isUnitClause* (*nth* (*getF ?state*) *clause*) *l* (*elements* (*getM ?state*))) ⟶
l ∈ (*set* (*getQ ?state'*)))
using *NotifyWatchesLoopQEffect*[*of ?state getM state opposite literal decision getWatchList ?state* (*opposite literal*) []]
using *assms*
unfolding *InvariantWatchListsUniq-def*
unfolding *InvariantWatchListsCharacterization-def*
unfolding *InvariantWatchListsContainOnlyClausesFromF-def*

unfolding *InvariantWatchCharacterization-def*
unfolding *assertLiteral-def*
unfolding *notifyWatches-def*
by (*simp add*: *Let-def*)

have *getConflictFlag state* ⟶ *getConflictFlag ?state'*
proof−
have *getConflictFlag ?state'* = (*getConflictFlag state* ∨
(∃ *clause*. *clause* ∈ *set* (*getWatchList ?state* (*opposite literal*))
∧
clauseFalse (*nth* (*getF ?state*) *clause*) (*elements*
(*getM ?state*))))
using *NotifyWatchesLoopConflictFlagEffect*[*of ?state*
getWatchList ?state (*opposite literal*)
opposite literal []]
using *assms*
unfolding *InvariantWatchListsUniq-def*
unfolding *InvariantWatchListsCharacterization-def*
unfolding *InvariantWatchListsContainOnlyClausesFromF-def*
unfolding *assertLiteral-def*
unfolding *notifyWatches-def*
by (*simp add*: *Let-def*)
thus *?thesis*
by *simp*
qed

{
assume ¬ *getConflictFlag ?state'*
with ‹*getConflictFlag state* ⟶ *getConflictFlag ?state'*›
have ¬ *getConflictFlag state*
by *simp*

have ∀ *l*. *l el* (*removeAll literal* (*getQ ?state'*)) =
(∃ *c*. *c el* (*getF ?state'*) ∧ *isUnitClause c l* (*elements* (*getM*
?state')))
proof
fix *l*::*Literal*
show *l el* (*removeAll literal* (*getQ ?state'*)) =
(∃ *c*. *c el* (*getF ?state'*) ∧ *isUnitClause c l* (*elements* (*getM*
?state')))
proof
assume *l el* (*removeAll literal* (*getQ ?state'*))
hence *l el* (*getQ ?state'*) *l* ≠ *literal*
by *auto*
show ∃ *c*. *c el* (*getF ?state'*) ∧ *isUnitClause c l* (*elements* (*getM*
?state'))
proof (*cases l el* (*getQ state*))
case *True*

from ‹¬ *getConflictFlag state*›
‹*InvariantQCharacterization* (*getConflictFlag state*) (*getQ state*) (*getF state*) (*getM state*)›
‹*l el* (*getQ state*)›
obtain *c*::*Clause*
where *c el* (*getF state*) *isUnitClause c l* (*elements* (*getM state*))
unfolding *InvariantQCharacterization-def*
by *auto*

show *?thesis*
proof (*cases l* $\neq$ *opposite literal*)
case *True*
hence *opposite l* $\neq$ *literal*
by *auto*

from ‹*isUnitClause c l* (*elements* (*getM state*))›
‹*opposite l* $\neq$ *literal*› ‹*l* $\neq$ *literal*›
have *isUnitClause c l* ((*elements* (*getM state*) @ [*literal*]))
using *isUnitClauseAppendValuation*[*of c l elements* (*getM state*) *literal*]
by *simp*
thus *?thesis*
using *assms*
using ‹*c el* (*getF state*)›
using *assertLiteralEffect*[*of state literal decision*]
by *auto*
next
case *False*
hence *opposite l* = *literal*
by *simp*

from ‹*isUnitClause c l* (*elements* (*getM state*))›
have *clauseFalse c* (*elements* (*getM ?state'*))
using *assms*
using *assertLiteralEffect*[*of state literal decision*]
using *unitBecomesFalse*[*of c l elements* (*getM state*)]
using ‹*opposite l* = *literal*›
by *simp*
with ‹*c el* (*getF state*)›
have *formulaFalse* (*getF state*) (*elements* (*getM ?state'*))
by (*auto simp add*: *formulaFalseIffContainsFalseClause*)

from *assms*
have *InvariantConflictFlagCharacterization* (*getConflictFlag ?state'*) (*getF ?state'*) (*getM ?state'*)
using *InvariantConflictFlagCharacterizationAfterAssertLiteral*
by (*simp add*: *Let-def*)

with ‹*formulaFalse* (*getF state*) (*elements* (*getM ?state′*))›
have *getConflictFlag ?state′*
using *assms*
using *assertLiteralEffect*[*of state literal decision*]
unfolding *InvariantConflictFlagCharacterization-def*
by *auto*
with ‹¬ *getConflictFlag ?state′*›
show *?thesis*
by *simp*
qed
next
case *False*
then obtain *c*::*Clause*
where *c el* (*getF ?state′*) ∧ *isUnitClause c l* (*elements* (*getM ?state′*))
using ∗
using ‹*l el* (*getQ ?state′*)›
using *assms*
using *assertLiteralEffect*[*of state literal decision*]
by *auto*
thus *?thesis*
using *formulaEntailsItsClauses*[*of c getF ?state′*]
by *auto*
qed
next
assume ∃ *c*. *c el* (*getF ?state′*) ∧ *isUnitClause c l* (*elements* (*getM ?state′*))
then obtain *c*::*Clause*
where *c el* (*getF ?state′*) *isUnitClause c l* (*elements* (*getM ?state′*))
by *auto*
then obtain *ci*::*nat*
where *0* ≤ *ci ci* < *length* (*getF ?state′*) *c* = (*nth* (*getF ?state′*) *ci*)
using *set-conv-nth*[*of getF ?state′*]
by *auto*
then obtain *w1*::*Literal* **and** *w2*::*Literal*
where *getWatch1 state ci* = *Some w1* **and** *getWatch2 state ci* = *Some w2* **and**
w1 el c **and** *w2 el c*
using ‹*InvariantWatchesEl* (*getF state*) (*getWatch1 state*) (*getWatch2 state*)›
using ‹*c* = (*nth* (*getF ?state′*) *ci*)›
unfolding *InvariantWatchesEl-def*
using *assms*
using *assertLiteralEffect*[*of state literal decision*]
by *auto*
hence *w1* ≠ *w2*
using ‹*ci* < *length* (*getF ?state′*)›

using ‹*InvariantWatchesDiffer* (*getF state*) (*getWatch1 state*) (*getWatch2 state*)›
unfolding *InvariantWatchesDiffer-def*
using *assms*
using *assertLiteralEffect*[*of state literal decision*]
by *auto*

show *l el* (*removeAll literal* (*getQ ?state′*))
proof (*cases isUnitClause c l* (*elements* (*getM state*)))
case *True*
with ‹*InvariantQCharacterization* (*getConflictFlag state*) (*getQ state*) (*getF state*) (*getM state*)›
‹¬ *getConflictFlag state*›
‹*c el* (*getF ?state′*)›
have *l el* (*getQ state*)
using *assms*
using *assertLiteralEffect*[*of state literal decision*]
unfolding *InvariantQCharacterization-def*
by *auto*
have *isPrefix* (*getQ state*) (*getQ ?state′*)
using *assms*
using *assertLiteralEffect*[*of state literal decision*]
by *simp*
then obtain Q'
where (*getQ state*) @ Q' = (*getQ ?state′*)
unfolding *isPrefix-def*
by *auto*
have *l el* (*getQ ?state′*)
using ‹*l el* (*getQ state*)›
‹(*getQ state*) @ Q' = (*getQ ?state′*)›[*THEN sym*]
by *simp*
moreover
have $l \neq$ *literal*
using ‹*isUnitClause c l* (*elements* (*getM ?state′*))›
using *assms*
using *assertLiteralEffect*[*of state literal decision*]
unfolding *isUnitClause-def*
by *simp*
ultimately
show *?thesis*
by *auto*
next
case *False*

thus *?thesis*
proof (*cases ci* ∈ *set* (*getWatchList ?state* (*opposite literal*)))
case *True*
with ∗∗
‹*isUnitClause c l* (*elements* (*getM ?state′*))›

‹*c* = (*nth* (*getF ?state'*) *ci*)›
have *l* ∈ *set* (*getQ ?state'*)
using *assms*
using *assertLiteralEffect*[*of state literal decision*]
by *simp*
moreover
have *l* ≠ *literal*
using ‹*isUnitClause c l* (*elements* (*getM ?state'*))›
unfolding *isUnitClause-def*
using *assms*
using *assertLiteralEffect*[*of state literal decision*]
by *simp*
ultimately
show *?thesis*
by *simp*
next
case *False*
with ‹*InvariantWatchListsCharacterization* (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)›
have *w1* ≠ *opposite literal w2* ≠ *opposite literal*
using ‹*getWatch1 state ci* = *Some w1*› **and** ‹*getWatch2 state ci* = *Some w2*›
unfolding *InvariantWatchListsCharacterization-def*
by *auto*
have *literalFalse w1* (*elements* (*getM state*)) ∨ *literalFalse w2* (*elements* (*getM state*))
proof−
{
assume ¬ *?thesis*
hence ¬ *literalFalse w1* (*elements* (*getM ?state'*)) ¬ *literalFalse w2* (*elements* (*getM ?state'*))
using ‹*w1* ≠ *opposite literal*› **and** ‹*w2* ≠ *opposite literal*›
using *assms*
using *assertLiteralEffect*[*of state literal decision*]
by *auto*
with ‹*w1* ≠ *w2*› ‹*w1 el c*› ‹*w2 el c*›
have ¬ *isUnitClause c l* (*elements* (*getM ?state'*))
unfolding *isUnitClause-def*
by *auto*
}
with ‹*isUnitClause c l* (*elements* (*getM ?state'*))›
show *?thesis*
by *auto*
qed

with ‹*InvariantWatchCharacterization* (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*)›
have $: (∃ *l*. *l el c* ∧ *literalTrue l* (*elements* (*getM state*)))

∨
(∀ *l*. *l el c* ∧
l ≠ *w1* ∧ *l* ≠ *w2* ⟶ *literalFalse l* (*elements* (*getM state*)))

using ‹*ci* < *length* (*getF ?state'*)›
using ‹*c* = (*nth* (*getF ?state'*) *ci*)›
using ‹*getWatch1 state ci* = *Some w1*›[*THEN sym*] **and** ‹*getWatch2 state ci* = *Some w2*›[*THEN sym*]
using *assms*
using *assertLiteralEffect*[*of state literal decision*]
unfolding *InvariantWatchCharacterization-def*
unfolding *watchCharacterizationCondition-def*
by *auto*
thus *?thesis*
proof(*cases* ∀ *l*. *l el c* ∧ *l* ≠ *w1* ∧ *l* ≠ *w2* ⟶ *literalFalse l* (*elements* (*getM state*)))
case *True*
with ‹*isUnitClause c l* (*elements* (*getM ?state'*))›
have *literalFalse w1* (*elements* (*getM state*)) ⟶
¬ *literalFalse w2* (*elements* (*getM state*)) ∧ ¬ *literalTrue w2* (*elements* (*getM state*)) ∧ *l* = *w2*
literalFalse w2 (*elements* (*getM state*)) ⟶
¬ *literalFalse w1* (*elements* (*getM state*)) ∧ ¬ *literalTrue w1* (*elements* (*getM state*)) ∧ *l* = *w1*
unfolding *isUnitClause-def*
using *assms*
using *assertLiteralEffect*[*of state literal decision*]
by *auto*

with ‹*literalFalse w1* (*elements* (*getM state*)) ∨ *literalFalse w2* (*elements* (*getM state*))›
have (*literalFalse w1* (*elements* (*getM state*)) ∧ ¬ *literalFalse w2* (*elements* (*getM state*)) ∧ ¬ *literalTrue w2* (*elements* (*getM state*)) ∧ *l* = *w2*) ∨
(*literalFalse w2* (*elements* (*getM state*)) ∧ ¬ *literalFalse w1* (*elements* (*getM state*)) ∧ ¬ *literalTrue w1* (*elements* (*getM state*)) ∧ *l* = *w1*)
by *blast*
hence *isUnitClause c l* (*elements* (*getM state*))
using ‹*w1 el c*› ‹*w2 el c*› *True*
unfolding *isUnitClause-def*
by *auto*
thus *?thesis*
using ‹¬ *isUnitClause c l* (*elements* (*getM state*))›
by *simp*
next
case *False*
then obtain *l'*::*Literal* **where**

l' *el c literalTrue* l' (*elements* (*getM state*))
using $
by *auto*
hence *literalTrue* l' (*elements* (*getM ?state'*))
using *assms*
using *assertLiteralEffect*[*of state literal decision*]
by *auto*

from ‹*InvariantConsistent* ((*getM state*) @ [(*literal*, *decision*)])›
‹l' *el c*› ‹*literalTrue* l' (*elements* (*getM ?state'*))›
show *?thesis*
using *containsTrueNotUnit*[*of* l' *c elements* (*getM ?state'*)]
using ‹*isUnitClause c l* (*elements* (*getM ?state'*))›
using *assms*
using *assertLiteralEffect*[*of state literal decision*]
unfolding *InvariantConsistent-def*
by *auto*
qed
qed
qed
qed
qed
}
thus *?thesis*
unfolding *InvariantQCharacterization-def*
by *simp*
qed

lemma *AssertLiteralStartQIreleveant*:
fixes *literal* :: *Literal* **and** *Wl* :: *nat list* **and** *newWl* :: *nat list* **and** *state* :: *State*
assumes
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
and
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*)
shows
let state' = (*assertLiteral literal decision* (*state*(| *getQ* := *Q'* |))) *in*
let state'' = (*assertLiteral literal decision* (*state*(| *getQ* := *Q''* |))) *in*
(*getM state'*) = (*getM state''*) ∧
(*getF state'*) = (*getF state''*) ∧
(*getSATFlag state'*) = (*getSATFlag state''*) ∧
(*getConflictFlag state'*) = (*getConflictFlag state''*)

using *assms*
unfolding *assertLiteral-def*
unfolding *notifyWatches-def*
unfolding *InvariantWatchListsContainOnlyClausesFromF-def*

using *notifyWatchesStartQIreleveant*[*of*
state⦇ *getQ* := *Q′*, *getM* := *getM state* @ [(*literal*, *decision*)] ⦈
getWatchList (*state*⦇*getM* := *getM state* @ [(*literal*, *decision*)]⦈) (*opposite literal*)
state⦇ *getQ* := *Q″*, *getM* := *getM state* @ [(*literal*, *decision*)] ⦈
opposite literal []]
by (*simp add*: *Let-def*)

lemma *assertedLiteralIsNotUnit*:
assumes
InvariantConsistent ((*getM state*) @ [(*literal*, *decision*)])
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*)
shows
let state′ = *assertLiteral literal decision state in*
¬ *literal* ∈ (*set* (*getQ state′*) − *set*(*getQ state*))
proof−
{
let *?state* = *state*⦇*getM* := *getM state* @ [(*literal*, *decision*)]⦈
let *?state′* = *assertLiteral literal decision state*

assume ¬ *?thesis*

have ∗:∀ *l*. *l* ∈ *set* (*getQ ?state′*) − *set* (*getQ ?state*) ⟶
(∃ *clause*. *clause el* (*getF ?state*) ∧ *isUnitClause clause l* (*elements* (*getM ?state*)))
using *NotifyWatchesLoopQEffect*[*of ?state getM state opposite literal decision getWatchList ?state* (*opposite literal*) []]
using *assms*
unfolding *InvariantWatchListsUniq-def*
unfolding *InvariantWatchListsCharacterization-def*
unfolding *InvariantWatchListsContainOnlyClausesFromF-def*
unfolding *InvariantWatchCharacterization-def*
unfolding *assertLiteral-def*
unfolding *notifyWatches-def*
by (*auto simp add*: *Let-def*)
with ‹¬ *?thesis*›
obtain *clause*
where *isUnitClause clause literal* (*elements* (*getM ?state*))
by (*auto simp add*: *Let-def*)

hence *False*
unfolding *isUnitClause-def*
by *simp*
}
thus *?thesis*
by *auto*
qed

lemma *InvariantQCharacterizationAfterAssertLiteralNotInQ*:
assumes
InvariantConsistent ((*getM state*) @ [(*literal*, *decision*)])
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*)
InvariantConflictFlagCharacterization (*getConflictFlag state*) (*getF state*) (*getM state*)
InvariantQCharacterization (*getConflictFlag state*) (*getQ state*) (*getF state*) (*getM state*)
¬ *literal el* (*getQ state*)
shows
let state′ = (*assertLiteral literal decision state*) *in*
InvariantQCharacterization (*getConflictFlag state′*) (*getQ state′*) (*getF state′*) (*getM state′*)
proof−
let *?state′* = *assertLiteral literal decision state*
have *InvariantQCharacterization* (*getConflictFlag ?state′*) (*removeAll literal* (*getQ ?state′*)) (*getF ?state′*) (*getM ?state′*)
using *assms*
using *InvariantQCharacterizationAfterAssertLiteral*
by (*simp add*: *Let-def*)
moreover
have ¬ *literal el* (*getQ ?state′*)
using *assms*
using *assertedLiteralIsNotUnit*[*of state literal decision*]
by (*simp add*: *Let-def*)
hence *removeAll literal* (*getQ ?state′*) = *getQ ?state′*
using *removeAll-id*[*of literal getQ ?state′*]
by *simp*
ultimately
show *?thesis*
by (*simp add*: *Let-def*)

qed

lemma *InvariantUniqQAfterAssertLiteral*:
assumes
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantUniqQ (*getQ state*)
shows
let state′ = assertLiteral literal decision state in
InvariantUniqQ (*getQ state′*)
using *assms*
using *InvariantUniqQAfterNotifyWatchesLoop*[*of state*⦇*getM* := *getM state* @ [(*literal*, *decision*)]⦈
getWatchList (*state*⦇*getM* := *getM state* @ [(*literal*, *decision*)]⦈) (*opposite literal*)
opposite literal []]
unfolding *assertLiteral-def*
unfolding *notifyWatches-def*
unfolding *InvariantWatchListsContainOnlyClausesFromF-def*
by (*auto simp add*: *Let-def*)

lemma *InvariantsNoDecisionsWhenConflictNorUnitAfterAssertLiteral*:
assumes
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantConflictFlagCharacterization (*getConflictFlag state*) (*getF state*) (*getM state*)
InvariantQCharacterization (*getConflictFlag state*) (*getQ state*) (*getF state*) (*getM state*)
InvariantNoDecisionsWhenConflict (*getF state*) (*getM state*) (*currentLevel* (*getM state*))
InvariantNoDecisionsWhenUnit (*getF state*) (*getM state*) (*currentLevel* (*getM state*))
decision $\longrightarrow$ ¬ (*getConflictFlag state*) ∧ (*getQ state*) = []
shows
let state′ = assertLiteral literal decision state in
InvariantNoDecisionsWhenConflict (*getF state′*) (*getM state′*) (*currentLevel* (*getM state′*)) ∧
InvariantNoDecisionsWhenUnit (*getF state′*) (*getM state′*) (*currentLevel* (*getM state′*))
proof−
{
let *?state′* = *assertLiteral literal decision state*
fix *level*
assume *level* < *currentLevel* (*getM ?state′*)

have ¬ *formulaFalse* (*getF ?state′*) (*elements* (*prefixToLevel level* (*getM ?state′*))) ∧
¬ (∃ *clause literal*. *clause el* (*getF ?state′*) ∧
isUnitClause clause literal (*elements* (*prefixToLevel level* (*getM ?state′*))))
proof (*cases level* < *currentLevel* (*getM state*))
case *True*
hence *prefixToLevel level* (*getM ?state′*) = *prefixToLevel level* (*getM state*)
using *assms*
using *assertLiteralEffect*[*of state literal decision*]
by (*auto simp add*: *prefixToLevelAppend*)
moreover
have ¬ *formulaFalse* (*getF state*) (*elements* (*prefixToLevel level* (*getM state*)))
using ‹*InvariantNoDecisionsWhenConflict* (*getF state*) (*getM state*) (*currentLevel* (*getM state*))›
using ‹*level* < *currentLevel* (*getM state*)›
unfolding *InvariantNoDecisionsWhenConflict-def*
by *simp*
moreover
have ¬ (∃ *clause literal*. *clause el* (*getF state*) ∧
isUnitClause clause literal (*elements* (*prefixToLevel level* (*getM state*))))
using ‹*InvariantNoDecisionsWhenUnit* (*getF state*) (*getM state*) (*currentLevel* (*getM state*))›
using ‹*level* < *currentLevel* (*getM state*)›
unfolding *InvariantNoDecisionsWhenUnit-def*
by *simp*
ultimately
show *?thesis*
using *assms*
using *assertLiteralEffect*[*of state literal decision*]
by *auto*
next
case *False*
thus *?thesis*
proof (*cases decision*)
case *False*
hence *currentLevel* (*getM ?state′*) = *currentLevel* (*getM state*)
using *assms*
using *assertLiteralEffect*[*of state literal decision*]
unfolding *currentLevel-def*
by (*auto simp add*: *markedElementsAppend*)
thus *?thesis*
using ‹¬ (*level* < *currentLevel* (*getM state*))›
using ‹*level* < *currentLevel* (*getM ?state′*)›
by *simp*
next

case *True*
hence *currentLevel* (*getM ?state′*) = *currentLevel* (*getM state*) + *1*
using *assms*
using *assertLiteralEffect*[*of state literal decision*]
unfolding *currentLevel-def*
by (*auto simp add*: *markedElementsAppend*)
hence *level* = *currentLevel* (*getM state*)
using ‹¬ (*level* < *currentLevel* (*getM state*))›
using ‹*level* < *currentLevel* (*getM ?state′*)›
by *simp*
hence *prefixToLevel level* (*getM ?state′*) = (*getM state*)
using ‹*decision*›
using *assms*
using *assertLiteralEffect*[*of state literal decision*]
using *prefixToLevelAppend*[*of currentLevel* (*getM state*) *getM state* [(*literal*, *True*)]]
by *auto*
thus *?thesis*
using ‹*decision*›
using ‹*decision* ⟶ ¬ (*getConflictFlag state*) ∧ (*getQ state*) = []›
using ‹*InvariantConflictFlagCharacterization* (*getConflictFlag state*) (*getF state*) (*getM state*)›
using ‹*InvariantQCharacterization* (*getConflictFlag state*) (*getQ state*) (*getF state*) (*getM state*)›
unfolding *InvariantConflictFlagCharacterization-def*
unfolding *InvariantQCharacterization-def*
using *assms*
using *assertLiteralEffect*[*of state literal decision*]
by *simp*
qed
qed
} **thus** *?thesis*
unfolding *InvariantNoDecisionsWhenConflict-def*
unfolding *InvariantNoDecisionsWhenUnit-def*
by *auto*
qed

lemma *InvariantVarsQAfterAssertLiteral*:
assumes
InvariantConsistent ((*getM state*) @ [(*literal*, *decision*)])
InvariantUniq ((*getM state*) @ [(*literal*, *decision*)])
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*)
InvariantWatchListsUniq (*getWatchList state*)
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1*

state) (*getWatch2 state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*)
InvariantVarsQ (*getQ state*) *F0 Vbl*
InvariantVarsF (*getF state*) *F0 Vbl*
shows
let state′ = assertLiteral literal decision state in
InvariantVarsQ (*getQ state′*) *F0 Vbl*
proof−
let *?Q′* = {*ul*. ∃ *uc*. *uc el* (*getF state*) ∧
(*opposite literal*) *el uc* ∧ *isUnitClause uc ul* (*elements* (*getM state*) @ [*literal*])}
let *?state′* = *assertLiteral literal decision state*
have *vars ?Q′* ⊆ *vars* (*getF state*)
proof
fix *vbl*::*Variable*
assume *vbl* ∈ *vars ?Q′*
then obtain *ul*::*Literal*
where *ul* ∈ *?Q′ var ul* = *vbl*
by *auto*
then obtain *uc*::*Clause*
where *uc el* (*getF state*) *isUnitClause uc ul* (*elements* (*getM state*) @ [*literal*])
by *auto*
hence *vars uc* ⊆ *vars* (*getF state*) *var ul* ∈ *vars uc*
using *formulaContainsItsClausesVariables*[*of uc getF state*]
using *clauseContainsItsLiteralsVariable*[*of ul uc*]
unfolding *isUnitClause-def*
by *auto*
thus *vbl* ∈ *vars* (*getF state*)
using ‹*var ul* = *vbl*›
by *auto*
qed
thus *?thesis*
using *assms*
using *assertLiteralQEffect*[*of state literal decision*]
using *varsClauseVarsSet*[*of getQ ?state′*]
using *varsClauseVarsSet*[*of getQ state*]
unfolding *InvariantVarsQ-def*
unfolding *InvariantVarsF-def*
by (*auto simp add*: *Let-def*)
qed

end
theory *UnitPropagate*
imports *AssertLiteral*

begin

lemma *applyUnitPropagateEffect*:
assumes
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantQCharacterization (*getConflictFlag state*) (*getQ state*) (*getF state*) (*getM state*)

¬ (*getConflictFlag state*)
getQ state ≠ []
shows
let uLiteral = *hd* (*getQ state*) *in*
let state' = *applyUnitPropagate state in*
∃ *uClause*. *formulaEntailsClause* (*getF state*) *uClause* ∧
isUnitClause uClause uLiteral (*elements* (*getM state*)) ∧
(*getM state'*) = (*getM state*) @ [(*uLiteral*, *False*)]
proof−
let *?uLiteral* = *hd* (*getQ state*)
obtain *uClause*
where *uClause el* (*getF state*) *isUnitClause uClause ?uLiteral* (*elements* (*getM state*))
using *assms*
unfolding *InvariantQCharacterization-def*
by *force*
thus *?thesis*
using *assms*
using *assertLiteralEffect*[*of state ?uLiteral False*]
unfolding *applyUnitPropagate-def*
using *formulaEntailsItsClauses*[*of uClause getF state*]
by (*auto simp add*: *Let-def*)
qed

lemma *InvariantConsistentAfterApplyUnitPropagate*:
assumes
InvariantConsistent (*getM state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantQCharacterization (*getConflictFlag state*) (*getQ state*) (*getF state*) (*getM state*)
getQ state ≠ []
¬ (*getConflictFlag state*)

shows
let state′ = applyUnitPropagate state in
InvariantConsistent (getM state′)

proof−
let *?uLiteral = hd (getQ state)*
let *?state′ = applyUnitPropagate state*
obtain *uClause*
where *isUnitClause uClause ?uLiteral (elements (getM state))* **and**
(getM ?state′) = (getM state) @ [(?uLiteral, False)]
using *assms*
using *applyUnitPropagateEffect[of state]*
by *(auto simp add: Let-def)*
thus *?thesis*
using *assms*
using *InvariantConsistentAfterUnitPropagate[of getM state uClause ?uLiteral getM ?state′]*
by *(auto simp add: Let-def)*
qed

lemma *InvariantUniqAfterApplyUnitPropagate*:
assumes
InvariantUniq (getM state)
InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state) **and**
InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state) **and**
InvariantQCharacterization (getConflictFlag state) (getQ state) (getF state) (getM state)
getQ state ≠ []
¬ (getConflictFlag state)
shows
let state′ = applyUnitPropagate state in
InvariantUniq (getM state′)

proof−
let *?uLiteral = hd (getQ state)*
let *?state′ = applyUnitPropagate state*
obtain *uClause*
where *isUnitClause uClause ?uLiteral (elements (getM state))* **and**
(getM ?state′) = (getM state) @ [(?uLiteral, False)]
using *assms*
using *applyUnitPropagateEffect[of state]*
by *(auto simp add: Let-def)*
thus *?thesis*
using *assms*
using *InvariantUniqAfterUnitPropagate[of getM state uClause ?uLiteral getM ?state′]*
by *(auto simp add: Let-def)*

qed

lemma *InvariantWatchCharacterizationAfterApplyUnitPropagate*:
assumes
InvariantConsistent (*getM state*)
InvariantUniq (*getM state*)
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*)
InvariantQCharacterization (*getConflictFlag state*) (*getQ state*) (*getF state*) (*getM state*)
(*getQ state*) ≠ []
¬ (*getConflictFlag state*)
shows
let state′ = applyUnitPropagate state in
InvariantWatchCharacterization (*getF state′*) (*getWatch1 state′*) (*getWatch2 state′*) (*getM state′*)
proof−
let *?uLiteral = hd* (*getQ state*)
let *?state′ = assertLiteral ?uLiteral False state*
let *?state′′ = applyUnitPropagate state*
have *InvariantConsistent* (*getM ?state′*)
using *assms*
using *InvariantConsistentAfterApplyUnitPropagate*[*of state*]
unfolding *applyUnitPropagate-def*
by (*auto simp add*: *Let-def*)
moreover
have *InvariantUniq* (*getM ?state′*)
using *assms*
using *InvariantUniqAfterApplyUnitPropagate*[*of state*]
unfolding *applyUnitPropagate-def*
by (*auto simp add*: *Let-def*)
ultimately
show *?thesis*
using *assms*
using *InvariantWatchCharacterizationAfterAssertLiteral*[*of state ?uLiteral False*]
using *assertLiteralEffect*
unfolding *applyUnitPropagate-def*
by (*simp add*: *Let-def*)
qed

lemma *InvariantConflictFlagCharacterizationAfterApplyUnitPropagate*:
assumes
InvariantConsistent (*getM state*)
InvariantUniq (*getM state*)
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*)
InvariantQCharacterization (*getConflictFlag state*) (*getQ state*) (*getF state*) (*getM state*)
InvariantConflictFlagCharacterization (*getConflictFlag state*) (*getF state*) (*getM state*)
$\neg$ *getConflictFlag state*
getQ state $\neq$ []
shows
let state′ = (*applyUnitPropagate state*) *in*
InvariantConflictFlagCharacterization (*getConflictFlag state′*) (*getF state′*) (*getM state′*)
proof−
let *?uLiteral* = *hd* (*getQ state*)
let *?state′* = *assertLiteral ?uLiteral False state*
let *?state″* = *applyUnitPropagate state*
have *InvariantConsistent* (*getM ?state′*)
using *assms*
using *InvariantConsistentAfterApplyUnitPropagate*[*of state*]
unfolding *applyUnitPropagate-def*
by (*auto simp add*: *Let-def*)
moreover
have *InvariantUniq* (*getM ?state′*)
using *assms*
using *InvariantUniqAfterApplyUnitPropagate*[*of state*]
unfolding *applyUnitPropagate-def*
by (*auto simp add*: *Let-def*)
ultimately
show *?thesis*
using *assms*
using *InvariantConflictFlagCharacterizationAfterAssertLiteral*[*of state ?uLiteral False*]
using *assertLiteralEffect*
unfolding *applyUnitPropagate-def*
by (*simp add*: *Let-def*)

qed

lemma *InvariantConflictClauseCharacterizationAfterApplyUnitPropagate*:
assumes
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
and
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*)
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchListsUniq (*getWatchList state*)
¬ *getConflictFlag state*
shows
let state′ = applyUnitPropagate state in
InvariantConflictClauseCharacterization (*getConflictFlag state′*) (*getConflictClause state′*) (*getF state′*) (*getM state′*)
using *assms*
using *InvariantConflictClauseCharacterizationAfterAssertLiteral*[*of state hd* (*getQ state*) *False*]
unfolding *applyUnitPropagate-def*
unfolding *InvariantWatchesEl-def*
unfolding *InvariantWatchListsContainOnlyClausesFromF-def*
unfolding *InvariantWatchListsCharacterization-def*
unfolding *InvariantWatchListsUniq-def*
unfolding *InvariantConflictClauseCharacterization-def*
by (*simp add*: *Let-def*)

lemma *InvariantQCharacterizationAfterApplyUnitPropagate*:
assumes
InvariantConsistent (*getM state*)
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
and
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*)
InvariantConflictFlagCharacterization (*getConflictFlag state*) (*getF state*) (*getM state*)
InvariantQCharacterization (*getConflictFlag state*) (*getQ state*) (*getF state*) (*getM state*)
InvariantUniqQ (*getQ state*)
(*getQ state*) ≠ []

¬ (*getConflictFlag state*)
shows
let state'' = applyUnitPropagate state in
InvariantQCharacterization (*getConflictFlag state''*) (*getQ state''*) (*getF state''*) (*getM state''*)
proof−
let *?uLiteral* = *hd* (*getQ state*)
let *?state'* = *assertLiteral ?uLiteral False state*
let *?state''* = *applyUnitPropagate state*
have *InvariantConsistent* (*getM ?state'*)
using *assms*
using *InvariantConsistentAfterApplyUnitPropagate*[*of state*]
unfolding *applyUnitPropagate-def*
by (*auto simp add*: *Let-def*)
hence *InvariantQCharacterization* (*getConflictFlag ?state'*) (*removeAll ?uLiteral* (*getQ ?state'*)) (*getF ?state'*) (*getM ?state'*)
using *assms*
using *InvariantQCharacterizationAfterAssertLiteral*[*of state ?uLiteral False*]
using *assertLiteralEffect*[*of state ?uLiteral False*]
by (*simp add*: *Let-def*)
moreover
have *InvariantUniqQ* (*getQ ?state'*)
using *assms*
using *InvariantUniqQAfterAssertLiteral*[*of state ?uLiteral False*]
by (*simp add*: *Let-def*)

have *?uLiteral* = (*hd* (*getQ ?state'*))
proof−
obtain *s*
where (*getQ state*) @ *s* = *getQ ?state'*
using *assms*
using *assertLiteralEffect*[*of state ?uLiteral False*]
unfolding *isPrefix-def*
by *auto*
hence *getQ ?state'* = (*getQ state*) @ *s*
by (*rule sym*)
thus *?thesis*
using ‹*getQ state* ≠ []›
using *hd-append*[*of getQ state s*]
by *auto*
qed

hence *set* (*getQ ?state''*) = *set* (*removeAll ?uLiteral* (*getQ ?state'*))
using *assms*
using ‹*InvariantUniqQ* (*getQ ?state'*)›
unfolding *InvariantUniqQ-def*
using *uniqHeadTailSet*[*of getQ ?state'*]
unfolding *applyUnitPropagate-def*

by (*simp add*: *Let-def*)
ultimately
show *?thesis*
unfolding *InvariantQCharacterization-def*
unfolding *applyUnitPropagate-def*
by (*simp add*: *Let-def*)
qed

lemma *InvariantUniqQAfterApplyUnitPropagate*:
assumes
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*)
InvariantUniqQ (*getQ state*)
getQ state ≠ []
shows
let state'' = *applyUnitPropagate state in*
InvariantUniqQ (*getQ state''*)
proof−
let *?uLiteral* = *hd* (*getQ state*)
let *?state'* = *assertLiteral ?uLiteral False state*
let *?state''* = *applyUnitPropagate state*
have *InvariantUniqQ* (*getQ ?state'*)
using *assms*
using *InvariantUniqQAfterAssertLiteral*[*of state ?uLiteral False*]
by (*simp add*: *Let-def*)
moreover
obtain *s*
where *getQ state* @ *s* = *getQ ?state'*
using *assms*
using *assertLiteralEffect*[*of state ?uLiteral False*]
unfolding *isPrefix-def*
by *auto*
hence *getQ ?state'* = *getQ state* @ *s*
by (*rule sym*)
with ‹*getQ state* ≠ []›
have *getQ ?state'* ≠ []
by *simp*
ultimately
show *?thesis*
using ‹*getQ state* ≠ []›
unfolding *InvariantUniqQ-def*
unfolding *applyUnitPropagate-def*
using *hd-Cons-tl*[*of getQ ?state'*]
using *uniqAppendIff*[*of* [*hd* (*getQ ?state'*)] *tl* (*getQ ?state'*)]
by (*simp add*: *Let-def*)
qed

lemma *InvariantNoDecisionsWhenConflictNorUnitAfterUnitPropagate*:
assumes
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*)
InvariantConflictFlagCharacterization (*getConflictFlag state*) (*getF state*) (*getM state*)
InvariantQCharacterization (*getConflictFlag state*) (*getQ state*) (*getF state*) (*getM state*)
InvariantNoDecisionsWhenConflict (*getF state*) (*getM state*) (*currentLevel* (*getM state*))
InvariantNoDecisionsWhenUnit (*getF state*) (*getM state*) (*currentLevel* (*getM state*))
shows
let state′ = applyUnitPropagate state in
InvariantNoDecisionsWhenConflict (*getF state′*) (*getM state′*) (*currentLevel* (*getM state′*)) ∧
InvariantNoDecisionsWhenUnit (*getF state′*) (*getM state′*) (*currentLevel* (*getM state′*))
using *assms*
unfolding *applyUnitPropagate-def*
using *InvariantsNoDecisionsWhenConflictNorUnitAfterAssertLiteral*[*of state False hd* (*getQ state*)]
unfolding *InvariantNoDecisionsWhenConflict-def*
by (*simp add*: *Let-def*)

lemma *InvariantGetReasonIsReasonAfterApplyUnitPropagate*:
assumes
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantConflictFlagCharacterization (*getConflictFlag state*) (*getF state*) (*getM state*) **and**
InvariantUniqQ (*getQ state*) **and**
InvariantGetReasonIsReason (*getReason state*) (*getF state*) (*getM state*) (*set* (*getQ state*)) **and**
getQ state ≠ [] **and**
¬ *getConflictFlag state*
shows
let state′ = applyUnitPropagate state in
InvariantGetReasonIsReason (*getReason state′*) (*getF state′*) (*getM state′*) (*set* (*getQ state′*))
proof−
let *?state0* = *state* ⦇ *getM* := *getM state* @ [(*hd* (*getQ state*), *False*)]⦈

let *?state′* = *assertLiteral* (*hd* (*getQ state*)) *False state*
let *?state′′* = *applyUnitPropagate state*

have *InvariantGetReasonIsReason* (*getReason ?state0*) (*getF ?state0*) (*getM ?state0*) (*set* (*removeAll* (*hd* (*getQ ?state0*)) (*getQ ?state0*)))
proof−

{
fix *l*::*Literal*
assume ∗: *l el* (*elements* (*getM ?state0*)) ∧ ¬ *l el* (*decisions* (*getM ?state0*)) ∧ *elementLevel l* (*getM ?state0*) > *0*
hence ∃ *reason*. *getReason ?state0 l* = *Some reason* ∧ *0* ≤ *reason* ∧ *reason* < *length* (*getF ?state0*) ∧
isReason (*nth* (*getF ?state0*) *reason*) *l* (*elements* (*getM ?state0*))
proof (*cases l el* (*elements* (*getM state*)))
case *True*
from ∗
have ¬ *l el* (*decisions* (*getM state*))
by (*auto simp add*: *markedElementsAppend*)
from ∗
have *elementLevel l* (*getM state*) > *0*
using *elementLevelAppend*[*of l getM state* [(*hd* (*getQ state*), *False*)]]
using ‹*l el* (*elements* (*getM state*))›
by *simp*
show *?thesis*
using ‹*InvariantGetReasonIsReason* (*getReason state*) (*getF state*) (*getM state*) (*set* (*getQ state*))›
using ‹*l el* (*elements* (*getM state*))›
using ‹¬ *l el* (*decisions* (*getM state*))›
using ‹*elementLevel l* (*getM state*) > *0*›
unfolding *InvariantGetReasonIsReason-def*
by (*auto simp add*: *isReasonAppend*)
next
case *False*
with ∗
have *l* = *hd* (*getQ state*)
by *simp*

have *currentLevel* (*getM ?state0*) > *0*
using ∗
using *elementLevelLeqCurrentLevel*[*of l getM ?state0*]
by *auto*
hence *currentLevel* (*getM state*) > *0*
unfolding *currentLevel-def*
by (*simp add*: *markedElementsAppend*)
moreover
have *hd* (*getQ ?state0*) *el* (*getQ state*)

using ‹*getQ state* ≠ []›
by *simp*
ultimately
obtain *reason*
where *getReason state* (*hd* (*getQ state*)) = *Some reason 0* ≤ *reason* ∧ *reason* < *length* (*getF state*)
isUnitClause (*nth* (*getF state*) *reason*) (*hd* (*getQ state*)) (*elements* (*getM state*)) ∨
clauseFalse (*nth* (*getF state*) *reason*) (*elements* (*getM state*))
using ‹*InvariantGetReasonIsReason* (*getReason state*) (*getF state*) (*getM state*) (*set* (*getQ state*))›
unfolding *InvariantGetReasonIsReason-def*
by *auto*
hence *isUnitClause* (*nth* (*getF state*) *reason*) (*hd* (*getQ state*)) (*elements* (*getM state*))
using ‹¬ *getConflictFlag state*›
using ‹*InvariantConflictFlagCharacterization* (*getConflictFlag state*) (*getF state*) (*getM state*)›
unfolding *InvariantConflictFlagCharacterization-def*
using *nth-mem*[*of reason getF state*]
using *formulaFalseIffContainsFalseClause*[*of getF state elements* (*getM state*)]
by *simp*
thus *?thesis*
using ‹*getReason state* (*hd* (*getQ state*)) = *Some reason*› ‹*0* ≤ *reason* ∧ *reason* < *length* (*getF state*)›
using *isUnitClauseIsReason*[*of nth* (*getF state*) *reason hd* (*getQ state*) *elements* (*getM state*) [*hd* (*getQ state*)]]
using ‹*l* = *hd* (*getQ state*)›
by *simp*
qed
}
moreover
{
fix *literal*::*Literal*
assume *currentLevel* (*getM ?state0*) > *0*
hence *currentLevel* (*getM state*) > *0*
unfolding *currentLevel-def*
by (*simp add*: *markedElementsAppend*)

assume*literal el removeAll* (*hd* (*getQ ?state0*)) (*getQ ?state0*)
hence *literal* ≠ *hd* (*getQ state*) *literal el getQ state*
by *auto*

then obtain *reason*
where *getReason state literal* = *Some reason 0* ≤ *reason* ∧ *reason* < *length* (*getF state*) **and**
∗: *isUnitClause* (*nth* (*getF state*) *reason*) *literal* (*elements* (*getM state*)) ∨

clauseFalse (*nth* (*getF state*) *reason*) (*elements* (*getM state*))
using ‹*currentLevel* (*getM state*) > *0*›
using ‹*InvariantGetReasonIsReason* (*getReason state*) (*getF state*) (*getM state*) (*set* (*getQ state*))›
unfolding *InvariantGetReasonIsReason-def*
by *auto*
hence ∃ *reason*. *getReason ?state0 literal* = *Some reason* ∧ *0* ≤ *reason* ∧ *reason* < *length* (*getF ?state0*) ∧
(*isUnitClause* (*nth* (*getF ?state0*) *reason*) *literal* (*elements* (*getM ?state0*)) ∨
clauseFalse (*nth* (*getF ?state0*) *reason*) (*elements* (*getM ?state0*)))
proof (*cases isUnitClause* (*nth* (*getF state*) *reason*) *literal* (*elements* (*getM state*)))
case *True*
show *?thesis*
proof (*cases opposite literal* = *hd* (*getQ state*))
case *True*
thus *?thesis*
using ‹*isUnitClause* (*nth* (*getF state*) *reason*) *literal* (*elements* (*getM state*))›
using ‹*getReason state literal* = *Some reason*›
using ‹*literal* ≠ *hd* (*getQ state*)›
using ‹*0* ≤ *reason* ∧ *reason* < *length* (*getF state*)›
unfolding *isUnitClause-def*
by (*auto simp add*: *clauseFalseIffAllLiteralsAreFalse*)
next
case *False*
thus *?thesis*
using ‹*isUnitClause* (*nth* (*getF state*) *reason*) *literal* (*elements* (*getM state*))›
using ‹*getReason state literal* = *Some reason*›
using ‹*literal* ≠ *hd* (*getQ state*)›
using ‹*0* ≤ *reason* ∧ *reason* < *length* (*getF state*)›
unfolding *isUnitClause-def*
by *auto*
qed
next
case *False*
with ∗
have *clauseFalse* (*nth* (*getF state*) *reason*) (*elements* (*getM state*))
by *simp*
thus *?thesis*
using ‹*getReason state literal* = *Some reason*›
using ‹*0* ≤ *reason* ∧ *reason* < *length* (*getF state*)›
using *clauseFalseAppendValuation*[*of nth* (*getF state*) *reason elements* (*getM state*) [*hd* (*getQ state*)]]
by *auto*

qed
}
ultimately
show *?thesis*
unfolding *InvariantGetReasonIsReason-def*
by *auto*
qed

hence *InvariantGetReasonIsReason* (*getReason ?state′*) (*getF ?state′*) (*getM ?state′*) (*set* (*removeAll* (*hd* (*getQ state*)) (*getQ state*)) ∪ (*set* (*getQ ?state′*) − *set* (*getQ state*)))
using *assms*
unfolding *assertLiteral-def*
unfolding *notifyWatches-def*
using *InvariantGetReasonIsReasonAfterNotifyWatches*[*of* *?state0* *getWatchList ?state0* (*opposite* (*hd* (*getQ state*))) *opposite* (*hd* (*getQ state*)) *getM state False*
set (*removeAll* (*hd* (*getQ ?state0*)) (*getQ ?state0*)) []]
unfolding *InvariantWatchListsContainOnlyClausesFromF-def*
unfolding *InvariantWatchListsCharacterization-def*
unfolding *InvariantWatchListsUniq-def*
by (*auto simp add*: *Let-def*)

obtain *s*
where *getQ state* @ *s* = *getQ ?state′*
using *assms*
using *assertLiteralEffect*[*of state hd* (*getQ state*) *False*]
unfolding *isPrefix-def*
by *auto*
hence *getQ ?state′* = *getQ state* @ *s*
by *simp*
hence *hd* (*getQ ?state′*) = *hd* (*getQ state*)
using *hd-append2*[*of getQ state s*]
using ‹*getQ state* ≠ []›
by *simp*

have *set* (*removeAll* (*hd* (*getQ state*)) (*getQ state*)) ∪ (*set* (*getQ ?state′*) − *set* (*getQ state*)) =
set (*removeAll* (*hd* (*getQ state*)) (*getQ ?state′*))
using ‹*getQ ?state′* = *getQ state* @ *s*›
using ‹*getQ state* ≠ []›
by *auto*

have *uniq* (*getQ ?state′*)
using *assms*
using *InvariantUniqQAfterAssertLiteral*[*of state hd* (*getQ state*) *False*]
unfolding *InvariantUniqQ-def*
by (*simp add*: *Let-def*)

have *set* (*getQ ?state''*) = *set* (*removeAll* (*hd* (*getQ state*)) (*getQ ?state'*))
using ‹*uniq* (*getQ ?state'*)›
using ‹*hd* (*getQ ?state'*) = *hd* (*getQ state*)›
using *uniqHeadTailSet*[*of getQ ?state'*]
unfolding *applyUnitPropagate-def*
by (*simp add*: *Let-def*)

thus *?thesis*
using ‹*InvariantGetReasonIsReason* (*getReason ?state'*) (*getF ?state'*) (*getM ?state'*) (*set* (*removeAll* (*hd* (*getQ state*)) (*getQ state*)) ∪ (*set* (*getQ ?state'*) − *set* (*getQ state*)))›
using ‹*set* (*getQ ?state''*) = *set* (*removeAll* (*hd* (*getQ state*)) (*getQ ?state'*))›
using ‹*set* (*removeAll* (*hd* (*getQ state*)) (*getQ state*)) ∪ (*set* (*getQ ?state'*) − *set* (*getQ state*)) =
set (*removeAll* (*hd* (*getQ state*)) (*getQ ?state'*))›
unfolding *applyUnitPropagate-def*
by (*simp add*: *Let-def*)
qed

lemma *InvariantEquivalentZLAfterApplyUnitPropagate*:
assumes
InvariantEquivalentZL (*getF state*) (*getM state*) *Phi*
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantQCharacterization (*getConflictFlag state*) (*getQ state*) (*getF state*) (*getM state*)

¬ (*getConflictFlag state*)
getQ state ≠ []
shows
let state' = *applyUnitPropagate state in*
InvariantEquivalentZL (*getF state'*) (*getM state'*) *Phi*

proof−
let *?uLiteral* = *hd* (*getQ state*)
let *?state'* = *applyUnitPropagate state*
let *?FM* = *getF state* @ *val2form* (*elements* (*prefixToLevel 0* (*getM state*)))
let *?FM'* = *getF ?state'* @ *val2form* (*elements* (*prefixToLevel 0* (*getM ?state'*)))

obtain *uClause*
where *formulaEntailsClause* (*getF state*) *uClause* **and**

isUnitClause uClause ?uLiteral (*elements* (*getM state*)) **and**
(*getM ?state′*) = (*getM state*) @ [(*?uLiteral*, *False*)]
(*getF ?state′*) = (*getF state*)
using *assms*
using *applyUnitPropagateEffect*[*of state*]
unfolding *applyUnitPropagate-def*
using *assertLiteralEffect*
by (*auto simp add*: *Let-def*)
note ∗ = *this*

show *?thesis*
proof (*cases currentLevel* (*getM state*) = *0*)
case *True*
hence *getM state* = *prefixToLevel 0* (*getM state*)
by (*rule currentLevelZeroTrailEqualsItsPrefixToLevelZero*)

have *?FM′* = *?FM* @ [[*?uLiteral*]]
using ∗
using ‹(*getM ?state′*) = (*getM state*) @ [(*?uLiteral*, *False*)]›
using *prefixToLevelAppend*[*of 0 getM state* [(*?uLiteral*, *False*)]]
using ‹*currentLevel* (*getM state*) = *0*›
using ‹*getM state* = *prefixToLevel 0* (*getM state*)›
by (*auto simp add*: *val2formAppend*)

have *formulaEntailsLiteral ?FM ?uLiteral*
using ∗
using *unitLiteralIsEntailed* [*of uClause ?uLiteral elements* (*getM state*) (*getF state*)]
using ‹*InvariantEquivalentZL* (*getF state*) (*getM state*) *Phi*›
using ‹*getM state* = *prefixToLevel 0* (*getM state*)›
unfolding *InvariantEquivalentZL-def*
by *simp*
hence *formulaEntailsClause ?FM* [*?uLiteral*]
unfolding *formulaEntailsLiteral-def*
unfolding *formulaEntailsClause-def*
by (*auto simp add*: *clauseTrueIffContainsTrueLiteral*)

show *?thesis*
using ‹*InvariantEquivalentZL* (*getF state*) (*getM state*) *Phi*›
using ‹*?FM′* = *?FM* @ [[*?uLiteral*]]›
using ‹*formulaEntailsClause ?FM* [*?uLiteral*]›
unfolding *InvariantEquivalentZL-def*
using *extendEquivalentFormulaWithEntailedClause*[*of Phi ?FM* [*?uLiteral*]]
by (*simp add*: *equivalentFormulaeSymmetry*)
next
case *False*
hence *?FM* = *?FM′*

using ∗
using *prefixToLevelAppend*[*of 0 getM state* [(*?uLiteral*, *False*)]]
by (*simp add*: *Let-def*)
thus *?thesis*
using ‹*InvariantEquivalentZL* (*getF state*) (*getM state*) *Phi*›
unfolding *InvariantEquivalentZL-def*
by (*simp add*: *Let-def*)
qed
qed

lemma *InvariantVarsQTl*:
assumes
InvariantVarsQ Q F0 Vbl
$Q \neq []$
shows
InvariantVarsQ (*tl Q*) *F0 Vbl*
proof−
have *InvariantVarsQ* ((*hd Q*) # (*tl Q*)) *F0 Vbl*
using *assms*
by *simp*
hence {*var* (*hd Q*)} ∪ *vars* (*tl Q*) ⊆ *vars F0* ∪ *Vbl*
unfolding *InvariantVarsQ-def*
by *simp*
thus *?thesis*
unfolding *InvariantVarsQ-def*
by *simp*
qed

lemma *InvariantsVarsAfterApplyUnitPropagate*:
assumes
InvariantConsistent (*getM state*)
InvariantUniq (*getM state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
and
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*)
(*getF state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1*
state) (*getWatch2 state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2*
state) **and**
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2*
state) (*getM state*) **and**
InvariantQCharacterization False (*getQ state*) (*getF state*) (*getM*
state) **and**
getQ state ≠ []
¬ *getConflictFlag state*
InvariantVarsM (*getM state*) *F0 Vbl* **and**

InvariantVarsQ (*getQ state*) *F0 Vbl* **and**
InvariantVarsF (*getF state*) *F0 Vbl*
shows
let state′ = applyUnitPropagate state in
InvariantVarsM (*getM state′*) *F0 Vbl* ∧
InvariantVarsQ (*getQ state′*) *F0 Vbl*
proof−
let *?state′* = *assertLiteral* (*hd* (*getQ state*)) *False state*
let *?state″* = *applyUnitPropagate state*
have *InvariantVarsQ* (*getQ ?state′*) *F0 Vbl*
using *assms*
using *InvariantConsistentAfterApplyUnitPropagate*[*of state*]
using *InvariantUniqAfterApplyUnitPropagate*[*of state*]
using *InvariantVarsQAfterAssertLiteral*[*of state hd* (*getQ state*) *False F0 Vbl*]
using *assertLiteralEffect*[*of state hd* (*getQ state*) *False*]
unfolding *applyUnitPropagate-def*
by (*simp add*: *Let-def*)
moreover
have (*getQ ?state′*) ≠ []
using *assms*
using *assertLiteralEffect*[*of state hd* (*getQ state*) *False*]
using ‹*getQ state* ≠ []›
unfolding *isPrefix-def*
by *auto*
ultimately
have *InvariantVarsQ* (*getQ ?state″*) *F0 Vbl*
unfolding *applyUnitPropagate-def*
using *InvariantVarsQTl*[*of getQ ?state′ F0 Vbl*]
by (*simp add*: *Let-def*)
moreover
have *var* (*hd* (*getQ state*)) ∈ *vars F0* ∪ *Vbl*
using ‹*getQ state* ≠ []›
using ‹*InvariantVarsQ* (*getQ state*) *F0 Vbl*›
using *hd-in-set*[*of getQ state*]
using *clauseContainsItsLiteralsVariable*[*of hd* (*getQ state*) *getQ state*]
unfolding *InvariantVarsQ-def*
by *auto*
hence *InvariantVarsM* (*getM ?state″*) *F0 Vbl*
using *assms*
using *assertLiteralEffect*[*of state hd* (*getQ state*) *False*]
using *varsAppendValuation*[*of elements* (*getM state*) [*hd* (*getQ state*)]]
unfolding *applyUnitPropagate-def*
unfolding *InvariantVarsM-def*
by (*simp add*: *Let-def*)
ultimately
show *?thesis*

by (*simp add*: *Let-def*)
qed

definition *lexLessState* (*Vbl*::*Variable set*) == {(*state1*, *state2*).
(*getM state1*, *getM state2*) ∈ *lexLessRestricted Vbl*}

lemma *exhaustiveUnitPropagateTermination*:
fixes
state::*State* **and** *Vbl*::*Variable set*
assumes
InvariantUniq (*getM state*)
InvariantConsistent (*getM state*)
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*)
InvariantConflictFlagCharacterization (*getConflictFlag state*) (*getF state*) (*getM state*)
InvariantQCharacterization (*getConflictFlag state*) (*getQ state*) (*getF state*) (*getM state*)
InvariantUniqQ (*getQ state*)
InvariantVarsM (*getM state*) *F0 Vbl*
InvariantVarsQ (*getQ state*) *F0 Vbl*
InvariantVarsF (*getF state*) *F0 Vbl*
finite Vbl
shows
exhaustiveUnitPropagate-dom state
using *assms*
proof (*induct rule*: *wf-induct*[*of lexLessState* (*vars F0* ∪ *Vbl*)])
case *1*
show *?case*
unfolding *wf-eq-minimal*
proof−
show ∀ *Q* (*state*::*State*). *state* ∈ *Q* ⟶ (∃ *stateMin*∈*Q*. ∀ *state'*. (*state'*, *stateMin*) ∈ *lexLessState* (*vars F0* ∪ *Vbl*) ⟶ *state'* ∉ *Q*)
proof−
{

fix *Q* :: *State set* **and** *state* :: *State*
assume *state* ∈ *Q*
let *?Q1* = {*M*::*LiteralTrail*. ∃ *state*. *state* ∈ *Q* ∧ (*getM state*) = *M*}
from ‹*state* ∈ *Q*›
have *getM state* ∈ *?Q1*
by *auto*
have *wf* (*lexLessRestricted* (*vars F0* ∪ *Vbl*))
using ‹*finite Vbl*›
using *finiteVarsFormula*[*of F0*]
using *wfLexLessRestricted*[*of vars F0* ∪ *Vbl*]
by *simp*
with ‹*getM state* ∈ *?Q1*›
obtain *Mmin* **where** *Mmin* ∈ *?Q1* ∀ *M'*. (*M'*, *Mmin*) ∈ *lexLessRestricted* (*vars F0* ∪ *Vbl*) ⟶ *M'* ∉ *?Q1*
unfolding *wf-eq-minimal*
apply (*erule-tac x*=*?Q1* **in** *allE*)
apply (*erule-tac x*=*getM state* **in** *allE*)
by *auto*
from ‹*Mmin* ∈ *?Q1*› **obtain** *stateMin*
where *stateMin* ∈ *Q* (*getM stateMin*) = *Mmin*
by *auto*
have ∀ *state'*. (*state'*, *stateMin*) ∈ *lexLessState* (*vars F0* ∪ *Vbl*) ⟶ *state'* ∉ *Q*
proof
fix *state'*
show (*state'*, *stateMin*) ∈ *lexLessState* (*vars F0* ∪ *Vbl*) ⟶ *state'* ∉ *Q*
proof
assume (*state'*, *stateMin*) ∈ *lexLessState* (*vars F0* ∪ *Vbl*)
hence (*getM state'*, *getM stateMin*) ∈ *lexLessRestricted* (*vars F0* ∪ *Vbl*)
unfolding *lexLessState-def*
by *auto*
from ‹∀ *M'*. (*M'*, *Mmin*) ∈ *lexLessRestricted* (*vars F0* ∪ *Vbl*) ⟶ *M'* ∉ *?Q1*›
‹(*getM state'*, *getM stateMin*) ∈ *lexLessRestricted* (*vars F0* ∪ *Vbl*)› ‹*getM stateMin* = *Mmin*›
have *getM state'* ∉ *?Q1*
by *simp*
with ‹*getM stateMin* = *Mmin*›
show *state'* ∉ *Q*
by *auto*
qed
qed
with ‹*stateMin* ∈ *Q*›
have ∃ *stateMin* ∈ *Q*. (∀ *state'*. (*state'*, *stateMin*) ∈ *lexLessState* (*vars F0* ∪ *Vbl*) ⟶ *state'* ∉ *Q*)
by *auto*

```
      }
      thus ?thesis
        by auto
    qed
  qed
next
  case (2 state′)
  note ih = this
  show ?case
  proof (cases getQ state′ = [] ∨ getConflictFlag state′)
    case False
    let ?state′′ = applyUnitPropagate state′

   have InvariantWatchListsContainOnlyClausesFromF (getWatchList
?state′′) (getF ?state′′) and
      InvariantWatchListsUniq (getWatchList ?state′′) and
    InvariantWatchListsCharacterization (getWatchList ?state′′) (getWatch1
?state′′) (getWatch2 ?state′′)
    InvariantWatchesEl (getF ?state′′) (getWatch1 ?state′′) (getWatch2
?state′′) and
    InvariantWatchesDiffer (getF ?state′′) (getWatch1 ?state′′) (getWatch2
?state′′)
      using ih
     using WatchInvariantsAfterAssertLiteral[of state′ hd (getQ state′)
False]
      unfolding applyUnitPropagate-def
      by (auto simp add: Let-def)
    moreover
     have InvariantWatchCharacterization (getF ?state′′) (getWatch1
?state′′) (getWatch2 ?state′′) (getM ?state′′)
      using ih
    using InvariantWatchCharacterizationAfterApplyUnitPropagate[of
state′]
      unfolding InvariantQCharacterization-def
      using False
      by (simp add: Let-def)
    moreover
    have InvariantQCharacterization (getConflictFlag ?state′′) (getQ
?state′′) (getF ?state′′) (getM ?state′′)
      using ih
        using InvariantQCharacterizationAfterApplyUnitPropagate[of
state′]
      using False
      by (simp add: Let-def)
    moreover
   have InvariantConflictFlagCharacterization (getConflictFlag ?state′′)
(getF ?state′′) (getM ?state′′)
      using ih
     using InvariantConflictFlagCharacterizationAfterApplyUnitProp-
```

agate[*of state'*]
using *False*
by (*simp add*: *Let-def*)
moreover
have *InvariantUniqQ* (*getQ ?state''*)
using *ih*
using *InvariantUniqQAfterApplyUnitPropagate*[*of state'*]
using *False*
by (*simp add*: *Let-def*)
moreover
have *InvariantConsistent* (*getM ?state''*)
using *ih*
using *InvariantConsistentAfterApplyUnitPropagate*[*of state'*]
using *False*
by (*simp add*: *Let-def*)
moreover
have *InvariantUniq* (*getM ?state''*)
using *ih*
using *InvariantUniqAfterApplyUnitPropagate*[*of state'*]
using *False*
by (*simp add*: *Let-def*)
moreover
have *InvariantVarsM* (*getM ?state''*) *F0 Vbl InvariantVarsQ* (*getQ ?state''*) *F0 Vbl*
using *ih*
using ‹¬ (*getQ state'* = [] ∨ *getConflictFlag state'*)›
using *InvariantsVarsAfterApplyUnitPropagate*[*of state' F0 Vbl*]
by (*auto simp add*: *Let-def*)
moreover
have *InvariantVarsF* (*getF ?state''*) *F0 Vbl*
unfolding *applyUnitPropagate-def*
using *assertLiteralEffect*[*of state' hd* (*getQ state'*) *False*]
using *ih*
by (*simp add*: *Let-def*)
moreover
have (*?state''*, *state'*) ∈ *lexLessState* (*vars F0* ∪ *Vbl*)
proof−
have *getM ?state''* = *getM state'* @ [(*hd* (*getQ state'*), *False*)]
unfolding *applyUnitPropagate-def*
using *ih*
using *assertLiteralEffect*[*of state' hd* (*getQ state'*) *False*]
by (*simp add*: *Let-def*)
thus *?thesis*
unfolding *lexLessState-def*
unfolding *lexLessRestricted-def*
using *lexLessAppend*[*of* [(*hd* (*getQ state'*), *False*)] *getM state'*]
using ‹*InvariantConsistent* (*getM ?state''*)›
unfolding *InvariantConsistent-def*
using ‹*InvariantConsistent* (*getM state'*)›

unfolding *InvariantConsistent-def*
using ‹*InvariantUniq* (*getM ?state''*)›
unfolding *InvariantUniq-def*
using ‹*InvariantUniq* (*getM state'*)›
unfolding *InvariantUniq-def*
using ‹*InvariantVarsM* (*getM ?state''*) *F0 Vbl*›
using ‹*InvariantVarsM* (*getM state'*) *F0 Vbl*›
unfolding *InvariantVarsM-def*
by *simp*
qed
ultimately
have *exhaustiveUnitPropagate-dom ?state''*
using *ih*
by *auto*
thus *?thesis*
using *exhaustiveUnitPropagate-dom.intros*[*of state'*]
using *False*
by *simp*
next
case *True*
show *?thesis*
apply (*rule exhaustiveUnitPropagate-dom.intros*)
using *True*
by *simp*
qed
qed

lemma *exhaustiveUnitPropagatePreservedVariables*:
assumes
exhaustiveUnitPropagate-dom state
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
shows
let state' = exhaustiveUnitPropagate state in
(*getSATFlag state'*) = (*getSATFlag state*)
using *assms*
proof (*induct state rule*: *exhaustiveUnitPropagate-dom.induct*)
case (*step state'*)
note *ih* = *this*
show *?case*
proof (*cases* (*getConflictFlag state'*) ∨ (*getQ state'*) = [])
case *True*

with *exhaustiveUnitPropagate.simps[of state′]*
have *exhaustiveUnitPropagate state′ = state′*
by *simp*
thus *?thesis*
by (*simp only*: *Let-def*)
next
case *False*
let *?state′′ = applyUnitPropagate state′*

have *exhaustiveUnitPropagate state′ = exhaustiveUnitPropagate ?state′′*
using *exhaustiveUnitPropagate.simps[of state′]*
using *False*
by *simp*
moreover
have *InvariantWatchListsContainOnlyClausesFromF* (*getWatchList ?state′′*) (*getF ?state′′*) **and**
InvariantWatchListsUniq (*getWatchList ?state′′*) **and**
InvariantWatchListsCharacterization (*getWatchList ?state′′*) (*getWatch1 ?state′′*) (*getWatch2 ?state′′*)
InvariantWatchesEl (*getF ?state′′*) (*getWatch1 ?state′′*) (*getWatch2 ?state′′*) **and**
InvariantWatchesDiffer (*getF ?state′′*) (*getWatch1 ?state′′*) (*getWatch2 ?state′′*)
using *ih*
using *WatchInvariantsAfterAssertLiteral[of state′ hd* (*getQ state′*) *False]*
unfolding *applyUnitPropagate-def*
by (*auto simp add*: *Let-def*)
moreover
have *getSATFlag ?state′′ = getSATFlag state′*
unfolding *applyUnitPropagate-def*
using *assertLiteralEffect[of state′ hd* (*getQ state′*) *False]*
using *ih*
by (*simp add*: *Let-def*)
ultimately
show *?thesis*
using *ih*
using *False*
by (*simp add*: *Let-def*)
qed
qed

lemma *exhaustiveUnitPropagatePreservesCurrentLevel*:
assumes
exhaustiveUnitPropagate-dom state
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**

InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
and
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
shows
let state′ = exhaustiveUnitPropagate state in
currentLevel (*getM state′*) = *currentLevel* (*getM state*)
using *assms*
proof (*induct state rule*: *exhaustiveUnitPropagate-dom.induct*)
case (*step state′*)
note *ih* = *this*
show *?case*
proof (*cases* (*getConflictFlag state′*) ∨ (*getQ state′*) = [])
case *True*
with *exhaustiveUnitPropagate.simps*[*of state′*]
have *exhaustiveUnitPropagate state′* = *state′*
by *simp*
thus *?thesis*
by (*simp only*: *Let-def*)
next
case *False*
let *?state′′* = *applyUnitPropagate state′*

have *exhaustiveUnitPropagate state′* = *exhaustiveUnitPropagate ?state′′*
using *exhaustiveUnitPropagate.simps*[*of state′*]
using *False*
by *simp*
moreover
have *InvariantWatchListsContainOnlyClausesFromF* (*getWatchList ?state′′*) (*getF ?state′′*) **and**
InvariantWatchListsUniq (*getWatchList ?state′′*) **and**
InvariantWatchListsCharacterization (*getWatchList ?state′′*) (*getWatch1 ?state′′*) (*getWatch2 ?state′′*)
InvariantWatchesEl (*getF ?state′′*) (*getWatch1 ?state′′*) (*getWatch2 ?state′′*) **and**
InvariantWatchesDiffer (*getF ?state′′*) (*getWatch1 ?state′′*) (*getWatch2 ?state′′*)
using *ih*
using *WatchInvariantsAfterAssertLiteral*[*of state′ hd* (*getQ state′*) *False*]
unfolding *applyUnitPropagate-def*
by (*auto simp add*: *Let-def*)
moreover
have *currentLevel* (*getM state′*) = *currentLevel* (*getM ?state′′*)
unfolding *applyUnitPropagate-def*
using *assertLiteralEffect*[*of state′ hd* (*getQ state′*) *False*]

using *ih*
unfolding *currentLevel-def*
by (*simp add*: *Let-def markedElementsAppend*)
ultimately
show *?thesis*
using *ih*
using *False*
by (*simp add*: *Let-def*)
qed
qed

lemma *InvariantsAfterExhaustiveUnitPropagate*:
assumes
exhaustiveUnitPropagate-dom state
InvariantConsistent (*getM state*)
InvariantUniq (*getM state*)
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*)
InvariantConflictFlagCharacterization (*getConflictFlag state*) (*getF state*) (*getM state*)
InvariantQCharacterization (*getConflictFlag state*) (*getQ state*) (*getF state*) (*getM state*)
InvariantUniqQ (*getQ state*)
InvariantVarsQ (*getQ state*) *F0 Vbl*
InvariantVarsM (*getM state*) *F0 Vbl*
InvariantVarsF (*getF state*) *F0 Vbl*
shows
let state′ = exhaustiveUnitPropagate state in
InvariantConsistent (*getM state′*) ∧
InvariantUniq (*getM state′*) ∧
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state′*) (*getF state′*) ∧
InvariantWatchListsUniq (*getWatchList state′*) ∧
InvariantWatchListsCharacterization (*getWatchList state′*) (*getWatch1 state′*) (*getWatch2 state′*) ∧
InvariantWatchesEl (*getF state′*) (*getWatch1 state′*) (*getWatch2 state′*) ∧
InvariantWatchesDiffer (*getF state′*) (*getWatch1 state′*) (*getWatch2 state′*) ∧

InvariantWatchCharacterization (*getF state′*) (*getWatch1 state′*) (*getWatch2 state′*) (*getM state′*) ∧
InvariantConflictFlagCharacterization (*getConflictFlag state′*) (*getF state′*) (*getM state′*) ∧
InvariantQCharacterization (*getConflictFlag state′*) (*getQ state′*) (*getF state′*) (*getM state′*) ∧
InvariantUniqQ (*getQ state′*) ∧
InvariantVarsQ (*getQ state′*) *F0 Vbl* ∧
InvariantVarsM (*getM state′*) *F0 Vbl* ∧
InvariantVarsF (*getF state′*) *F0 Vbl*

using *assms*
proof (*induct state rule*: *exhaustiveUnitPropagate-dom.induct*)
case (*step state′*)
note *ih* = *this*
show *?case*
proof (*cases* (*getConflictFlag state′*) ∨ (*getQ state′*) = [])
case *True*
with *exhaustiveUnitPropagate.simps*[*of state′*]
have *exhaustiveUnitPropagate state′* = *state′*
by *simp*
thus *?thesis*
using *ih*
by (*auto simp only*: *Let-def*)
next
case *False*
let *?state′′* = *applyUnitPropagate state′*

have *exhaustiveUnitPropagate state′* = *exhaustiveUnitPropagate ?state′′*
using *exhaustiveUnitPropagate.simps*[*of state′*]
using *False*
by *simp*
moreover
have *InvariantWatchListsContainOnlyClausesFromF* (*getWatchList ?state′′*) (*getF ?state′′*) **and**
InvariantWatchListsUniq (*getWatchList ?state′′*) **and**
InvariantWatchListsCharacterization (*getWatchList ?state′′*) (*getWatch1 ?state′′*) (*getWatch2 ?state′′*)
InvariantWatchesEl (*getF ?state′′*) (*getWatch1 ?state′′*) (*getWatch2 ?state′′*) **and**
InvariantWatchesDiffer (*getF ?state′′*) (*getWatch1 ?state′′*) (*getWatch2 ?state′′*)
using *ih*
using *WatchInvariantsAfterAssertLiteral*[*of state′ hd* (*getQ state′*) *False*]
unfolding *applyUnitPropagate-def*
by (*auto simp add*: *Let-def*)
moreover

have *InvariantWatchCharacterization* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*) (*getM ?state''*)
using *ih*
using *InvariantWatchCharacterizationAfterApplyUnitPropagate*[*of state'*]
unfolding *InvariantQCharacterization-def*
using *False*
by (*simp add*: *Let-def*)
moreover
have *InvariantQCharacterization* (*getConflictFlag ?state''*) (*getQ ?state''*) (*getF ?state''*) (*getM ?state''*)
using *ih*
using *InvariantQCharacterizationAfterApplyUnitPropagate*[*of state'*]
using *False*
by (*simp add*: *Let-def*)
moreover
have *InvariantConflictFlagCharacterization* (*getConflictFlag ?state''*) (*getF ?state''*) (*getM ?state''*)
using *ih*
using *InvariantConflictFlagCharacterizationAfterApplyUnitPropagate*[*of state'*]
using *False*
by (*simp add*: *Let-def*)
moreover
have *InvariantUniqQ* (*getQ ?state''*)
using *ih*
using *InvariantUniqQAfterApplyUnitPropagate*[*of state'*]
using *False*
by (*simp add*: *Let-def*)
moreover
have *InvariantConsistent* (*getM ?state''*)
using *ih*
using *InvariantConsistentAfterApplyUnitPropagate*[*of state'*]
using *False*
by (*simp add*: *Let-def*)
moreover
have *InvariantUniq* (*getM ?state''*)
using *ih*
using *InvariantUniqAfterApplyUnitPropagate*[*of state'*]
using *False*
by (*simp add*: *Let-def*)
moreover
have *InvariantVarsM* (*getM ?state''*) *F0 Vbl InvariantVarsQ* (*getQ ?state''*) *F0 Vbl*
using *ih*
using ‹¬ (*getConflictFlag state'* ∨ *getQ state'* = [])›
using *InvariantsVarsAfterApplyUnitPropagate*[*of state' F0 Vbl*]
by (*auto simp add*: *Let-def*)

moreover
have *InvariantVarsF* (*getF ?state''*) *F0 Vbl*
unfolding *applyUnitPropagate-def*
using *assertLiteralEffect*[*of state' hd* (*getQ state'*) *False*]
using *ih*
by (*simp add*: *Let-def*)
ultimately
show *?thesis*
using *ih*
using *False*
by (*simp add*: *Let-def*)
qed
qed

lemma *InvariantConflictClauseCharacterizationAfterExhaustivePropagate*:
assumes
exhaustiveUnitPropagate-dom state
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantConflictClauseCharacterization (*getConflictFlag state*) (*getConflictClause state*) (*getF state*) (*getM state*)
shows
let state' = *exhaustiveUnitPropagate state in*
InvariantConflictClauseCharacterization (*getConflictFlag state'*) (*getConflictClause state'*) (*getF state'*) (*getM state'*)
using *assms*
proof (*induct state rule*: *exhaustiveUnitPropagate-dom.induct*)
case (*step state'*)
note *ih* = *this*
show *?case*
proof (*cases* (*getConflictFlag state'*) ∨ (*getQ state'*) = [])
case *True*
with *exhaustiveUnitPropagate.simps*[*of state'*]
have *exhaustiveUnitPropagate state'* = *state'*
by *simp*
thus *?thesis*
using *ih*
by (*auto simp only*: *Let-def*)
next
case *False*
let *?state''* = *applyUnitPropagate state'*

have *exhaustiveUnitPropagate state′* = *exhaustiveUnitPropagate ?state′′*
using *exhaustiveUnitPropagate.simps*[*of state′*]
using *False*
by *simp*
moreover
have *InvariantWatchListsContainOnlyClausesFromF* (*getWatchList ?state′′*) (*getF ?state′′*) **and**
InvariantWatchListsUniq (*getWatchList ?state′′*) **and**
InvariantWatchListsCharacterization (*getWatchList ?state′′*) (*getWatch1 ?state′′*) (*getWatch2 ?state′′*)
InvariantWatchesEl (*getF ?state′′*) (*getWatch1 ?state′′*) (*getWatch2 ?state′′*) **and**
InvariantWatchesDiffer (*getF ?state′′*) (*getWatch1 ?state′′*) (*getWatch2 ?state′′*)
using *ih*(*2*) *ih*(*3*) *ih*(*4*) *ih*(*5*) *ih*(*6*) *ih*(*7*)
using *WatchInvariantsAfterAssertLiteral*[*of state′ hd* (*getQ state′*) *False*]
unfolding *applyUnitPropagate-def*
by (*auto simp add*: *Let-def*)
moreover
have *InvariantConflictClauseCharacterization* (*getConflictFlag ?state′′*) (*getConflictClause ?state′′*) (*getF ?state′′*) (*getM ?state′′*)
using *ih*(*2*) *ih*(*3*) *ih*(*4*) *ih*(*5*) *ih*(*6*)
using ‹¬ (*getConflictFlag state′* ∨ *getQ state′* = [])›
using *InvariantConflictClauseCharacterizationAfterApplyUnitPropagate*[*of state′*]
by (*auto simp add*: *Let-def*)
ultimately
show *?thesis*
using *ih*(*1*) *ih*(*2*)
using *False*
by (*simp only*: *Let-def*) (*blast*)
qed
qed

lemma *InvariantsNoDecisionsWhenConflictNorUnitAfterExhaustivePropagate*:
assumes
exhaustiveUnitPropagate-dom state
InvariantConsistent (*getM state*)
InvariantUniq (*getM state*)
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**

InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*)
InvariantConflictFlagCharacterization (*getConflictFlag state*) (*getF state*) (*getM state*)
InvariantQCharacterization (*getConflictFlag state*) (*getQ state*) (*getF state*) (*getM state*)
InvariantUniqQ (*getQ state*)
InvariantNoDecisionsWhenConflict (*getF state*) (*getM state*) (*currentLevel* (*getM state*))
InvariantNoDecisionsWhenUnit (*getF state*) (*getM state*) (*currentLevel* (*getM state*))
shows
let state′ = exhaustiveUnitPropagate state in
InvariantNoDecisionsWhenConflict (*getF state′*) (*getM state′*) (*currentLevel* (*getM state′*)) ∧
InvariantNoDecisionsWhenUnit (*getF state′*) (*getM state′*) (*currentLevel* (*getM state′*))
using *assms*
proof (*induct state rule*: *exhaustiveUnitPropagate-dom.induct*)
case (*step state′*)
note *ih* = *this*
show *?case*
proof (*cases* (*getConflictFlag state′*) ∨ (*getQ state′*) = [])
case *True*
with *exhaustiveUnitPropagate.simps*[*of state′*]
have *exhaustiveUnitPropagate state′* = *state′*
by *simp*
thus *?thesis*
using *ih*
by (*auto simp only*: *Let-def*)
next
case *False*
let *?state″* = *applyUnitPropagate state′*

have *exhaustiveUnitPropagate state′* = *exhaustiveUnitPropagate ?state″*
using *exhaustiveUnitPropagate.simps*[*of state′*]
using *False*
by *simp*
moreover
have *InvariantWatchListsContainOnlyClausesFromF* (*getWatchList ?state″*) (*getF ?state″*) **and**
InvariantWatchListsUniq (*getWatchList ?state″*) **and**
InvariantWatchListsCharacterization (*getWatchList ?state″*) (*getWatch1 ?state″*) (*getWatch2 ?state″*)
InvariantWatchesEl (*getF ?state″*) (*getWatch1 ?state″*) (*getWatch2 ?state″*) **and**

InvariantWatchesDiffer (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)
using *ih*(*5*) *ih*(*6*) *ih*(*7*) *ih*(*8*) *ih*(*9*)
using *WatchInvariantsAfterAssertLiteral*[*of state' hd* (*getQ state'*) *False*]
unfolding *applyUnitPropagate-def*
by (*auto simp add*: *Let-def*)
moreover
have *InvariantWatchCharacterization* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*) (*getM ?state''*)
using *ih*
using *InvariantWatchCharacterizationAfterApplyUnitPropagate*[*of state'*]
unfolding *InvariantQCharacterization-def*
using *False*
by (*simp add*: *Let-def*)
moreover
have *InvariantQCharacterization* (*getConflictFlag ?state''*) (*getQ ?state''*) (*getF ?state''*) (*getM ?state''*)
using *ih*
using *InvariantQCharacterizationAfterApplyUnitPropagate*[*of state'*]
using *False*
by (*simp add*: *Let-def*)
moreover
have *InvariantConflictFlagCharacterization* (*getConflictFlag ?state''*) (*getF ?state''*) (*getM ?state''*)
using *ih*
using *InvariantConflictFlagCharacterizationAfterApplyUnitPropagate*[*of state'*]
using *False*
by (*simp add*: *Let-def*)
moreover
have *InvariantUniqQ* (*getQ ?state''*)
using *ih*
using *InvariantUniqQAfterApplyUnitPropagate*[*of state'*]
using *False*
by (*simp add*: *Let-def*)
moreover
have *InvariantConsistent* (*getM ?state''*)
using *ih*
using *InvariantConsistentAfterApplyUnitPropagate*[*of state'*]
using *False*
by (*simp add*: *Let-def*)
moreover
have *InvariantUniq* (*getM ?state''*)
using *ih*
using *InvariantUniqAfterApplyUnitPropagate*[*of state'*]
using *False*

by (*simp add*: *Let-def*)
moreover
have *InvariantNoDecisionsWhenUnit* (*getF ?state′′*) (*getM ?state′′*) (*currentLevel* (*getM ?state′′*))
InvariantNoDecisionsWhenConflict (*getF ?state′′*) (*getM ?state′′*) (*currentLevel* (*getM ?state′′*))
using *ih*(*5*) *ih*(*8*) *ih*(*11*) *ih*(*12*) *ih*(*14*) *ih*(*15*)
using *InvariantNoDecisionsWhenConflictNorUnitAfterUnitPropagate*[*of state′*]
by (*auto simp add*: *Let-def*)
ultimately
show *?thesis*
using *ih*(*1*) *ih*(*2*)
using *False*
by (*simp add*: *Let-def*)
qed
qed

lemma *InvariantGetReasonIsReasonAfterExhaustiveUnitPropagate*:
assumes
exhaustiveUnitPropagate-dom state
InvariantConsistent (*getM state*)
InvariantUniq (*getM state*)
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*)
InvariantConflictFlagCharacterization (*getConflictFlag state*) (*getF state*) (*getM state*)
InvariantQCharacterization (*getConflictFlag state*) (*getQ state*) (*getF state*) (*getM state*)
InvariantUniqQ (*getQ state*) **and**
InvariantGetReasonIsReason (*getReason state*) (*getF state*) (*getM state*) (*set* (*getQ state*))
shows
let state′ = *exhaustiveUnitPropagate state in*
InvariantGetReasonIsReason (*getReason state′*) (*getF state′*) (*getM state′*) (*set* (*getQ state′*))
using *assms*
proof (*induct state rule*: *exhaustiveUnitPropagate-dom.induct*)
case (*step state′*)

note *ih* = *this*
show *?case*
proof (*cases* (*getConflictFlag state'*) ∨ (*getQ state'*) = [])
case *True*
with *exhaustiveUnitPropagate.simps*[*of state'*]
have *exhaustiveUnitPropagate state'* = *state'*
by *simp*
thus *?thesis*
using *ih*
by (*auto simp only*: *Let-def*)
next
case *False*
let *?state''* = *applyUnitPropagate state'*

have *exhaustiveUnitPropagate state'* = *exhaustiveUnitPropagate ?state''*
using *exhaustiveUnitPropagate.simps*[*of state'*]
using *False*
by *simp*
moreover
have *InvariantWatchListsContainOnlyClausesFromF* (*getWatchList ?state''*) (*getF ?state''*) **and**
InvariantWatchListsUniq (*getWatchList ?state''*) **and**
InvariantWatchListsCharacterization (*getWatchList ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)
InvariantWatchesEl (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*) **and**
InvariantWatchesDiffer (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)
using *ih*
using *WatchInvariantsAfterAssertLiteral*[*of state' hd* (*getQ state'*) *False*]
unfolding *applyUnitPropagate-def*
by (*auto simp add*: *Let-def*)
moreover
have *InvariantWatchCharacterization* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*) (*getM ?state''*)
using *ih*
using *InvariantWatchCharacterizationAfterApplyUnitPropagate*[*of state'*]
unfolding *InvariantQCharacterization-def*
using *False*
by (*simp add*: *Let-def*)
moreover
have *InvariantQCharacterization* (*getConflictFlag ?state''*) (*getQ ?state''*) (*getF ?state''*) (*getM ?state''*)
using *ih*
using *InvariantQCharacterizationAfterApplyUnitPropagate*[*of state'*]

using *False*
by (*simp add*: *Let-def*)
moreover
have *InvariantConflictFlagCharacterization* (*getConflictFlag ?state''*) (*getF ?state''*) (*getM ?state''*)
using *ih*
using *InvariantConflictFlagCharacterizationAfterApplyUnitPropagate*[*of state'*]
using *False*
by (*simp add*: *Let-def*)
moreover
have *InvariantUniqQ* (*getQ ?state''*)
using *ih*
using *InvariantUniqQAfterApplyUnitPropagate*[*of state'*]
using *False*
by (*simp add*: *Let-def*)
moreover
have *InvariantConsistent* (*getM ?state''*)
using *ih*
using *InvariantConsistentAfterApplyUnitPropagate*[*of state'*]
using *False*
by (*simp add*: *Let-def*)
moreover
have *InvariantUniq* (*getM ?state''*)
using *ih*
using *InvariantUniqAfterApplyUnitPropagate*[*of state'*]
using *False*
by (*simp add*: *Let-def*)
moreover
have *InvariantGetReasonIsReason* (*getReason ?state''*) (*getF ?state''*) (*getM ?state''*) (*set* (*getQ ?state''*))
using *ih*
using *InvariantGetReasonIsReasonAfterApplyUnitPropagate*[*of state'*]
using *False*
by (*simp add*: *Let-def*)
ultimately
show *?thesis*
using *ih*
using *False*
by (*simp add*: *Let-def*)
qed
qed

lemma *InvariantEquivalentZLAfterExhaustiveUnitPropagate*:
assumes
exhaustiveUnitPropagate-dom state
InvariantConsistent (*getM state*)

InvariantUniq (*getM state*)
InvariantEquivalentZL (*getF state*) (*getM state*) *Phi*
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*)
InvariantConflictFlagCharacterization (*getConflictFlag state*) (*getF state*) (*getM state*)
InvariantQCharacterization (*getConflictFlag state*) (*getQ state*) (*getF state*) (*getM state*)
InvariantUniqQ (*getQ state*)
shows
let state′ = exhaustiveUnitPropagate state in
InvariantEquivalentZL (*getF state′*) (*getM state′*) *Phi*

using *assms*
proof (*induct state rule*: *exhaustiveUnitPropagate-dom.induct*)
case (*step state′*)
note *ih* = *this*
show *?case*
proof (*cases* (*getConflictFlag state′*) ∨ (*getQ state′*) = [])
case *True*
with *exhaustiveUnitPropagate.simps*[*of state′*]
have *exhaustiveUnitPropagate state′* = *state′*
by *simp*
thus *?thesis*
using *ih*
by (*simp only*: *Let-def*)
next
case *False*
let *?state″* = *applyUnitPropagate state′*

have *exhaustiveUnitPropagate state′* = *exhaustiveUnitPropagate ?state″*
using *exhaustiveUnitPropagate.simps*[*of state′*]
using *False*
by *simp*
moreover
have *InvariantWatchListsContainOnlyClausesFromF* (*getWatchList ?state″*) (*getF ?state″*) **and**
InvariantWatchListsUniq (*getWatchList ?state″*) **and**
InvariantWatchListsCharacterization (*getWatchList ?state″*) (*getWatch1*

?state'') (*getWatch2 ?state''*)
InvariantWatchesEl (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*) **and**
InvariantWatchesDiffer (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*)
using *ih*
using *WatchInvariantsAfterAssertLiteral*[*of state' hd* (*getQ state'*) *False*]
unfolding *applyUnitPropagate-def*
by (*auto simp add*: *Let-def*)
moreover
have *InvariantWatchCharacterization* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*) (*getM ?state''*)
using *ih*
using *InvariantWatchCharacterizationAfterApplyUnitPropagate*[*of state'*]
unfolding *InvariantQCharacterization-def*
using *False*
by (*simp add*: *Let-def*)
moreover
have *InvariantQCharacterization* (*getConflictFlag ?state''*) (*getQ ?state''*) (*getF ?state''*) (*getM ?state''*)
using *ih*
using *InvariantQCharacterizationAfterApplyUnitPropagate*[*of state'*]
using *False*
by (*simp add*: *Let-def*)
moreover
have *InvariantConflictFlagCharacterization* (*getConflictFlag ?state''*) (*getF ?state''*) (*getM ?state''*)
using *ih*
using *InvariantConflictFlagCharacterizationAfterApplyUnitPropagate*[*of state'*]
using *False*
by (*simp add*: *Let-def*)
moreover
have *InvariantUniqQ* (*getQ ?state''*)
using *ih*
using *InvariantUniqQAfterApplyUnitPropagate*[*of state'*]
using *False*
by (*simp add*: *Let-def*)
moreover
have *InvariantConsistent* (*getM ?state''*)
using *ih*
using *InvariantConsistentAfterApplyUnitPropagate*[*of state'*]
using *False*
by (*simp add*: *Let-def*)
moreover
have *InvariantUniq* (*getM ?state''*)

```
      using ih
      using InvariantUniqAfterApplyUnitPropagate[of state']
      using False
      by (simp add: Let-def)
    moreover
    have InvariantEquivalentZL (getF ?state'') (getM ?state'') Phi
      using ih
       using InvariantEquivalentZLAfterApplyUnitPropagate[of state'
Phi]
      using False
      by (simp add: Let-def)
    moreover
    have currentLevel (getM state') = currentLevel (getM ?state'')
      unfolding applyUnitPropagate-def
      using assertLiteralEffect[of state' hd (getQ state') False]
      using ih
      unfolding currentLevel-def
      by (simp add: Let-def markedElementsAppend)
    ultimately
    show ?thesis
      using ih
      using False
      by (auto simp only: Let-def)
  qed
qed

lemma conflictFlagOrQEmptyAfterExhaustiveUnitPropagate:
assumes
exhaustiveUnitPropagate-dom state
shows
let state' = exhaustiveUnitPropagate state in
    (getConflictFlag state') ∨ (getQ state' = [])
using assms
proof (induct state rule: exhaustiveUnitPropagate-dom.induct)
  case (step state')
  note ih = this
  show ?case
  proof (cases (getConflictFlag state') ∨ (getQ state') = [])
    case True
    with exhaustiveUnitPropagate.simps[of state']
    have exhaustiveUnitPropagate state' = state'
      by simp
    thus ?thesis
      using True
      by (simp only: Let-def)
  next
    case False
    let ?state'' = applyUnitPropagate state'
```

have *exhaustiveUnitPropagate state′* = *exhaustiveUnitPropagate ?state′′*
using *exhaustiveUnitPropagate.simps*[*of state′*]
using *False*
by *simp*
thus *?thesis*
using *ih*
using *False*
by (*simp add*: *Let-def*)
qed
qed

end
theory *Initialization*
imports *UnitPropagate*
begin

lemma *InvariantsAfterAddClause*:
fixes *state*::*State* **and** *clause* :: *Clause* **and** *Vbl* :: *Variable set*
assumes
InvariantConsistent (*getM state*)
InvariantUniq (*getM state*)
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*)
InvariantConflictFlagCharacterization (*getConflictFlag state*) (*getF state*) (*getM state*)
InvariantConflictClauseCharacterization (*getConflictFlag state*) (*getConflictClause state*) (*getF state*) (*getM state*)
InvariantQCharacterization (*getConflictFlag state*) (*getQ state*) (*getF state*) (*getM state*)
InvariantUniqQ (*getQ state*)
InvariantGetReasonIsReason (*getReason state*) (*getF state*) (*getM state*) (*set* (*getQ state*))
currentLevel (*getM state*) = *0*
(*getConflictFlag state*) ∨ (*getQ state*) = []

InvariantVarsM (*getM state*) *F0 Vbl*
InvariantVarsQ (*getQ state*) *F0 Vbl*
InvariantVarsF (*getF state*) *F0 Vbl*
finite Vbl
vars clause ⊆ *vars F0*
shows
let state′ = (*addClause clause state*) *in*
InvariantConsistent (*getM state′*) ∧
InvariantUniq (*getM state′*) ∧
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state′*) (*getF state′*) ∧
InvariantWatchListsUniq (*getWatchList state′*) ∧
InvariantWatchListsCharacterization (*getWatchList state′*) (*getWatch1 state′*) (*getWatch2 state′*) ∧
InvariantWatchesEl (*getF state′*) (*getWatch1 state′*) (*getWatch2 state′*) ∧
InvariantWatchesDiffer (*getF state′*) (*getWatch1 state′*) (*getWatch2 state′*) ∧
InvariantWatchCharacterization (*getF state′*) (*getWatch1 state′*) (*getWatch2 state′*) (*getM state′*) ∧
InvariantConflictFlagCharacterization (*getConflictFlag state′*) (*getF state′*) (*getM state′*) ∧
InvariantConflictClauseCharacterization (*getConflictFlag state′*) (*getConflictClause state′*) (*getF state′*) (*getM state′*) ∧
InvariantQCharacterization (*getConflictFlag state′*) (*getQ state′*) (*getF state′*) (*getM state′*) ∧
InvariantGetReasonIsReason (*getReason state′*) (*getF state′*) (*getM state′*) (*set* (*getQ state′*)) ∧
InvariantUniqQ (*getQ state′*) ∧
InvariantVarsQ (*getQ state′*) *F0 Vbl* ∧
InvariantVarsM (*getM state′*) *F0 Vbl* ∧
InvariantVarsF (*getF state′*) *F0 Vbl* ∧
currentLevel (*getM state′*) = *0* ∧
((*getConflictFlag state′*) ∨ (*getQ state′*) = [])

proof−
let *?clause′* = *remdups* (*removeFalseLiterals clause* (*elements* (*getM state*)))

have ∗: ∀ *l*. *l el ?clause′* ⟶ ¬ *literalFalse l* (*elements* (*getM state*))
unfolding *removeFalseLiterals-def*
by *auto*

have *vars ?clause′* ⊆ *vars clause*
using *varsSubsetValuation*[*of ?clause′ clause*]
unfolding *removeFalseLiterals-def*
by *auto*
hence *vars ?clause′* ⊆ *vars F0*
using ‹*vars clause* ⊆ *vars F0*›

```
    by simp

  show ?thesis
  proof (cases clauseTrue ?clause' (elements (getM state)))
    case True
    thus ?thesis
      using assms
      unfolding addClause-def
      by simp
  next
    case False
    show ?thesis
    proof (cases ?clause' = [])
      case True
      thus ?thesis
        using assms
        using ‹¬ clauseTrue ?clause' (elements (getM state))›
        unfolding addClause-def
        by simp
    next
      case False
      thus ?thesis
      proof (cases length ?clause' = 1)
        case True
        let ?state' = assertLiteral (hd ?clause') False state
        have addClause clause state = exhaustiveUnitPropagate ?state'
          using ‹¬ clauseTrue ?clause' (elements (getM state))›
          using ‹¬ ?clause' = []›
          using ‹length ?clause' = 1›
          unfolding addClause-def
          by (simp add: Let-def)
        moreover
        from ‹?clause' ≠ []›
        have hd ?clause' ∈ set ?clause'
          using hd-in-set[of ?clause']
          by simp
        with *
        have ¬ literalFalse (hd ?clause') (elements (getM state))
          by simp
          hence consistent (elements ((getM state) @ [(hd ?clause',
False)]))
          using assms
          unfolding InvariantConsistent-def
          using consistentAppendElement[of elements (getM state) hd
?clause']
          by simp
        hence consistent (elements (getM ?state'))
          using assms
          using assertLiteralEffect[of state hd ?clause' False]
```

by *simp*
moreover
from ‹¬ *clauseTrue ?clause′* (*elements* (*getM state*))›
have *uniq* (*elements* (*getM ?state′*))
using *assms*
using *assertLiteralEffect*[*of state hd ?clause′ False*]
using ‹*hd ?clause′* ∈ *set ?clause′*›
unfolding *InvariantUniq-def*
by (*simp add*: *uniqAppendIff clauseTrueIffContainsTrueLiteral*)
moreover
have *InvariantWatchListsContainOnlyClausesFromF* (*getWatchList ?state′*) (*getF ?state′*) **and**
InvariantWatchListsUniq (*getWatchList ?state′*) **and**
InvariantWatchListsCharacterization (*getWatchList ?state′*) (*getWatch1 ?state′*) (*getWatch2 ?state′*)
InvariantWatchesEl (*getF ?state′*) (*getWatch1 ?state′*) (*getWatch2 ?state′*) **and**
InvariantWatchesDiffer (*getF ?state′*) (*getWatch1 ?state′*) (*getWatch2 ?state′*)
using *assms*
using *WatchInvariantsAfterAssertLiteral*[*of state hd ?clause′ False*]
by (*auto simp add*: *Let-def*)
moreover
have *InvariantWatchCharacterization* (*getF ?state′*) (*getWatch1 ?state′*) (*getWatch2 ?state′*) (*getM ?state′*)
using *assms*
using *InvariantWatchCharacterizationAfterAssertLiteral*[*of state hd ?clause′ False*]
using ‹*uniq* (*elements* (*getM ?state′*))›
using ‹*consistent* (*elements* (*getM ?state′*))›
unfolding *InvariantConsistent-def*
unfolding *InvariantUniq-def*
using *assertLiteralEffect*[*of state hd ?clause′ False*]
by (*simp add*: *Let-def*)
moreover
have *InvariantConflictFlagCharacterization* (*getConflictFlag ?state′*) (*getF ?state′*) (*getM ?state′*)
using *assms*
using *InvariantConflictFlagCharacterizationAfterAssertLiteral*[*of state hd ?clause′ False*]
using ‹*consistent* (*elements* (*getM ?state′*))›
unfolding *InvariantConsistent-def*
using *assertLiteralEffect*[*of state hd ?clause′ False*]
by (*simp add*: *Let-def*)
moreover
have *InvariantConflictClauseCharacterization* (*getConflictFlag ?state′*) (*getConflictClause ?state′*) (*getF ?state′*) (*getM ?state′*)
using *assms*

using *InvariantConflictClauseCharacterizationAfterAssertLiteral*[*of state hd ?clause′ False*]
by (*simp add*: *Let-def*)
moreover
let *?state′′* = *?state′*⦇ *getM* := (*getM ?state′*) @ [(*hd ?clause′*, *False*)] ⦈
have *InvariantQCharacterization* (*getConflictFlag ?state′*) (*getQ ?state′*) (*getF ?state′*) (*getM ?state′*)
proof (*cases getConflictFlag state*)
case *True*
hence *getConflictFlag ?state′*
using *assms*
using *assertLiteralConflictFlagEffect*[*of state hd ?clause′ False*]
using ‹*uniq* (*elements* (*getM ?state′*))›
using ‹*consistent* (*elements* (*getM ?state′*))›
unfolding *InvariantConsistent-def*
unfolding *InvariantUniq-def*
using *assertLiteralEffect*[*of state hd ?clause′ False*]
by (*auto simp add*: *Let-def*)
thus *?thesis*
using *assms*
unfolding *InvariantQCharacterization-def*
by *simp*
next
case *False*
with ‹(*getConflictFlag state*) ∨ (*getQ state*) = []›
have *getQ state* = []
by *simp*
thus *?thesis*
using *InvariantQCharacterizationAfterAssertLiteralNotInQ*[*of state hd ?clause′ False*]
using *assms*
using ‹*uniq* (*elements* (*getM ?state′*))›
using ‹*consistent* (*elements* (*getM ?state′*))›
unfolding *InvariantConsistent-def*
unfolding *InvariantUniq-def*
using *assertLiteralEffect*[*of state hd ?clause′ False*]
by (*auto simp add*: *Let-def*)
qed
moreover
have *InvariantUniqQ* (*getQ ?state′*)
using *assms*
using *InvariantUniqQAfterAssertLiteral*[*of state hd ?clause′ False*]
by (*simp add*: *Let-def*)
moreover
have *currentLevel* (*getM ?state′*) = *0*
using *assms*

using ‹¬ *clauseTrue ?clause′* (*elements* (*getM state*))›
using ‹¬ *?clause′* = []›
using *assertLiteralEffect*[*of state hd ?clause′ False*]
unfolding *addClause-def*
unfolding *currentLevel-def*
by (*simp add*:*Let-def markedElementsAppend*)
moreover
hence *InvariantGetReasonIsReason* (*getReason ?state′*) (*getF ?state′*) (*getM ?state′*) (*set* (*getQ ?state′*))
unfolding *InvariantGetReasonIsReason-def*
using *elementLevelLeqCurrentLevel*[*of - getM ?state′*]
by *auto*
moreover
have *var* (*hd ?clause′*) ∈ *vars F0*
using ‹*?clause′* ≠ []›
using *hd-in-set*[*of ?clause′*]
using ‹*vars ?clause′* ⊆ *vars F0*›
using *clauseContainsItsLiteralsVariable*[*of hd ?clause′ ?clause′*]
by *auto*
hence *InvariantVarsQ* (*getQ ?state′*) *F0 Vbl*
InvariantVarsM (*getM ?state′*) *F0 Vbl*
InvariantVarsF (*getF ?state′*) *F0 Vbl*
using ‹*InvariantWatchListsContainOnlyClausesFromF* (*getWatchList state*) (*getF state*)›
using ‹*InvariantWatchesEl* (*getF state*) (*getWatch1 state*) (*getWatch2 state*)›
using ‹*InvariantWatchListsUniq* (*getWatchList state*)›
using ‹*InvariantWatchListsCharacterization* (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)›
using ‹*InvariantWatchesDiffer* (*getF state*) (*getWatch1 state*) (*getWatch2 state*)›
using ‹*InvariantWatchCharacterization* (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*)›
using ‹*InvariantVarsF* (*getF state*) *F0 Vbl*›
using ‹*InvariantVarsM* (*getM state*) *F0 Vbl*›
using ‹*InvariantVarsQ* (*getQ state*) *F0 Vbl*›
using ‹*consistent* (*elements* (*getM ?state′*))›
using ‹*uniq* (*elements* (*getM ?state′*))›
using *assertLiteralEffect*[*of state hd ?clause′ False*]
using *varsAppendValuation*[*of elements* (*getM state*) [*hd ?clause′*]]
using *InvariantVarsQAfterAssertLiteral*[*of state hd ?clause′ False F0 Vbl*]
unfolding *InvariantVarsM-def*
unfolding *InvariantConsistent-def*
unfolding *InvariantUniq-def*
by (*auto simp add*: *Let-def*)
moreover
have *exhaustiveUnitPropagate-dom ?state′*

using *exhaustiveUnitPropagateTermination*[*of ?state' F0 Vbl*]
using ‹*InvariantUniqQ* (*getQ ?state'*)›
using ‹*InvariantWatchListsContainOnlyClausesFromF* (*getWatchList ?state'*) (*getF ?state'*)›
using ‹*InvariantWatchListsUniq* (*getWatchList ?state'*)›
using ‹*InvariantWatchListsCharacterization* (*getWatchList ?state'*) (*getWatch1 ?state'*) (*getWatch2 ?state'*)›
using ‹*InvariantWatchesEl* (*getF ?state'*) (*getWatch1 ?state'*) (*getWatch2 ?state'*)›
using ‹*InvariantWatchesDiffer* (*getF ?state'*) (*getWatch1 ?state'*) (*getWatch2 ?state'*)›
using ‹*InvariantQCharacterization* (*getConflictFlag ?state'*) (*getQ ?state'*) (*getF ?state'*) (*getM ?state'*)›
using ‹*InvariantWatchCharacterization* (*getF ?state'*) (*getWatch1 ?state'*) (*getWatch2 ?state'*) (*getM ?state'*)›
using ‹*InvariantConflictFlagCharacterization* (*getConflictFlag ?state'*) (*getF ?state'*) (*getM ?state'*)›
using ‹*consistent* (*elements* (*getM ?state'*))›
using ‹*uniq* (*elements* (*getM ?state'*))›
using ‹*finite Vbl*›
using ‹*InvariantVarsQ* (*getQ ?state'*) *F0 Vbl*›
using ‹*InvariantVarsM* (*getM ?state'*) *F0 Vbl*›
using ‹*InvariantVarsF* (*getF ?state'*) *F0 Vbl*›
unfolding *InvariantConsistent-def*
unfolding *InvariantUniq-def*
by *simp*
ultimately
show *?thesis*
using ‹*exhaustiveUnitPropagate-dom ?state'*›
using *InvariantsAfterExhaustiveUnitPropagate*[*of ?state'*]
using *InvariantConflictClauseCharacterizationAfterExhaustivePropagate*[*of ?state'*]
using *conflictFlagOrQEmptyAfterExhaustiveUnitPropagate*[*of ?state'*]
using *exhaustiveUnitPropagatePreservesCurrentLevel*[*of ?state'*]
using *InvariantGetReasonIsReasonAfterExhaustiveUnitPropagate*[*of ?state'*]
using *assms*
using *assertLiteralEffect*[*of state hd ?clause' False*]
unfolding *InvariantConsistent-def*
unfolding *InvariantUniq-def*
by (*auto simp only*:*Let-def*)
next
case *False*
thus *?thesis*
proof (*cases clauseTautology ?clause'*)
case *True*
thus *?thesis*
using *assms*

using ‹¬ *?clause'* = []›
using ‹¬ *clauseTrue ?clause'* (*elements* (*getM state*))›
using ‹*length ?clause'* ≠ *1*›
unfolding *addClause-def*
by *simp*
next
case *False*
from ‹¬ *?clause'* = []› ‹*length ?clause'* ≠ *1*›
have *length ?clause'* > *1*
by (*induct* (*?clause'*)) *auto*

hence *nth ?clause' 0* ≠ *nth ?clause' 1*
using *distinct-remdups*[*of ?clause'*]
using *nth-eq-iff-index-eq*[*of ?clause' 0 1*]
using ‹¬ *?clause'* = []›
by *auto*

let *?state'* = *let clauseIndex* = *length* (*getF state*) *in*
let state' = *state*⦇ *getF* := (*getF state*) @
[*?clause'*]⦈ *in*
let state'' = *setWatch1 clauseIndex* (*nth ?clause'*
0) *state' in*
let state''' = *setWatch2 clauseIndex* (*nth ?clause'*
1) *state'' in*
state'''

have *InvariantWatchesEl* (*getF ?state'*) (*getWatch1 ?state'*)
(*getWatch2 ?state'*)
using ‹*InvariantWatchesEl* (*getF state*) (*getWatch1 state*)
(*getWatch2 state*)›
using ‹*length ?clause'* > *1*›
using ‹*?clause'* ≠ []›
using *nth-mem*[*of 0 ?clause'*]
using *nth-mem*[*of 1 ?clause'*]
unfolding *InvariantWatchesEl-def*
unfolding *setWatch1-def*
unfolding *setWatch2-def*
by (*auto simp add*: *Let-def nth-append*)
moreover
have *InvariantWatchesDiffer* (*getF ?state'*) (*getWatch1 ?state'*)
(*getWatch2 ?state'*)
using ‹*InvariantWatchesDiffer* (*getF state*) (*getWatch1 state*)
(*getWatch2 state*)›
using ‹*nth ?clause' 0* ≠ *nth ?clause' 1*›
unfolding *InvariantWatchesDiffer-def*
unfolding *setWatch1-def*
unfolding *setWatch2-def*
by (*auto simp add*: *Let-def*)
moreover

have *InvariantWatchListsContainOnlyClausesFromF* (*getWatchList* *?state'*) (*getF ?state'*)
using ‹*InvariantWatchListsContainOnlyClausesFromF* (*getWatchList state*) (*getF state*)›
unfolding *InvariantWatchListsContainOnlyClausesFromF-def*
unfolding *setWatch1-def*
unfolding *setWatch2-def*
by (*auto simp add*:*Let-def*) (*force*)+
moreover
have *InvariantWatchListsCharacterization* (*getWatchList* *?state'*) (*getWatch1 ?state'*) (*getWatch2 ?state'*)
using ‹*InvariantWatchListsCharacterization* (*getWatchList* *state*) (*getWatch1 state*) (*getWatch2 state*)›
using ‹*InvariantWatchListsContainOnlyClausesFromF* (*getWatchList state*) (*getF state*)›
using ‹*nth ?clause' 0* ≠ *nth ?clause' 1*›
unfolding *InvariantWatchListsCharacterization-def*
unfolding *InvariantWatchListsContainOnlyClausesFromF-def*
unfolding *setWatch1-def*
unfolding *setWatch2-def*
by (*auto simp add*:*Let-def*)
moreover
have *InvariantWatchCharacterization* (*getF ?state'*) (*getWatch1* *?state'*) (*getWatch2 ?state'*) (*getM ?state'*)
proof−
{
fix *c*
assume *0* ≤ *c* ∧ *c* < *length* (*getF ?state'*)
fix *www1 www2*
assume *Some www1* = (*getWatch1 ?state' c*) *Some www2* = (*getWatch2 ?state' c*)
have *watchCharacterizationCondition www1 www2* (*getM* *?state'*) (*nth* (*getF ?state'*) *c*) ∧
watchCharacterizationCondition www2 www1 (*getM* *?state'*) (*nth* (*getF ?state'*) *c*)
proof (*cases c* < *length* (*getF state*))
case *True*
hence (*nth* (*getF ?state'*) *c*) = (*nth* (*getF state*) *c*)
unfolding *setWatch1-def*
unfolding *setWatch2-def*
by (*auto simp add*: *Let-def nth-append*)
have *Some www1* = (*getWatch1 state c*) *Some www2* = (*getWatch2 state c*)
using *True*
using ‹*Some www1* = (*getWatch1 ?state' c*)› ‹*Some* *www2* = (*getWatch2 ?state' c*)›
unfolding *setWatch1-def*
unfolding *setWatch2-def*
by (*auto simp add*: *Let-def*)

```
      thus ?thesis
        using ‹InvariantWatchCharacterization (getF state)
(getWatch1 state) (getWatch2 state) (getM state)›
        unfolding InvariantWatchCharacterization-def
        using ‹(nth (getF ?state') c) = (nth (getF state) c)›
        using True
        unfolding setWatch1-def
        unfolding setWatch2-def
        by (auto simp add: Let-def)
    next
      case False
      with ‹0 ≤ c ∧ c < length (getF ?state')›
      have c = length (getF state)
        unfolding setWatch1-def
        unfolding setWatch2-def
        by (auto simp add: Let-def)
         from ‹InvariantWatchesEl (getF ?state') (getWatch1
?state') (getWatch2 ?state')›
      obtain w1 w2
        where
        w1 el ?clause' w2 el ?clause'
        getWatch1 ?state' (length (getF state)) = Some w1
        getWatch2 ?state' (length (getF state)) = Some w2
        unfolding InvariantWatchesEl-def
        unfolding setWatch2-def
        unfolding setWatch1-def
        by (auto simp add: Let-def)
      hence w1 = www1 and w2 = www2
          using ‹Some www1 = (getWatch1 ?state' c)› ‹Some
www2 = (getWatch2 ?state' c)›
        using ‹c = length (getF state)›
        by auto
      have ¬ literalFalse  w1 (elements (getM ?state'))
        ¬ literalFalse w2 (elements (getM ?state'))
        using ‹w1 el ?clause'› ‹w2 el ?clause'›
        using ∗
        unfolding setWatch2-def
        unfolding setWatch1-def
        by (auto simp add: Let-def)
      thus ?thesis
        using ‹w1 = www1› ‹w2 = www2›
        unfolding watchCharacterizationCondition-def
        unfolding setWatch2-def
        unfolding setWatch1-def
        by (auto simp add: Let-def)
    qed
  } thus ?thesis
    unfolding InvariantWatchCharacterization-def
    by auto
```

qed
moreover
have ∀ *l. length* (*getF state*) ∉ *set* (*getWatchList state l*)
using ‹*InvariantWatchListsContainOnlyClausesFromF* (*getWatchList state*) (*getF state*)›
unfolding *InvariantWatchListsContainOnlyClausesFromF-def*
by *auto*
hence *InvariantWatchListsUniq* (*getWatchList ?state′*)
using ‹*InvariantWatchListsUniq* (*getWatchList state*)›
using ‹*nth ?clause′ 0* ≠ *nth ?clause′ 1*›
unfolding *InvariantWatchListsUniq-def*
unfolding *setWatch1-def*
unfolding *setWatch2-def*
by (*auto simp add*:*Let-def uniqAppendIff*)
moreover
from ∗
have ¬ *clauseFalse ?clause′* (*elements* (*getM state*))
using ‹*?clause′* ≠ []›
by (*auto simp add*: *clauseFalseIffAllLiteralsAreFalse*)
hence *InvariantConflictFlagCharacterization* (*getConflictFlag ?state′*) (*getF ?state′*) (*getM ?state′*)
using ‹*InvariantConflictFlagCharacterization* (*getConflictFlag state*) (*getF state*) (*getM state*)›
unfolding *InvariantConflictFlagCharacterization-def*
unfolding *setWatch1-def*
unfolding *setWatch2-def*
by (*auto simp add*: *Let-def formulaFalseIffContainsFalseClause*)
moreover
have ¬ (∃ *l. isUnitClause ?clause′ l* (*elements* (*getM state*)))
proof−
{
assume ¬ *?thesis*
then obtain *l*
where *isUnitClause ?clause′ l* (*elements* (*getM state*))
by *auto*
hence *l el ?clause′*
unfolding *isUnitClause-def*
by *simp*
have ∃ *l′. l′ el ?clause′* ∧ *l* ≠ *l′*
proof−
from ‹*length ?clause′* > *1*›
obtain *a1*::*Literal* **and** *a2*::*Literal*
where *a1 el ?clause′ a2 el ?clause′ a1* ≠ *a2*
using *lengthGtOneTwoDistinctElements*[*of ?clause′*]
by (*auto simp add*: *uniqDistinct*) (*force*)
thus *?thesis*
proof (*cases a1* = *l*)
case *True*

thus *?thesis*
using ‹*a1* ≠ *a2*› ‹*a2 el ?clause'*›
by *auto*
next
case *False*
thus *?thesis*
using ‹*a1 el ?clause'*›
by *auto*
qed
qed
then obtain *l'::Literal*
where *l* ≠ *l' l' el ?clause'*
by *auto*
with ∗
have ¬ *literalFalse l'* (*elements* (*getM state*))
by *simp*
hence *False*
using ‹*isUnitClause ?clause' l* (*elements* (*getM state*))›
using ‹*l* ≠ *l'*› ‹*l' el ?clause'*›
unfolding *isUnitClause-def*
by *auto*
} **thus** *?thesis*
by *auto*
qed
hence *InvariantQCharacterization* (*getConflictFlag ?state'*) (*getQ ?state'*) (*getF ?state'*) (*getM ?state'*)
using *assms*
unfolding *InvariantQCharacterization-def*
unfolding *setWatch2-def*
unfolding *setWatch1-def*
by (*auto simp add*: *Let-def*)
moreover
have *InvariantConflictClauseCharacterization* (*getConflictFlag state*) (*getConflictClause state*) (*getF state* @ [*?clause'*]) (*getM state*)
proof (*cases getConflictFlag state*)
case *False*
thus *?thesis*
unfolding *InvariantConflictClauseCharacterization-def*
by *simp*
next
case *True*
hence *getConflictClause state* < *length* (*getF state*)
using ‹*InvariantConflictClauseCharacterization* (*getConflictFlag state*) (*getConflictClause state*) (*getF state*) (*getM state*)›
unfolding *InvariantConflictClauseCharacterization-def*
by (*auto simp add*: *Let-def*)
hence *nth* ((*getF state*) @ [*?clause'*]) (*getConflictClause state*) =
nth (*getF state*) (*getConflictClause state*)

by (*simp add*: *nth-append*)
thus *?thesis*
using ‹*InvariantConflictClauseCharacterization* (*getConflictFlag state*) (*getConflictClause state*) (*getF state*) (*getM state*)›
unfolding *InvariantConflictClauseCharacterization-def*
by (*auto simp add*: *Let-def clauseFalseAppendValuation*)
qed
moreover
have *InvariantGetReasonIsReason* (*getReason ?state′*) (*getF ?state′*) (*getM ?state′*) (*set* (*getQ ?state′*))
using ‹*currentLevel* (*getM state*) = *0*›
using *elementLevelLeqCurrentLevel*[*of - getM state*]
unfolding *setWatch1-def*
unfolding *setWatch2-def*
unfolding *InvariantGetReasonIsReason-def*
by (*simp add*: *Let-def*)
moreover
have *InvariantVarsF* (*getF ?state′*) *F0 Vbl*
using ‹*InvariantVarsF* (*getF state*) *F0 Vbl*›
using ‹*vars ?clause′* ⊆ *vars F0*›
using *varsAppendFormulae*[*of getF state* [*?clause′*]]
unfolding *setWatch2-def*
unfolding *setWatch1-def*
unfolding *InvariantVarsF-def*
by (*auto simp add*: *Let-def*)
ultimately
show *?thesis*
using *assms*
using ‹*length ?clause′* > *1*›
using ‹¬ *?clause′* = []›
using ‹¬ *clauseTrue ?clause′* (*elements* (*getM state*))›
using ‹*length ?clause′* ≠ *1*›
using ‹¬ *clauseTautology ?clause′*›
unfolding *addClause-def*
unfolding *setWatch1-def*
unfolding *setWatch2-def*
by (*auto simp add*: *Let-def*)
qed
qed
qed
qed
qed

lemma *InvariantEquivalentZLAfterAddClause*:
fixes *Phi* :: *Formula* **and** *clause* :: *Clause* **and** *state* :: *State* **and** *Vbl* :: *Variable set*
assumes
∗:(*getSATFlag state* = *UNDEF* ∧ *InvariantEquivalentZL* (*getF state*)

(*getM state*) *Phi*) ∨
(*getSATFlag state* = *FALSE* ∧ ¬ *satisfiable Phi*)
InvariantConsistent (*getM state*)
InvariantUniq (*getM state*)
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*)
(*getF state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
and
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*)
InvariantConflictFlagCharacterization (*getConflictFlag state*) (*getF state*) (*getM state*)
InvariantQCharacterization (*getConflictFlag state*) (*getQ state*) (*getF state*) (*getM state*)
InvariantUniqQ (*getQ state*)
(*getConflictFlag state*) ∨ (*getQ state*) = []
currentLevel (*getM state*) = *0*
InvariantVarsM (*getM state*) *F0 Vbl*
InvariantVarsQ (*getQ state*) *F0 Vbl*
InvariantVarsF (*getF state*) *F0 Vbl*
finite Vbl
vars clause ⊆ *vars F0*
shows
let state′ = *addClause clause state in*
let Phi′ = *Phi* @ [*clause*] *in*
let Phi′′ = (*if* (*clauseTautology clause*) *then Phi else Phi′*) *in*
(*getSATFlag state′* = *UNDEF* ∧ *InvariantEquivalentZL* (*getF state′*)
(*getM state′*) *Phi′′*) ∨
(*getSATFlag state′* = *FALSE* ∧ ¬*satisfiable Phi′′*)
proof−
let *?clause′* = *remdups* (*removeFalseLiterals clause* (*elements* (*getM state*)))

from ‹*currentLevel* (*getM state*) = *0*›
have *getM state* = *prefixToLevel 0* (*getM state*)
by (*rule currentLevelZeroTrailEqualsItsPrefixToLevelZero*)

have ∗∗: ∀ *l*. *l el ?clause′* ⟶ ¬ *literalFalse l* (*elements* (*getM state*))
unfolding *removeFalseLiterals-def*
by *auto*

have *vars ?clause′* ⊆ *vars clause*
using *varsSubsetValuation*[*of ?clause′ clause*]
unfolding *removeFalseLiterals-def*

by *auto*
hence *vars ?clause′* ⊆ *vars F0*
using ‹*vars clause* ⊆ *vars F0*›
by *simp*

show *?thesis*
proof (*cases clauseTrue ?clause′* (*elements* (*getM state*)))
case *True*
show *?thesis*
proof−
from *True*
have *clauseTrue clause* (*elements* (*getM state*))
using *clauseTrueRemoveDuplicateLiterals*
[*of removeFalseLiterals clause* (*elements* (*getM state*)) *elements*
(*getM state*)]
using *clauseTrueRemoveFalseLiterals*
[*of elements* (*getM state*) *clause*]
using ‹*InvariantConsistent* (*getM state*)›
unfolding *InvariantConsistent-def*
by *simp*
show *?thesis*
proof (*cases getSATFlag state* = *UNDEF*)
case *True*
thus *?thesis*
using ∗
using ‹*clauseTrue clause* (*elements* (*getM state*))›
using ‹*getM state* = *prefixToLevel 0* (*getM state*)›
using *satisfiedClauseCanBeRemoved*
[*of getF state* (*elements* (*prefixToLevel 0* (*getM state*))) *Phi*
clause]
using ‹*clauseTrue ?clause′* (*elements* (*getM state*))›
unfolding *addClause-def*
unfolding *InvariantEquivalentZL-def*
by *auto*
next
case *False*
thus *?thesis*
using ∗
using ‹*clauseTrue ?clause′* (*elements* (*getM state*))›
using *satisfiableAppend*[*of Phi* [*clause*]]
unfolding *addClause-def*
by *force*
qed
qed
next
case *False*
show *?thesis*
proof (*cases ?clause′* = [])
case *True*

show *?thesis*
proof (*cases getSATFlag state* = *UNDEF*)
case *True*
thus *?thesis*
using ∗
using *falseAndDuplicateLiteralsCanBeRemoved*
[*of getF state* (*elements* (*prefixToLevel 0* (*getM state*))) [] *Phi clause*]
using ‹*getM state* = *prefixToLevel 0* (*getM state*)›
using *formulaWithEmptyClauseIsUnsatisfiable*[*of* (*getF state* @ *val2form* (*elements* (*getM state*)) @ [[]])]
using *satisfiableEquivalent*
using ‹*?clause′* = []›
unfolding *addClause-def*
unfolding *InvariantEquivalentZL-def*
using *satisfiableAppendTautology*
by *auto*
next
case *False*
thus *?thesis*
using ‹*?clause′* = []›
using ∗
using *satisfiableAppend*[*of Phi* [*clause*]]
unfolding *addClause-def*
by *force*
qed
next
case *False*
thus *?thesis*
proof (*cases length ?clause′* = *1*)
case *True*
from ‹*length ?clause′* = *1*›
have [*hd ?clause′*] = *?clause′*
using *lengthOneCharacterisation*[*of ?clause′*]
by *simp*

with ‹*length ?clause′* = *1*›
have *val2form* (*elements* (*getM state*)) @ [*?clause′*] = *val2form* ((*elements* (*getM state*)) @ *?clause′*)
using *val2formAppend*[*of elements* (*getM state*) *?clause′*]
using *val2formOfSingleLiteralValuation*[*of ?clause′*]
by *auto*

let *?state′* = *assertLiteral* (*hd ?clause′*) *False state*
have *addClause clause state* = *exhaustiveUnitPropagate ?state′*
using ‹¬ *clauseTrue ?clause′* (*elements* (*getM state*))›
using ‹¬ *?clause′* = []›
using ‹*length ?clause′* = *1*›
unfolding *addClause-def*

by (*simp add*: *Let-def*)
moreover
from ‹*?clause′* ≠ []›
have *hd ?clause′* ∈ *set ?clause′*
using *hd-in-set*[*of ?clause′*]
by *simp*
with ∗∗
have ¬ *literalFalse* (*hd ?clause′*) (*elements* (*getM state*))
by *simp*
hence *consistent* (*elements* ((*getM state*) @ [(*hd ?clause′*, *False*)]))
using *assms*
unfolding *InvariantConsistent-def*
using *consistentAppendElement*[*of elements* (*getM state*) *hd ?clause′*]
by *simp*
hence *consistent* (*elements* (*getM ?state′*))
using *assms*
using *assertLiteralEffect*[*of state hd ?clause′ False*]
by *simp*
moreover
from ‹¬ *clauseTrue ?clause′* (*elements* (*getM state*))›
have *uniq* (*elements* (*getM ?state′*))
using *assms*
using *assertLiteralEffect*[*of state hd ?clause′ False*]
using ‹*hd ?clause′* ∈ *set ?clause′*›
unfolding *InvariantUniq-def*
by (*simp add*: *uniqAppendIff clauseTrueIffContainsTrueLiteral*)
moreover
have *InvariantWatchListsContainOnlyClausesFromF* (*getWatchList ?state′*) (*getF ?state′*) **and**
InvariantWatchListsUniq (*getWatchList ?state′*) **and**
InvariantWatchListsCharacterization (*getWatchList ?state′*) (*getWatch1 ?state′*) (*getWatch2 ?state′*)
InvariantWatchesEl (*getF ?state′*) (*getWatch1 ?state′*) (*getWatch2 ?state′*) **and**
InvariantWatchesDiffer (*getF ?state′*) (*getWatch1 ?state′*) (*getWatch2 ?state′*)
using *assms*
using *WatchInvariantsAfterAssertLiteral*[*of state hd ?clause′ False*]
by (*auto simp add*: *Let-def*)
moreover
have *InvariantWatchCharacterization* (*getF ?state′*) (*getWatch1 ?state′*) (*getWatch2 ?state′*) (*getM ?state′*)
using *assms*
using *InvariantWatchCharacterizationAfterAssertLiteral*[*of state hd ?clause′ False*]
using ‹*uniq* (*elements* (*getM ?state′*))›

using ‹*consistent* (*elements* (*getM ?state'*))›
unfolding *InvariantConsistent-def*
unfolding *InvariantUniq-def*
using *assertLiteralEffect*[*of state hd ?clause' False*]
by (*simp add*: *Let-def*)
moreover
have *InvariantConflictFlagCharacterization* (*getConflictFlag ?state'*) (*getF ?state'*) (*getM ?state'*)
using *assms*
using *InvariantConflictFlagCharacterizationAfterAssertLiteral*[*of state hd ?clause' False*]
using ‹*consistent* (*elements* (*getM ?state'*))›
unfolding *InvariantConsistent-def*
using *assertLiteralEffect*[*of state hd ?clause' False*]
by (*simp add*: *Let-def*)
moreover
have *InvariantQCharacterization* (*getConflictFlag ?state'*) (*getQ ?state'*) (*getF ?state'*) (*getM ?state'*)
proof (*cases getConflictFlag state*)
case *True*
hence *getConflictFlag ?state'*
using *assms*
using *assertLiteralConflictFlagEffect*[*of state hd ?clause' False*]
using ‹*uniq* (*elements* (*getM ?state'*))›
using ‹*consistent* (*elements* (*getM ?state'*))›
unfolding *InvariantConsistent-def*
unfolding *InvariantUniq-def*
using *assertLiteralEffect*[*of state hd ?clause' False*]
by (*auto simp add*: *Let-def*)
thus *?thesis*
using *assms*
unfolding *InvariantQCharacterization-def*
by *simp*
next
case *False*
with ‹(*getConflictFlag state*) ∨ (*getQ state*) = []›
have *getQ state* = []
by *simp*
thus *?thesis*
using *InvariantQCharacterizationAfterAssertLiteralNotInQ*[*of state hd ?clause' False*]
using *assms*
using ‹*uniq* (*elements* (*getM ?state'*))›
using ‹*consistent* (*elements* (*getM ?state'*))›
unfolding *InvariantConsistent-def*
unfolding *InvariantUniq-def*
using *assertLiteralEffect*[*of state hd ?clause' False*]
by (*auto simp add*: *Let-def*)

qed
moreover
have$InvariantUniqQ$ ($getQ$ $?state'$)
using *assms*
using *InvariantUniqQAfterAssertLiteral*[*of state hd ?clause' False*]
by (*simp add*: *Let-def*)
moreover
have *currentLevel* (*getM ?state'*) = *0*
using *assms*
using ‹¬ *clauseTrue ?clause'* (*elements* (*getM state*))›
using ‹¬ *?clause'* = []›
using *assertLiteralEffect*[*of state hd ?clause' False*]
unfolding *addClause-def*
unfolding *currentLevel-def*
by (*simp add*:*Let-def markedElementsAppend*)
moreover
have *var* (*hd ?clause'*) ∈ *vars F0*
using ‹*?clause'* ≠ []›
using *hd-in-set*[*of ?clause'*]
using ‹*vars ?clause'* ⊆ *vars F0*›
using *clauseContainsItsLiteralsVariable*[*of hd ?clause' ?clause'*]
by *auto*
hence *InvariantVarsM* (*getM ?state'*) *F0 Vbl*
InvariantVarsQ (*getQ ?state'*) *F0 Vbl*
InvariantVarsF (*getF ?state'*) *F0 Vbl*
using ‹*InvariantWatchListsContainOnlyClausesFromF* (*getWatchList state*) (*getF state*)›
using ‹*InvariantWatchesEl* (*getF state*) (*getWatch1 state*) (*getWatch2 state*)›
using ‹*InvariantWatchListsUniq* (*getWatchList state*)›
using ‹*InvariantWatchListsCharacterization* (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)›
using ‹*InvariantWatchesDiffer* (*getF state*) (*getWatch1 state*) (*getWatch2 state*)›
using ‹*InvariantWatchCharacterization* (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*)›
using ‹*InvariantVarsF* (*getF state*) *F0 Vbl*›
using ‹*InvariantVarsM* (*getM state*) *F0 Vbl*›
using ‹*InvariantVarsQ* (*getQ state*) *F0 Vbl*›
using ‹*consistent* (*elements* (*getM ?state'*))›
using ‹*uniq* (*elements* (*getM ?state'*))›
using *assertLiteralEffect*[*of state hd ?clause' False*]
using *varsAppendValuation*[*of elements* (*getM state*) [*hd ?clause'*]]
using *InvariantVarsQAfterAssertLiteral*[*of state hd ?clause' False F0 Vbl*]
unfolding *InvariantVarsM-def*
unfolding *InvariantConsistent-def*

unfolding *InvariantUniq-def*
by (*auto simp add*: *Let-def*)
moreover
have *exhaustiveUnitPropagate-dom ?state′*
using *exhaustiveUnitPropagateTermination*[*of ?state′ F0 Vbl*]
using ‹*InvariantUniqQ* (*getQ ?state′*)›
using ‹*InvariantWatchListsContainOnlyClausesFromF* (*getWatchList ?state′*) (*getF ?state′*)›
using ‹*InvariantWatchListsUniq* (*getWatchList ?state′*)›
using ‹*InvariantWatchListsCharacterization* (*getWatchList ?state′*) (*getWatch1 ?state′*) (*getWatch2 ?state′*)›
using ‹*InvariantWatchesEl* (*getF ?state′*) (*getWatch1 ?state′*) (*getWatch2 ?state′*)›
using ‹*InvariantWatchesDiffer* (*getF ?state′*) (*getWatch1 ?state′*) (*getWatch2 ?state′*)›
using ‹*InvariantQCharacterization* (*getConflictFlag ?state′*) (*getQ ?state′*) (*getF ?state′*) (*getM ?state′*)›
using ‹*InvariantWatchCharacterization* (*getF ?state′*) (*getWatch1 ?state′*) (*getWatch2 ?state′*) (*getM ?state′*)›
using ‹*InvariantConflictFlagCharacterization* (*getConflictFlag ?state′*) (*getF ?state′*) (*getM ?state′*)›
using ‹*consistent* (*elements* (*getM ?state′*))›
using ‹*uniq* (*elements* (*getM ?state′*))›
using ‹*finite Vbl*›
using ‹*InvariantVarsM* (*getM ?state′*) *F0 Vbl*›
using ‹*InvariantVarsQ* (*getQ ?state′*) *F0 Vbl*›
using ‹*InvariantVarsF* (*getF ?state′*) *F0 Vbl*›
unfolding *InvariantConsistent-def*
unfolding *InvariantUniq-def*
by *simp*
moreover
have ¬ *clauseTautology clause*
proof−
{
assume ¬ *?thesis*
then obtain *l′*
where *l′ el clause opposite l′ el clause*
by (*auto simp add*: *clauseTautologyCharacterization*)
have *False*
proof (*cases l′ el ?clause′*)
case *True*
have *opposite l′ el ?clause′*
proof−
{
assume ¬ *?thesis*
hence *literalFalse l′* (*elements* (*getM state*))
using ‹*l′ el clause*›
using ‹*opposite l′ el clause*›
using ‹¬ *clauseTrue ?clause′* (*elements* (*getM state*))›

```
                using clauseTrueIffContainsTrueLiteral[of ?clause'
elements (getM state)]
              unfolding removeFalseLiterals-def
              by auto
            hence False
              using ‹l' el ?clause'›
              unfolding removeFalseLiterals-def
              by auto
          } thus ?thesis
            by auto
        qed
        have ∀ x. x el ?clause' ⟶ x = l'
          using ‹l' el ?clause'›
          using ‹length ?clause' = 1›
          using lengthOneImpliesOnlyElement[of ?clause' l']
          by simp
        thus ?thesis
          using ‹opposite l' el ?clause'›
          by auto
      next
        case False
        hence literalFalse l' (elements (getM state))
          using ‹l' el clause›
          unfolding removeFalseLiterals-def
          by simp
        hence ¬ literalFalse (opposite l') (elements (getM state))
          using ‹InvariantConsistent (getM state)›
          unfolding InvariantConsistent-def
          by (auto simp add: inconsistentCharacterization)
        hence opposite l' el ?clause'
          using ‹opposite l' el clause›
          unfolding removeFalseLiterals-def
          by auto
        thus ?thesis
          using ‹literalFalse l' (elements (getM state))›
          using ‹¬ clauseTrue ?clause' (elements (getM state))›
          by (simp add: clauseTrueIffContainsTrueLiteral)
      qed
    } thus ?thesis
      by auto
  qed
  moreover
  note clc = calculation

  show ?thesis
  proof (cases getSATFlag state = UNDEF)
    case True
    hence InvariantEquivalentZL (getF state) (getM state) Phi
      using assms
```

by *simp*
hence *InvariantEquivalentZL* (*getF ?state'*) (*getM ?state'*) (*Phi* @ [*clause*])
using ∗
using *falseAndDuplicateLiteralsCanBeRemoved*
[*of getF state* (*elements* (*prefixToLevel 0* (*getM state*))) [] *Phi clause*]
using ‹[*hd ?clause'*] = *?clause'*›
using ‹*getM state* = *prefixToLevel 0* (*getM state*)›
using ‹*currentLevel* (*getM state*) = *0*›
using *prefixToLevelAppend*[*of 0 getM state* [(*hd ?clause'*, *False*)]]
using ‹*InvariantWatchesEl* (*getF state*) (*getWatch1 state*) (*getWatch2 state*)›
using ‹*InvariantWatchListsContainOnlyClausesFromF* (*getWatchList state*) (*getF state*)›
using *assertLiteralEffect*[*of state hd ?clause' False*]
using ‹*val2form* (*elements* (*getM state*)) @ [*?clause'*] = *val2form* ((*elements* (*getM state*)) @ *?clause'*)›
using ‹¬ *?clause'* = []›
using ‹¬ *clauseTrue ?clause'* (*elements* (*getM state*))›
using ‹*length ?clause'* = *1*›
using ‹*getSATFlag state* = *UNDEF*›
unfolding *addClause-def*
unfolding *InvariantEquivalentZL-def*
by (*simp add*: *Let-def*)
hence *let state''* = *addClause clause state in*
InvariantEquivalentZL (*getF state''*) (*getM state''*) (*Phi* @ [*clause*]) ∧
getSATFlag state'' = *getSATFlag state*
using *clc*
using *InvariantEquivalentZLAfterExhaustiveUnitPropagate*[*of ?state' Phi* @ [*clause*]]
using *exhaustiveUnitPropagatePreservedVariables*[*of ?state'*]
using *assms*
unfolding *InvariantConsistent-def*
unfolding *InvariantUniq-def*
using *assertLiteralEffect*[*of state hd ?clause' False*]
by (*auto simp only*: *Let-def*)
thus *?thesis*
using *True*
using ‹¬ *clauseTautology clause*›
by (*auto simp only*: *Let-def split*: *if-split*)
next
case *False*
hence *getSATFlag state* = *FALSE* ¬ *satisfiable Phi*
using ∗
by *auto*
hence *getSATFlag ?state'* = *FALSE*

```
      using assertLiteralEffect[of state hd ?clause' False]
      using assms
      by simp
    hence getSATFlag (exhaustiveUnitPropagate ?state') = FALSE

      using clc
      using exhaustiveUnitPropagatePreservedVariables[of ?state']
      by (auto simp only: Let-def)
    moreover
    have ¬ satisfiable (Phi @ [clause])
      using satisfiableAppend[of Phi [clause]]
      using ‹¬ satisfiable Phi›
      by auto
    ultimately
    show ?thesis
      using clc
      using ‹¬ clauseTautology clause›
      by (simp only: Let-def) simp
  qed
next
  case False
  thus ?thesis
  proof (cases clauseTautology ?clause')
    case True
    moreover
    hence clauseTautology clause
      unfolding removeFalseLiterals-def
      by (auto simp add: clauseTautologyCharacterization)
    ultimately
    show ?thesis
      using *
      using ‹¬ ?clause' = []›
      using ‹¬ clauseTrue ?clause' (elements (getM state))›
      using ‹length ?clause' ≠ 1›
      using satisfiableAppend[of Phi [clause]]
      unfolding addClause-def

      by (auto simp add: Let-def)
  next
    case False
    have ¬ clauseTautology clause
    proof−
      {
        assume ¬ ?thesis
        then obtain l'
          where l' el clause opposite l' el clause
          by (auto simp add: clauseTautologyCharacterization)
        have False
        proof (cases l' el ?clause')
```

```
        case True
        hence ¬ opposite l' el ?clause'
          using ‹¬ clauseTautology ?clause'›
          by (auto simp add: clauseTautologyCharacterization)
        hence literalFalse (opposite l') (elements (getM state))
          using ‹opposite l' el clause›
          unfolding removeFalseLiterals-def
          by auto
        thus ?thesis
          using ‹¬ clauseTrue ?clause' (elements (getM state))›
          using ‹l' el ?clause'›
          by (simp add: clauseTrueIffContainsTrueLiteral)
      next
        case False
        hence literalFalse l' (elements (getM state))
          using ‹l' el clause›
          unfolding removeFalseLiterals-def
          by auto
        hence ¬ literalFalse (opposite l') (elements (getM state))
          using ‹InvariantConsistent (getM state)›
          unfolding InvariantConsistent-def
          by (auto simp add: inconsistentCharacterization)
        hence opposite l' el ?clause'
          using ‹opposite l' el clause›
          unfolding removeFalseLiterals-def
          by auto
        thus ?thesis
          using ‹¬ clauseTrue ?clause' (elements (getM state))›
          using ‹literalFalse l' (elements (getM state))›
          by (simp add: clauseTrueIffContainsTrueLiteral)
      qed
    } thus ?thesis
      by auto
  qed
  show ?thesis
  proof (cases getSATFlag state = UNDEF)
    case True
    show ?thesis
      using *
      using falseAndDuplicateLiteralsCanBeRemoved
      [of getF state (elements (prefixToLevel 0 (getM state))) []
Phi clause]
      using ‹getM state = prefixToLevel 0 (getM state)›
      using ‹¬ ?clause' = []›
      using ‹¬ clauseTrue ?clause' (elements (getM state))›
      using ‹length ?clause' ≠ 1›
      using ‹¬ clauseTautology ?clause'›
      using ‹¬ clauseTautology clause›
      using ‹getSATFlag state = UNDEF›
```

unfolding *addClause-def*
unfolding *InvariantEquivalentZL-def*
unfolding *setWatch1-def*
unfolding *setWatch2-def*
using *clauseOrderIrrelevant*[*of getF state* [*?clause'*] *val2form* (*elements* (*getM state*)) []]
using *equivalentFormulaeTransitivity*[*of*
getF state @ *remdups* (*removeFalseLiterals clause* (*elements* (*getM state*))) # *val2form* (*elements* (*getM state*))
getF state @ *val2form* (*elements* (*getM state*)) @ [*remdups* (*removeFalseLiterals clause* (*elements* (*getM state*)))]
Phi @ [*clause*]]
by (*auto simp add*: *Let-def*)
next
case *False*
thus *?thesis*
using ∗
using *satisfiableAppend*[*of Phi* [*clause*]]
using ‹¬ *clauseTrue ?clause'* (*elements* (*getM state*))›
using ‹*length ?clause'* ≠ *1*›
using ‹¬ *clauseTautology ?clause'*›
using ‹¬ *clauseTautology clause*›
unfolding *addClause-def*
unfolding *setWatch1-def*
unfolding *setWatch2-def*
by (*auto simp add*: *Let-def*)
qed
qed
qed
qed
qed
qed

lemma *InvariantsAfterInitializationStep*:
fixes
state :: *State* **and** *Phi* :: *Formula* **and** *Vbl*::*Variable set*
assumes
InvariantConsistent (*getM state*)
InvariantUniq (*getM state*)
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)

and
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*)
InvariantConflictFlagCharacterization (*getConflictFlag state*) (*getF state*) (*getM state*)
InvariantConflictClauseCharacterization (*getConflictFlag state*) (*getConflictClause state*) (*getF state*) (*getM state*)
InvariantQCharacterization (*getConflictFlag state*) (*getQ state*) (*getF state*) (*getM state*)
InvariantGetReasonIsReason (*getReason state*) (*getF state*) (*getM state*) (*set* (*getQ state*))
InvariantUniqQ (*getQ state*)
(*getConflictFlag state*) $\vee$ (*getQ state*) = []
currentLevel (*getM state*) = *0*
finite Vbl
InvariantVarsM (*getM state*) *F0 Vbl*
InvariantVarsQ (*getQ state*) *F0 Vbl*
InvariantVarsF (*getF state*) *F0 Vbl*
state' = *initialize Phi state*
set Phi $\subseteq$ *set F0*
shows
InvariantConsistent (*getM state'*) $\wedge$
InvariantUniq (*getM state'*) $\wedge$
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state'*) (*getF state'*) $\wedge$
InvariantWatchListsUniq (*getWatchList state'*) $\wedge$
InvariantWatchListsCharacterization (*getWatchList state'*) (*getWatch1 state'*) (*getWatch2 state'*) $\wedge$
InvariantWatchesEl (*getF state'*) (*getWatch1 state'*) (*getWatch2 state'*) $\wedge$
InvariantWatchesDiffer (*getF state'*) (*getWatch1 state'*) (*getWatch2 state'*) $\wedge$
InvariantWatchCharacterization (*getF state'*) (*getWatch1 state'*) (*getWatch2 state'*) (*getM state'*) $\wedge$
InvariantConflictFlagCharacterization (*getConflictFlag state'*) (*getF state'*) (*getM state'*) $\wedge$
InvariantConflictClauseCharacterization (*getConflictFlag state'*) (*getConflictClause state'*) (*getF state'*) (*getM state'*) $\wedge$
InvariantQCharacterization (*getConflictFlag state'*) (*getQ state'*) (*getF state'*) (*getM state'*) $\wedge$
InvariantUniqQ (*getQ state'*) $\wedge$
InvariantGetReasonIsReason (*getReason state'*) (*getF state'*) (*getM state'*) (*set* (*getQ state'*)) $\wedge$
InvariantVarsM (*getM state'*) *F0 Vbl* $\wedge$
InvariantVarsQ (*getQ state'*) *F0 Vbl* $\wedge$
InvariantVarsF (*getF state'*) *F0 Vbl* $\wedge$
((*getConflictFlag state'*) $\vee$ (*getQ state'*) = []) $\wedge$

currentLevel (*getM state'*) = *0* (**is** *?Inv state'*)
using *assms*
proof (*induct Phi arbitrary*: *state*)
case *Nil*
thus *?case*
by *simp*
next
case (*Cons clause Phi'*)
let *?state'* = *addClause clause state*
have *?Inv ?state'*
using *Cons*
using *InvariantsAfterAddClause*[*of state F0 Vbl clause*]
using *formulaContainsItsClausesVariables*[*of clause F0*]
by (*simp add*: *Let-def*)
thus *?case*
using *Cons*(*1*)[*of ?state'*] ‹*finite Vbl*› *Cons*(*18*) *Cons*(*19*) *Cons*(*20*) *Cons*(*21*) *Cons*(*22*)
by (*simp add*: *Let-def*)
qed

lemma *InvariantEquivalentZLAfterInitializationStep*:
fixes *Phi* :: *Formula*
assumes
(*getSATFlag state* = *UNDEF* ∧ *InvariantEquivalentZL* (*getF state*) (*getM state*) (*filter* (λ *c*. ¬ *clauseTautology c*) *Phi*)) ∨
(*getSATFlag state* = *FALSE* ∧ ¬ *satisfiable* (*filter* (λ *c*. ¬ *clause-Tautology c*) *Phi*))
InvariantConsistent (*getM state*)
InvariantUniq (*getM state*)
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*)
InvariantConflictFlagCharacterization (*getConflictFlag state*) (*getF state*) (*getM state*)
InvariantConflictClauseCharacterization (*getConflictFlag state*) (*getConflictClause state*) (*getF state*) (*getM state*)
InvariantQCharacterization (*getConflictFlag state*) (*getQ state*) (*getF state*) (*getM state*)
InvariantNoDecisionsWhenConflict (*getF state*) (*getM state*) (*currentLevel* (*getM state*))
InvariantNoDecisionsWhenUnit (*getF state*) (*getM state*) (*currentLevel*

(*getM state*))
InvariantGetReasonIsReason (*getReason state*) (*getF state*) (*getM state*) (*set* (*getQ state*))
InvariantUniqQ (*getQ state*)
finite Vbl
InvariantVarsM (*getM state*) *F0 Vbl*
InvariantVarsQ (*getQ state*) *F0 Vbl*
InvariantVarsF (*getF state*) *F0 Vbl*
(*getConflictFlag state*) ∨ (*getQ state*) = []
currentLevel (*getM state*) = *0*
F0 = *Phi* @ *Phi'*
shows
let state' = *initialize Phi' state in*
(*getSATFlag state'* = *UNDEF* ∧ *InvariantEquivalentZL* (*getF state'*) (*getM state'*) (*filter* (*λ c.* ¬ *clauseTautology c*) *F0*)) ∨
(*getSATFlag state'* = *FALSE* ∧ ¬*satisfiable* (*filter* (*λ c.* ¬ *clauseTautology c*) *F0*))
using *assms*
proof (*induct Phi' arbitrary*: *state Phi*)
case *Nil*
thus *?case*
unfolding *prefixToLevel-def equivalentFormulae-def*
by *simp*
next
case (*Cons clause Phi''*)
let *?filt* = *λ F.* (*filter* (*λ c.* ¬ *clauseTautology c*) *F*)
let *?state'* = *addClause clause state*
let *?Phi'* = *?filt Phi* @ [*clause*]
let *?Phi''* = *if clauseTautology clause then ?filt Phi else ?Phi'*
from *Cons*
have *getSATFlag ?state'* = *UNDEF* ∧ *InvariantEquivalentZL* (*getF ?state'*) (*getM ?state'*) (*?filt ?Phi''*) ∨
getSATFlag ?state' = *FALSE* ∧ ¬ *satisfiable* (*?filt ?Phi''*)
using *formulaContainsItsClausesVariables*[*of clause F0*]
using *InvariantEquivalentZLAfterAddClause*[*of state ?filt Phi F0 Vbl clause*]
by (*simp add*:*Let-def*)
hence *getSATFlag ?state'* = *UNDEF* ∧ *InvariantEquivalentZL* (*getF ?state'*) (*getM ?state'*) (*?filt* (*Phi* @ [*clause*])) ∨
getSATFlag ?state' = *FALSE* ∧ ¬ *satisfiable* (*?filt* (*Phi* @ [*clause*]))
by *auto*
moreover
from *Cons*
have *InvariantConsistent* (*getM ?state'*) ∧
InvariantUniq (*getM ?state'*) ∧
InvariantWatchListsContainOnlyClausesFromF (*getWatchList ?state'*) (*getF ?state'*) ∧
InvariantWatchListsUniq (*getWatchList ?state'*) ∧

InvariantWatchListsCharacterization (*getWatchList ?state′*) (*getWatch1 ?state′*) (*getWatch2 ?state′*) ∧
InvariantWatchesEl (*getF ?state′*) (*getWatch1 ?state′*) (*getWatch2 ?state′*) ∧
InvariantWatchesDiffer (*getF ?state′*) (*getWatch1 ?state′*) (*getWatch2 ?state′*) ∧
InvariantWatchCharacterization (*getF ?state′*) (*getWatch1 ?state′*) (*getWatch2 ?state′*) (*getM ?state′*) ∧
InvariantConflictFlagCharacterization (*getConflictFlag ?state′*) (*getF ?state′*) (*getM ?state′*) ∧
InvariantConflictClauseCharacterization (*getConflictFlag ?state′*) (*getConflictClause ?state′*) (*getF ?state′*) (*getM ?state′*) ∧
InvariantQCharacterization (*getConflictFlag ?state′*) (*getQ ?state′*) (*getF ?state′*) (*getM ?state′*) ∧
InvariantGetReasonIsReason (*getReason ?state′*) (*getF ?state′*) (*getM ?state′*) (*set* (*getQ ?state′*)) ∧
InvariantUniqQ (*getQ ?state′*) ∧
InvariantVarsM (*getM ?state′*) *F0 Vbl* ∧
InvariantVarsQ (*getQ ?state′*) *F0 Vbl* ∧
InvariantVarsF (*getF ?state′*) *F0 Vbl* ∧
((*getConflictFlag ?state′*) ∨ (*getQ ?state′*) = []) ∧
currentLevel (*getM ?state′*) = *0*
using *formulaContainsItsClausesVariables*[*of clause F0*]
using *InvariantsAfterAddClause*
by (*simp add*: *Let-def*)
moreover
hence *InvariantNoDecisionsWhenConflict* (*getF ?state′*) (*getM ?state′*) (*currentLevel* (*getM ?state′*))
InvariantNoDecisionsWhenUnit (*getF ?state′*) (*getM ?state′*) (*currentLevel* (*getM ?state′*))
unfolding *InvariantNoDecisionsWhenConflict-def*
unfolding *InvariantNoDecisionsWhenUnit-def*
by *auto*
ultimately
show *?case*
using *Cons*(*1*)[*of ?state′ Phi* @ [*clause*]] ‹*finite Vbl*› *Cons*(*23*) *Cons*(*24*)
by (*simp add*: *Let-def*)
qed

lemma *InvariantsAfterInitialization*:
shows
let state′ = (*initialize F0 initialState*) *in*
InvariantConsistent (*getM state′*) ∧
InvariantUniq (*getM state′*) ∧
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state′*) (*getF state′*) ∧
InvariantWatchListsUniq (*getWatchList state′*) ∧
InvariantWatchListsCharacterization (*getWatchList state′*) (*getWatch1*

state′) (*getWatch2 state′*) ∧
InvariantWatchesEl (*getF state′*) (*getWatch1 state′*) (*getWatch2 state′*) ∧
InvariantWatchesDiffer (*getF state′*) (*getWatch1 state′*) (*getWatch2 state′*) ∧
InvariantWatchCharacterization (*getF state′*) (*getWatch1 state′*) (*getWatch2 state′*) (*getM state′*) ∧
InvariantConflictFlagCharacterization (*getConflictFlag state′*) (*getF state′*) (*getM state′*) ∧
InvariantConflictClauseCharacterization (*getConflictFlag state′*) (*getConflictClause state′*) (*getF state′*) (*getM state′*) ∧
InvariantQCharacterization (*getConflictFlag state′*) (*getQ state′*) (*getF state′*) (*getM state′*) ∧
InvariantNoDecisionsWhenConflict (*getF state′*) (*getM state′*) (*currentLevel* (*getM state′*)) ∧
InvariantNoDecisionsWhenUnit (*getF state′*) (*getM state′*) (*currentLevel* (*getM state′*)) ∧
InvariantGetReasonIsReason (*getReason state′*) (*getF state′*) (*getM state′*) (*set* (*getQ state′*)) ∧
InvariantUniqQ (*getQ state′*) ∧
InvariantVarsM (*getM state′*) *F0* {} ∧
InvariantVarsQ (*getQ state′*) *F0* {} ∧
InvariantVarsF (*getF state′*) *F0* {} ∧
((*getConflictFlag state′*) ∨ (*getQ state′*) = []) ∧
currentLevel (*getM state′*) = *0*
using *InvariantsAfterInitializationStep*[*of initialState* {} *F0 initialize F0 initialState F0*]
unfolding *initialState-def*
unfolding *InvariantConsistent-def*
unfolding *InvariantUniq-def*
unfolding *InvariantWatchListsContainOnlyClausesFromF-def*
unfolding *InvariantWatchListsUniq-def*
unfolding *InvariantWatchListsCharacterization-def*
unfolding *InvariantWatchesEl-def*
unfolding *InvariantWatchesDiffer-def*
unfolding *InvariantWatchCharacterization-def*
unfolding *watchCharacterizationCondition-def*
unfolding *InvariantConflictFlagCharacterization-def*
unfolding *InvariantConflictClauseCharacterization-def*
unfolding *InvariantQCharacterization-def*
unfolding *InvariantUniqQ-def*
unfolding *InvariantNoDecisionsWhenConflict-def*
unfolding *InvariantNoDecisionsWhenUnit-def*
unfolding *InvariantGetReasonIsReason-def*
unfolding *InvariantVarsM-def*
unfolding *InvariantVarsQ-def*
unfolding *InvariantVarsF-def*
unfolding *currentLevel-def*
by (*simp*) (*force*)

lemma *InvariantEquivalentZLAfterInitialization*:
fixes *F0* :: *Formula*
shows
let state′ = (*initialize F0 initialState*) *in*
let F0′ = (*filter* (λ *c*. ¬ *clauseTautology c*) *F0*) *in*
(*getSATFlag state′* = *UNDEF* ∧ *InvariantEquivalentZL* (*getF state′*) (*getM state′*) *F0′*) ∨
(*getSATFlag state′* = *FALSE* ∧ ¬ *satisfiable F0′*)
using *InvariantEquivalentZLAfterInitializationStep*[*of initialState* [] {} *F0 F0*]
unfolding *initialState-def*
unfolding *InvariantEquivalentZL-def*
unfolding *InvariantConsistent-def*
unfolding *InvariantUniq-def*
unfolding *InvariantWatchesEl-def*
unfolding *InvariantWatchesDiffer-def*
unfolding *InvariantWatchListsContainOnlyClausesFromF-def*
unfolding *InvariantWatchListsUniq-def*
unfolding *InvariantWatchListsCharacterization-def*
unfolding *InvariantWatchCharacterization-def*
unfolding *InvariantConflictFlagCharacterization-def*
unfolding *InvariantConflictClauseCharacterization-def*
unfolding *InvariantQCharacterization-def*
unfolding *InvariantNoDecisionsWhenConflict-def*
unfolding *InvariantNoDecisionsWhenUnit-def*
unfolding *InvariantGetReasonIsReason-def*
unfolding *InvariantVarsM-def*
unfolding *InvariantVarsQ-def*
unfolding *InvariantVarsF-def*
unfolding *watchCharacterizationCondition-def*
unfolding *InvariantUniqQ-def*
unfolding *prefixToLevel-def*
unfolding *equivalentFormulae-def*
unfolding *currentLevel-def*
by (*auto simp add*: *Let-def*)

end
theory *ConflictAnalysis*
imports *AssertLiteral*
begin

lemma *clauseFalseInPrefixToLastAssertedLiteral*:
assumes
isLastAssertedLiteral l (*oppositeLiteralList c*) (*elements M*) **and**

clauseFalse c (*elements M*) **and**
uniq (*elements M*)
shows *clauseFalse c* (*elements* (*prefixToLevel* (*elementLevel l M*) *M*))
proof−
{
fix *l*′::*Literal*
assume *l*′ *el c*
hence *literalFalse l*′ (*elements M*)
using ‹*clauseFalse c* (*elements M*)›
by (*simp add*: *clauseFalseIffAllLiteralsAreFalse*)
hence *literalTrue* (*opposite l*′) (*elements M*)
by *simp*

have *opposite l*′ *el oppositeLiteralList c*
using ‹*l*′ *el c*›
using *literalElListIffOppositeLiteralElOppositeLiteralList*[*of l*′ *c*]
by *simp*

have *elementLevel* (*opposite l*′) *M* ≤ *elementLevel l M*
using *lastAssertedLiteralHasHighestElementLevel*[*of l oppositeLiteralList c M*]
using ‹*isLastAssertedLiteral l* (*oppositeLiteralList c*) (*elements M*)›
using ‹*uniq* (*elements M*)›
using ‹*opposite l*′ *el oppositeLiteralList c*›
using ‹*literalTrue* (*opposite l*′) (*elements M*)›
by *auto*
hence *opposite l*′ *el* (*elements* (*prefixToLevel* (*elementLevel l M*) *M*))
using *elementLevelLtLevelImpliesMemberPrefixToLevel*[*of opposite l*′ *M elementLevel l M*]
using ‹*literalTrue* (*opposite l*′) (*elements M*)›
by *simp*
} **thus** *?thesis*
by (*simp add*: *clauseFalseIffAllLiteralsAreFalse*)
qed

lemma *InvariantNoDecisionsWhenConflictEnsuresCurrentLevelCl*:
assumes
InvariantNoDecisionsWhenConflict F M (*currentLevel M*)
clause el F
clauseFalse clause (*elements M*)
uniq (*elements M*)
currentLevel M > *0*
shows
clause ≠ [] ∧
(*let Cl* = *getLastAssertedLiteral* (*oppositeLiteralList clause*) (*elements*

M) *in*
InvariantClCurrentLevel Cl M)
proof−
have *clause* ≠ []
proof−
{
assume ¬ *?thesis*
hence *clauseFalse clause* (*elements* (*prefixToLevel* ((*currentLevel M*) − *1*) *M*))
by *simp*
hence *False*
using ‹*InvariantNoDecisionsWhenConflict F M* (*currentLevel M*)›
using ‹*currentLevel M* > *0*›
using ‹*clause el F*›
unfolding *InvariantNoDecisionsWhenConflict-def*
by (*simp add*: *formulaFalseIffContainsFalseClause*)
} **thus** *?thesis*
by *auto*
qed
moreover
let *?Cl* = *getLastAssertedLiteral* (*oppositeLiteralList clause*) (*elements M*)
have *elementLevel ?Cl M* = *currentLevel M*
proof−
have *elementLevel ?Cl M* ≤ *currentLevel M*
using *elementLevelLeqCurrentLevel*[*of ?Cl M*]
by *simp*
moreover
have *elementLevel ?Cl M* ≥ *currentLevel M*
proof−
{
assume *elementLevel ?Cl M* < *currentLevel M*
have *isLastAssertedLiteral ?Cl* (*oppositeLiteralList clause*) (*elements M*)
using *getLastAssertedLiteralCharacterization*[*of clause elements M*]
using ‹*uniq* (*elements M*)›
using ‹*clauseFalse clause* (*elements M*)›
using ‹*clause* ≠ []›
by *simp*
hence *clauseFalse clause* (*elements* (*prefixToLevel* (*elementLevel ?Cl M*) *M*))
using *clauseFalseInPrefixToLastAssertedLiteral*[*of ?Cl clause M*]
using ‹*clauseFalse clause* (*elements M*)›
using ‹*uniq* (*elements M*)›
by *simp*
hence *False*

using ‹*clause el F*›
using ‹*InvariantNoDecisionsWhenConflict F M* (*currentLevel M*)›
using ‹*currentLevel M* > *0*›
unfolding *InvariantNoDecisionsWhenConflict-def*
using ‹*elementLevel ?Cl M* < *currentLevel M*›
by (*simp add*: *formulaFalseIffContainsFalseClause*)
} **thus** *?thesis*
by *force*
qed
ultimately
show *?thesis*
by *simp*
qed
ultimately
show *?thesis*
unfolding *InvariantClCurrentLevel-def*
by (*simp add*: *Let-def*)
qed

lemma *InvariantsClAfterApplyConflict*:
assumes
getConflictFlag state
InvariantUniq (*getM state*)
InvariantNoDecisionsWhenConflict (*getF state*) (*getM state*) (*currentLevel* (*getM state*))
InvariantEquivalentZL (*getF state*) (*getM state*) *F0*
InvariantConflictClauseCharacterization (*getConflictFlag state*) (*getConflictClause state*) (*getF state*) (*getM state*)
currentLevel (*getM state*) > *0*
shows
let state′ = *applyConflict state in*
InvariantCFalse (*getConflictFlag state′*) (*getM state′*) (*getC state′*) ∧
InvariantCEntailed (*getConflictFlag state′*) *F0* (*getC state′*) ∧

InvariantClCharacterization (*getCl state′*) (*getC state′*) (*getM state′*) ∧
InvariantClCurrentLevel (*getCl state′*) (*getM state′*) ∧
InvariantCnCharacterization (*getCn state′*) (*getC state′*) (*getM state′*) ∧
InvariantUniqC (*getC state′*)
proof−
let *?M0* = *elements* (*prefixToLevel 0* (*getM state*))
let *?oppM0* = *oppositeLiteralList ?M0*

let *?clause′* = *nth* (*getF state*) (*getConflictClause state*)
let *?clause′′* = *list-diff ?clause′ ?oppM0*
let *?clause* = *remdups ?clause′′*

let *?l* = *getLastAssertedLiteral* (*oppositeLiteralList ?clause'*) (*elements* (*getM state*))

have *clauseFalse ?clause'* (*elements* (*getM state*)) *?clause' el* (*getF state*)
using ‹*getConflictFlag state*›
using ‹*InvariantConflictClauseCharacterization* (*getConflictFlag state*) (*getConflictClause state*) (*getF state*) (*getM state*)›
unfolding *InvariantConflictClauseCharacterization-def*
by (*auto simp add*: *Let-def*)

have *?clause'* ≠ [] *elementLevel ?l* (*getM state*) = *currentLevel* (*getM state*)
using *InvariantNoDecisionsWhenConflictEnsuresCurrentLevelCl*[*of getF state getM state ?clause'*]
using ‹*?clause' el* (*getF state*)›
using ‹*clauseFalse ?clause'* (*elements* (*getM state*))›
using ‹*InvariantNoDecisionsWhenConflict* (*getF state*) (*getM state*) (*currentLevel* (*getM state*))›
using ‹*currentLevel* (*getM state*) > *0*›
using ‹*InvariantUniq* (*getM state*)›
unfolding *InvariantUniq-def*
unfolding *InvariantClCurrentLevel-def*
by (*auto simp add*: *Let-def*)

have *isLastAssertedLiteral ?l* (*oppositeLiteralList ?clause'*) (*elements* (*getM state*))
using ‹*?clause'* ≠ []›
using ‹*clauseFalse ?clause'* (*elements* (*getM state*))›
using ‹*InvariantUniq* (*getM state*)›
unfolding *InvariantUniq-def*
using *getLastAssertedLiteralCharacterization*[*of ?clause' elements* (*getM state*)]
by *simp*
hence *?l el* (*oppositeLiteralList ?clause'*)
unfolding *isLastAssertedLiteral-def*
by *simp*
hence *opposite ?l el ?clause'*
using *literalElListIffOppositeLiteralElOppositeLiteralList*[*of opposite ?l ?clause'*]
by *auto*

have ¬ *?l el ?M0*
proof−
{
assume ¬ *?thesis*
hence *elementLevel ?l* (*getM state*) = *0*
using *prefixToLevelElementsElementLevel*[*of ?l 0 getM state*]

by *simp*
hence *False*
using ‹*elementLevel ?l* (*getM state*) = *currentLevel* (*getM state*)›
using ‹*currentLevel* (*getM state*) > *0*›
by *simp*
}
thus *?thesis*
by *auto*
qed

hence ¬ *opposite ?l el ?oppM0*
using *literalElListIffOppositeLiteralElOppositeLiteralList*[*of ?l elements* (*prefixToLevel 0* (*getM state*))]
by *simp*

have *opposite ?l el ?clause''*
using ‹*opposite ?l el ?clause'*›
using ‹¬ *opposite ?l el ?oppM0*›
using *listDiffIff*[*of opposite ?l ?clause' ?oppM0*]
by *simp*
hence *?l el* (*oppositeLiteralList ?clause''*)
using *literalElListIffOppositeLiteralElOppositeLiteralList*[*of opposite ?l ?clause''*]
by *simp*

have *set* (*oppositeLiteralList ?clause''*) ⊆ *set* (*oppositeLiteralList ?clause'*)
proof
fix *x*
assume *x* ∈ *set* (*oppositeLiteralList ?clause''*)
thus *x* ∈ *set* (*oppositeLiteralList ?clause'*)
using *literalElListIffOppositeLiteralElOppositeLiteralList*[*of opposite x ?clause''*]
using *literalElListIffOppositeLiteralElOppositeLiteralList*[*of opposite x ?clause'*]
using *listDiffIff*[*of opposite x ?clause' oppositeLiteralList* (*elements* (*prefixToLevel 0* (*getM state*)))]
by *auto*
qed

have *isLastAssertedLiteral ?l* (*oppositeLiteralList ?clause''*) (*elements* (*getM state*))
using ‹*?l el* (*oppositeLiteralList ?clause''*)›
using ‹*set* (*oppositeLiteralList ?clause''*) ⊆ *set* (*oppositeLiteralList ?clause'*)›
using ‹*isLastAssertedLiteral ?l* (*oppositeLiteralList ?clause'*) (*elements* (*getM state*))›
using *isLastAssertedLiteralSubset*[*of ?l oppositeLiteralList ?clause' elements* (*getM state*) *oppositeLiteralList ?clause''*]

by *auto*
moreover
have *set* (*oppositeLiteralList ?clause*) = *set* (*oppositeLiteralList ?clause*′′)
unfolding *oppositeLiteralList-def*
by *simp*
ultimately
have *isLastAssertedLiteral ?l* (*oppositeLiteralList ?clause*) (*elements* (*getM state*))
unfolding *isLastAssertedLiteral-def*
by *auto*

hence *?l el* (*oppositeLiteralList ?clause*)
unfolding *isLastAssertedLiteral-def*
by *simp*
hence *opposite ?l el ?clause*
using *literalElListIffOppositeLiteralElOppositeLiteralList*[*of opposite ?l ?clause*]
by *simp*
hence *?clause* ≠ []
by *auto*

have *clauseFalse ?clause*′′ (*elements* (*getM state*))
proof−
{
fix *l*::*Literal*
assume *l el ?clause*′′
hence *l el ?clause*′
using *listDiffIff*[*of l ?clause*′ *?oppM0*]
by *simp*
hence *literalFalse l* (*elements* (*getM state*))
using ‹*clauseFalse ?clause*′ (*elements* (*getM state*))›
by (*simp add*: *clauseFalseIffAllLiteralsAreFalse*)
}
thus *?thesis*
by (*simp add*: *clauseFalseIffAllLiteralsAreFalse*)
qed
hence *clauseFalse ?clause* (*elements* (*getM state*))
by (*simp add*: *clauseFalseIffAllLiteralsAreFalse*)

let *?l*′ = *getLastAssertedLiteral* (*oppositeLiteralList ?clause*) (*elements* (*getM state*))
have *isLastAssertedLiteral ?l*′ (*oppositeLiteralList ?clause*) (*elements* (*getM state*))
using ‹*?clause* ≠ []›
using ‹*clauseFalse ?clause* (*elements* (*getM state*))›
using ‹*InvariantUniq* (*getM state*)›
unfolding *InvariantUniq-def*
using *getLastAssertedLiteralCharacterization*[*of ?clause elements* (*getM state*)]

by *simp*
with ‹*isLastAssertedLiteral ?l* (*oppositeLiteralList ?clause*) (*elements* (*getM state*))›
have *?l* = *?l'*
using *lastAssertedLiteralIsUniq*
by *simp*

have *formulaEntailsClause* (*getF state*) *?clause'*
using ‹*?clause' el* (*getF state*)›
by (*simp add*: *formulaEntailsItsClauses*)

let *?F0* = (*getF state*) @ *val2form ?M0*

have *formulaEntailsClause ?F0 ?clause'*
using ‹*formulaEntailsClause* (*getF state*) *?clause'*›
by (*simp add*: *formulaEntailsClauseAppend*)

hence *formulaEntailsClause ?F0 ?clause''*
using ‹*formulaEntailsClause* (*getF state*) *?clause'*›
using *formulaEntailsClauseRemoveEntailedLiteralOpposites*[*of ?F0* *?clause' ?M0*]
using *val2formIsEntailed*[*of getF state ?M0* []]
by *simp*
hence *formulaEntailsClause ?F0 ?clause*
unfolding *formulaEntailsClause-def*
by (*simp add*: *clauseTrueIffContainsTrueLiteral*)

hence *formulaEntailsClause F0 ?clause*
using ‹*InvariantEquivalentZL* (*getF state*) (*getM state*) *F0*›
unfolding *InvariantEquivalentZL-def*
unfolding *formulaEntailsClause-def*
unfolding *equivalentFormulae-def*
by *auto*

show *?thesis*
using ‹*isLastAssertedLiteral ?l'* (*oppositeLiteralList ?clause*) (*elements* (*getM state*))›
using ‹*?l* = *?l'*›
using ‹*elementLevel ?l* (*getM state*) = *currentLevel* (*getM state*)›
using ‹*clauseFalse ?clause* (*elements* (*getM state*))›
using ‹*formulaEntailsClause F0 ?clause*›
unfolding *applyConflict-def*
unfolding *setConflictAnalysisClause-def*
unfolding *InvariantClCharacterization-def*
unfolding *InvariantClCurrentLevel-def*
unfolding *InvariantCFalse-def*
unfolding *InvariantCEntailed-def*
unfolding *InvariantCnCharacterization-def*
unfolding *InvariantUniqC-def*

by (*auto simp add*: *findLastAssertedLiteral-def countCurrentLevel-Literals-def Let-def uniqDistinct distinct-remdups-id*)
qed

lemma *CnEqual1IffUIP*:
assumes
InvariantClCharacterization (*getCl state*) (*getC state*) (*getM state*)
InvariantClCurrentLevel (*getCl state*) (*getM state*)
InvariantCnCharacterization (*getCn state*) (*getC state*) (*getM state*)
shows
(*getCn state* = *1*) = *isUIP* (*opposite* (*getCl state*)) (*getC state*) (*getM state*)
proof−
let *?clls* = *filter* (λ *l*. *elementLevel* (*opposite l*) (*getM state*) = *currentLevel* (*getM state*)) (*remdups* (*getC state*))
let *?Cl* = *getCl state*

have *isLastAssertedLiteral ?Cl* (*oppositeLiteralList* (*getC state*)) (*elements* (*getM state*))
using ‹*InvariantClCharacterization* (*getCl state*) (*getC state*) (*getM state*)›
unfolding *InvariantClCharacterization-def*
.
hence *literalTrue ?Cl* (*elements* (*getM state*)) *?Cl el* (*oppositeLiteralList* (*getC state*))
unfolding *isLastAssertedLiteral-def*
by *auto*
hence *opposite ?Cl el getC state*
using *literalElListIffOppositeLiteralElOppositeLiteralList*[*of opposite ?Cl getC state*]
by *simp*

hence *opposite ?Cl el ?clls*
using ‹*InvariantClCurrentLevel* (*getCl state*) (*getM state*)›
unfolding *InvariantClCurrentLevel-def*
by *auto*
hence *?clls* ≠ []
by *force*
hence *length ?clls* > *0*
by *simp*

have *uniq ?clls*
by (*simp add*: *uniqDistinct*)

{

assume *getCn state* ≠ *1*
hence *length ?clls* > *1*
using *assms*
using ‹*length ?clls* > *0*›
unfolding *InvariantCnCharacterization-def*
by (*simp* (*no-asm*))
then obtain *literal1*::*Literal* **and** *literal2*::*Literal*
where *literal1 el ?clls literal2 el ?clls literal1* ≠ *literal2*
using ‹*uniq ?clls*›
using ‹*?clls* ≠ []›
using *lengthGtOneTwoDistinctElements*[*of ?clls*]
by *auto*
then obtain *literal*::*Literal*
where *literal el ?clls literal* ≠ *opposite ?Cl*
using ‹*opposite ?Cl el ?clls*›
by *auto*
hence ¬ *isUIP* (*opposite ?Cl*) (*getC state*) (*getM state*)
using ‹*opposite ?Cl el ?clls*›
unfolding *isUIP-def*
by *auto*
}
moreover
{
assume *getCn state* = *1*
hence *length ?clls* = *1*
using ‹*InvariantCnCharacterization* (*getCn state*) (*getC state*) (*getM state*)›
unfolding *InvariantCnCharacterization-def*
by *auto*
{
fix *literal*::*Literal*
assume *literal el* (*getC state*) *literal* ≠ *opposite ?Cl*
have *elementLevel* (*opposite literal*) (*getM state*) < *currentLevel* (*getM state*)
proof−
have *elementLevel* (*opposite literal*) (*getM state*) ≤ *currentLevel* (*getM state*)
using *elementLevelLeqCurrentLevel*[*of opposite literal getM state*]
by *simp*
moreover
have *elementLevel* (*opposite literal*) (*getM state*) ≠ *currentLevel* (*getM state*)
proof−
{
assume ¬ *?thesis*
with ‹*literal el* (*getC state*)›
have *literal el ?clls*
by *simp*

hence *False*
using ‹*length ?clls* = *1*›
using ‹*opposite ?Cl el ?clls*›
using ‹*literal* ≠ *opposite ?Cl*›
using *lengthOneImpliesOnlyElement*[*of ?clls opposite ?Cl*]
by *auto*
}
thus *?thesis*
by *auto*
qed
ultimately
show *?thesis*
by *simp*
qed
}
hence *isUIP* (*opposite ?Cl*) (*getC state*) (*getM state*)
using ‹*isLastAssertedLiteral ?Cl* (*oppositeLiteralList* (*getC state*)) (*elements* (*getM state*))›
using ‹*opposite ?Cl el ?clls*›
unfolding *isUIP-def*
by *auto*
}
ultimately
show *?thesis*
by *auto*
qed

lemma *InvariantsClAfterApplyExplain*:
assumes
InvariantUniq (*getM state*)
InvariantCFalse (*getConflictFlag state*) (*getM state*) (*getC state*)
InvariantClCharacterization (*getCl state*) (*getC state*) (*getM state*)
InvariantClCurrentLevel (*getCl state*) (*getM state*)
InvariantCEntailed (*getConflictFlag state*) *F0* (*getC state*)
InvariantCnCharacterization (*getCn state*) (*getC state*) (*getM state*)
InvariantEquivalentZL (*getF state*) (*getM state*) *F0*
InvariantGetReasonIsReason (*getReason state*) (*getF state*) (*getM state*) (*set* (*getQ state*))
getCn state ≠ *1*
getConflictFlag state
currentLevel (*getM state*) > *0*
shows
let state′ = *applyExplain* (*getCl state*) *state in*
InvariantCFalse (*getConflictFlag state′*) (*getM state′*) (*getC state′*) ∧
InvariantCEntailed (*getConflictFlag state′*) *F0* (*getC state′*) ∧
InvariantClCharacterization (*getCl state′*) (*getC state′*) (*getM state′*) ∧

InvariantClCurrentLevel (*getCl state'*) (*getM state'*) ∧
InvariantCnCharacterization (*getCn state'*) (*getC state'*) (*getM state'*) ∧
InvariantUniqC (*getC state'*)
proof−
let *?Cl* = *getCl state*
let *?oppM0* = *oppositeLiteralList* (*elements* (*prefixToLevel 0* (*getM state*)))

have *isLastAssertedLiteral ?Cl* (*oppositeLiteralList* (*getC state*)) (*elements* (*getM state*))
using ‹*InvariantClCharacterization* (*getCl state*) (*getC state*) (*getM state*)›
unfolding *InvariantClCharacterization-def*
.
hence *literalTrue ?Cl* (*elements* (*getM state*)) *?Cl el* (*oppositeLiteralList* (*getC state*))
unfolding *isLastAssertedLiteral-def*
by *auto*
hence *opposite ?Cl el getC state*
using *literalElListIffOppositeLiteralElOppositeLiteralList*[*of opposite ?Cl getC state*]
by *simp*

have *clauseFalse* (*getC state*) (*elements* (*getM state*))
using ‹*getConflictFlag state*›
using ‹*InvariantCFalse* (*getConflictFlag state*) (*getM state*) (*getC state*)›
unfolding *InvariantCFalse-def*
by *simp*

have ¬ *isUIP* (*opposite ?Cl*) (*getC state*) (*getM state*)
using *CnEqual1IffUIP*[*of state*]
using *assms*
by *simp*

have ¬ *?Cl el* (*decisions* (*getM state*))
proof−
{
assume ¬ *?thesis*
hence *isUIP* (*opposite ?Cl*) (*getC state*) (*getM state*)
using ‹*InvariantUniq* (*getM state*)›
using ‹*isLastAssertedLiteral ?Cl* (*oppositeLiteralList* (*getC state*)) (*elements* (*getM state*))›
using ‹*clauseFalse* (*getC state*) (*elements* (*getM state*))›
using *lastDecisionThenUIP*[*of getM state opposite ?Cl getC state*]

unfolding *InvariantUniq-def*
by *simp*
with ‹¬ *isUIP* (*opposite ?Cl*) (*getC state*) (*getM state*)›
have *False*
by *simp*
} **thus** *?thesis*
by *auto*
qed

have *elementLevel ?Cl* (*getM state*) = *currentLevel* (*getM state*)
using ‹*InvariantClCurrentLevel* (*getCl state*) (*getM state*)›
unfolding *InvariantClCurrentLevel-def*
by *simp*
hence *elementLevel ?Cl* (*getM state*) > *0*
using ‹*currentLevel* (*getM state*) > *0*›
by *simp*

obtain *reason*
where *isReason* (*nth* (*getF state*) *reason*) *?Cl* (*elements* (*getM state*))
getReason state ?Cl = *Some reason 0* ≤ *reason* ∧ *reason* < *length* (*getF state*)
using ‹*InvariantGetReasonIsReason* (*getReason state*) (*getF state*) (*getM state*) (*set* (*getQ state*))›
unfolding *InvariantGetReasonIsReason-def*
using ‹*literalTrue ?Cl* (*elements* (*getM state*))›
using ‹¬ *?Cl el* (*decisions* (*getM state*))›
using ‹*elementLevel ?Cl* (*getM state*) > *0*›
by *auto*

let *?res* = *resolve* (*getC state*) (*getF state* ! *reason*) (*opposite ?Cl*)

obtain *ol*::*Literal*
where *ol el* (*getC state*)
ol ≠ *opposite ?Cl*
elementLevel (*opposite ol*) (*getM state*) ≥ *elementLevel ?Cl* (*getM state*)
using ‹*isLastAssertedLiteral ?Cl* (*oppositeLiteralList* (*getC state*)) (*elements* (*getM state*))›
using ‹¬ *isUIP* (*opposite ?Cl*) (*getC state*) (*getM state*)›
unfolding *isUIP-def*
by *auto*
hence *ol el ?res*
unfolding *resolve-def*
by *simp*
hence *?res* ≠ []
by *auto*
have *opposite ol el* (*oppositeLiteralList ?res*)
using ‹*ol el ?res*›

using *literalElListIffOppositeLiteralElOppositeLiteralList*[*of ol ?res*]
by *simp*

have *opposite ol el* (*oppositeLiteralList* (*getC state*))
using ‹*ol el* (*getC state*)›
using *literalElListIffOppositeLiteralElOppositeLiteralList*[*of ol getC state*]
by *simp*

have *literalFalse ol* (*elements* (*getM state*))
using ‹*clauseFalse* (*getC state*) (*elements* (*getM state*))›
using ‹*ol el getC state*›
by (*simp add*: *clauseFalseIffAllLiteralsAreFalse*)

have *elementLevel* (*opposite ol*) (*getM state*) = *elementLevel ?Cl* (*getM state*)
using ‹*elementLevel* (*opposite ol*) (*getM state*) ≥ *elementLevel ?Cl* (*getM state*)›
using ‹*isLastAssertedLiteral ?Cl* (*oppositeLiteralList* (*getC state*)) (*elements* (*getM state*))›
using *lastAssertedLiteralHasHighestElementLevel*[*of ?Cl oppositeLiteralList* (*getC state*) *getM state*]
using ‹*InvariantUniq* (*getM state*)›
unfolding *InvariantUniq-def*
using ‹*opposite ol el* (*oppositeLiteralList* (*getC state*))›
using ‹*literalFalse ol* (*elements* (*getM state*))›
by *auto*
hence *elementLevel* (*opposite ol*) (*getM state*) = *currentLevel* (*getM state*)
using ‹*elementLevel ?Cl* (*getM state*) = *currentLevel* (*getM state*)›
by *simp*

have *InvariantCFalse* (*getConflictFlag state*) (*getM state*) *?res*
using ‹*InvariantCFalse* (*getConflictFlag state*) (*getM state*) (*getC state*)›
using *InvariantCFalseAfterExplain*[*of getConflictFlag state getM state getC state ?Cl nth* (*getF state*) *reason ?res*]
using ‹*isReason* (*nth* (*getF state*) *reason*) *?Cl* (*elements* (*getM state*))›
using ‹*opposite ?Cl el* (*getC state*)›
by *simp*
hence *clauseFalse ?res* (*elements* (*getM state*))
using ‹*getConflictFlag state*›
unfolding *InvariantCFalse-def*
by *simp*

let *?rc* = *nth* (*getF state*) *reason*
let *?M0* = *elements* (*prefixToLevel 0* (*getM state*))
let *?F0* = (*getF state*) @ (*val2form ?M0*)

let *?C′ = list-diff ?res ?oppM0*
let *?C = remdups ?C′*

have *formulaEntailsClause (getF state) ?rc*
using ‹*0 ≤ reason ∧ reason < length (getF state)*›
using *nth-mem[of reason getF state]*
by (*simp add*: *formulaEntailsItsClauses*)
hence *formulaEntailsClause ?F0 ?rc*
by (*simp add*: *formulaEntailsClauseAppend*)

hence *formulaEntailsClause F0 ?rc*
using ‹*InvariantEquivalentZL (getF state) (getM state) F0*›
unfolding *InvariantEquivalentZL-def*
unfolding *formulaEntailsClause-def*
unfolding *equivalentFormulae-def*
by *simp*

hence *formulaEntailsClause F0 ?res*
using ‹*getConflictFlag state*›
using ‹*InvariantCEntailed (getConflictFlag state) F0 (getC state)*›
using *InvariantCEntailedAfterExplain[of getConflictFlag state F0 getC state nth (getF state) reason ?res getCl state]*
unfolding *InvariantCEntailed-def*
by *auto*
hence *formulaEntailsClause ?F0 ?res*
using ‹*InvariantEquivalentZL (getF state) (getM state) F0*›
unfolding *InvariantEquivalentZL-def*
unfolding *formulaEntailsClause-def*
unfolding *equivalentFormulae-def*
by *simp*

hence *formulaEntailsClause ?F0 ?C*
using *formulaEntailsClauseRemoveEntailedLiteralOpposites[of ?F0 ?res ?M0]*
using *val2formIsEntailed[of getF state ?M0 []]*
unfolding *formulaEntailsClause-def*
by (*auto simp add*: *clauseTrueIffContainsTrueLiteral*)

hence *formulaEntailsClause F0 ?C*
using ‹*InvariantEquivalentZL (getF state) (getM state) F0*›
unfolding *InvariantEquivalentZL-def*
unfolding *formulaEntailsClause-def*
unfolding *equivalentFormulae-def*
by *simp*

let *?ll = getLastAssertedLiteral (oppositeLiteralList ?res) (elements (getM state))*
have *isLastAssertedLiteral ?ll (oppositeLiteralList ?res) (elements (getM state))*

using ‹*?res* ≠ []›
using ‹*clauseFalse ?res* (*elements* (*getM state*))›
using ‹*InvariantUniq* (*getM state*)›
unfolding *InvariantUniq-def*
using *getLastAssertedLiteralCharacterization*[*of ?res elements* (*getM state*)]
by *simp*

hence *elementLevel* (*opposite ol*) (*getM state*) ≤ *elementLevel ?ll* (*getM state*)
using ‹*opposite ol el* (*oppositeLiteralList* (*getC state*))›
using *lastAssertedLiteralHasHighestElementLevel*[*of ?ll oppositeLiteralList ?res getM state*]
using ‹*InvariantUniq* (*getM state*)›
using ‹*opposite ol el* (*oppositeLiteralList ?res*)›
using ‹*literalFalse ol* (*elements* (*getM state*))›
unfolding *InvariantUniq-def*
by *simp*
hence *elementLevel ?ll* (*getM state*) = *currentLevel* (*getM state*)
using ‹*elementLevel* (*opposite ol*) (*getM state*) = *currentLevel* (*getM state*)›
using *elementLevelLeqCurrentLevel*[*of ?ll getM state*]
by *simp*

have *?ll el* (*oppositeLiteralList ?res*)
using ‹*isLastAssertedLiteral ?ll* (*oppositeLiteralList ?res*) (*elements* (*getM state*))›
unfolding *isLastAssertedLiteral-def*
by *simp*
hence *opposite ?ll el ?res*
using *literalElListIffOppositeLiteralElOppositeLiteralList*[*of opposite ?ll ?res*]
by *simp*

have ¬ *?ll el* (*elements* (*prefixToLevel 0* (*getM state*)))
proof−
{
assume ¬ *?thesis*
hence *elementLevel ?ll* (*getM state*) = *0*
using *prefixToLevelElementsElementLevel*[*of ?ll 0 getM state*]
by *simp*
hence *False*
using ‹*elementLevel ?ll* (*getM state*) = *currentLevel* (*getM state*)›
using ‹*currentLevel* (*getM state*) > *0*›
by *simp*
}
thus *?thesis*
by *auto*

qed
hence ¬ *opposite ?ll el ?oppM0*
using *literalElListIffOppositeLiteralElOppositeLiteralList*[*of ?ll elements* (*prefixToLevel 0* (*getM state*))]
by *simp*

have *opposite ?ll el ?C′*
using ‹*opposite ?ll el ?res*›
using ‹¬ *opposite ?ll el ?oppM0*›
using *listDiffIff*[*of opposite ?ll ?res ?oppM0*]
by *simp*
hence *?ll el* (*oppositeLiteralList ?C′*)
using *literalElListIffOppositeLiteralElOppositeLiteralList*[*of opposite ?ll ?C′*]
by *simp*

have *set* (*oppositeLiteralList ?C′*) ⊆ *set* (*oppositeLiteralList ?res*)
proof
fix *x*
assume *x* ∈ *set* (*oppositeLiteralList ?C′*)
thus *x* ∈ *set* (*oppositeLiteralList ?res*)
using *literalElListIffOppositeLiteralElOppositeLiteralList*[*of opposite x ?C′*]
using *literalElListIffOppositeLiteralElOppositeLiteralList*[*of opposite x ?res*]
using *listDiffIff*[*of opposite x ?res ?oppM0*]
by *auto*
qed

have *isLastAssertedLiteral ?ll* (*oppositeLiteralList ?C′*) (*elements* (*getM state*))
using ‹*?ll el* (*oppositeLiteralList ?C′*)›
using ‹*set* (*oppositeLiteralList ?C′*) ⊆ *set* (*oppositeLiteralList ?res*)›
using ‹*isLastAssertedLiteral ?ll* (*oppositeLiteralList ?res*) (*elements* (*getM state*))›
using *isLastAssertedLiteralSubset*[*of ?ll oppositeLiteralList ?res elements* (*getM state*) *oppositeLiteralList ?C′*]
by *auto*
moreover
have *set* (*oppositeLiteralList ?C*) = *set* (*oppositeLiteralList ?C′*)
unfolding *oppositeLiteralList-def*
by *simp*
ultimately
have *isLastAssertedLiteral ?ll* (*oppositeLiteralList ?C*) (*elements* (*getM state*))
unfolding *isLastAssertedLiteral-def*
by *auto*

hence *?ll el* (*oppositeLiteralList ?C*)
unfolding *isLastAssertedLiteral-def*
by *simp*
hence *opposite ?ll el ?C*
using *literalElListIffOppositeLiteralElOppositeLiteralList*[*of opposite ?ll ?C*]
by *simp*
hence *?C* ≠ []
by *auto*

have *clauseFalse ?C′* (*elements* (*getM state*))
proof−
{
fix *l*::*Literal*
assume *l el ?C′*
hence *l el ?res*
using *listDiffIff*[*of l ?res ?oppM0*]
by *simp*
hence *literalFalse l* (*elements* (*getM state*))
using ‹*clauseFalse ?res* (*elements* (*getM state*))›
by (*simp add*: *clauseFalseIffAllLiteralsAreFalse*)
}
thus *?thesis*
by (*simp add*: *clauseFalseIffAllLiteralsAreFalse*)
qed
hence *clauseFalse ?C* (*elements* (*getM state*))
by (*simp add*: *clauseFalseIffAllLiteralsAreFalse*)

let *?l′* = *getLastAssertedLiteral* (*oppositeLiteralList ?C*) (*elements* (*getM state*))
have *isLastAssertedLiteral ?l′* (*oppositeLiteralList ?C*) (*elements* (*getM state*))
using ‹*?C* ≠ []›
using ‹*clauseFalse ?C* (*elements* (*getM state*))›
using ‹*InvariantUniq* (*getM state*)›
unfolding *InvariantUniq-def*
using *getLastAssertedLiteralCharacterization*[*of ?C elements* (*getM state*)]
by *simp*
with ‹*isLastAssertedLiteral ?ll* (*oppositeLiteralList ?C*) (*elements* (*getM state*))›
have *?ll* = *?l′*
using *lastAssertedLiteralIsUniq*
by *simp*

show *?thesis*
using ‹*isLastAssertedLiteral ?l′* (*oppositeLiteralList ?C*) (*elements* (*getM state*))›
using ‹*?ll* = *?l′*›

using ‹*elementLevel ?ll* (*getM state*) = *currentLevel* (*getM state*)›
using ‹*getReason state ?Cl* = *Some reason*›
using ‹*clauseFalse ?C* (*elements* (*getM state*))›
using ‹*formulaEntailsClause F0 ?C*›
unfolding *applyExplain-def*
unfolding *InvariantCFalse-def*
unfolding *InvariantCEntailed-def*
unfolding *InvariantClCharacterization-def*
unfolding *InvariantClCurrentLevel-def*
unfolding *InvariantCnCharacterization-def*
unfolding *InvariantUniqC-def*
unfolding *setConflictAnalysisClause-def*
by (*simp add*: *findLastAssertedLiteral-def countCurrentLevelLiterals-def Let-def uniqDistinct distinct-remdups-id*)
qed

definition
multLessState = {(*state1*, *state2*). (*getM state1* = *getM state2*) ∧ (*getC state1*, *getC state2*) ∈ *multLess* (*getM state1*)}

lemma *ApplyExplainUIPTermination*:
assumes
InvariantUniq (*getM state*)
InvariantGetReasonIsReason (*getReason state*) (*getF state*) (*getM state*) (*set* (*getQ state*))
InvariantCFalse (*getConflictFlag state*) (*getM state*) (*getC state*)
InvariantClCurrentLevel (*getCl state*) (*getM state*)
InvariantClCharacterization (*getCl state*) (*getC state*) (*getM state*)
InvariantCnCharacterization (*getCn state*) (*getC state*) (*getM state*)
InvariantCEntailed (*getConflictFlag state*) *F0* (*getC state*)
InvariantEquivalentZL (*getF state*) (*getM state*) *F0*
getConflictFlag state
currentLevel (*getM state*) > *0*
shows
applyExplainUIP-dom state
using *assms*
proof (*induct rule*: *wf-induct*[*of multLessState*])
case *1*
thus *?case*
unfolding *wf-eq-minimal*
proof−
show ∀ *Q* (*state*::*State*). *state* ∈ *Q* ⟶ (∃ *stateMin* ∈ *Q*. ∀ *state*′. (*state*′, *stateMin*) ∈ *multLessState* ⟶ *state*′ ∉ *Q*)
proof−
{

fix *Q* :: *State set* **and** *state* :: *State*
assume *state* ∈ *Q*
let *?M* = (*getM state*)
let *?Q1* = {*C*::*Clause*. ∃ *state*. *state* ∈ *Q* ∧ (*getM state*) = *?M* ∧ (*getC state*) = *C*}
from ‹*state* ∈ *Q*›
have *getC state* ∈ *?Q1*
by *auto*
with *wfMultLess*[*of ?M*]
obtain *Cmin* **where** *Cmin* ∈ *?Q1* ∀ *C'*. (*C'*, *Cmin*) ∈ *multLess ?M* ⟶ *C'* ∉ *?Q1*
unfolding *wf-eq-minimal*
apply (*erule-tac x=?Q1* **in** *allE*)
apply (*erule-tac x=getC state* **in** *allE*)
by *auto*
from ‹*Cmin* ∈ *?Q1*› **obtain** *stateMin*
where *stateMin* ∈ *Q* (*getM stateMin*) = *?M getC stateMin* = *Cmin*
by *auto*
have ∀ *state'*. (*state'*, *stateMin*) ∈ *multLessState* ⟶ *state'* ∉ *Q*
proof
fix *state'*
show (*state'*, *stateMin*) ∈ *multLessState* ⟶ *state'* ∉ *Q*
proof
assume (*state'*, *stateMin*) ∈ *multLessState*
with ‹*getM stateMin* = *?M*›
have *getM state'* = *getM stateMin* (*getC state'*, *getC stateMin*) ∈ *multLess ?M*
unfolding *multLessState-def*
by *auto*
from ‹∀ *C'*. (*C'*, *Cmin*) ∈ *multLess ?M* ⟶ *C'* ∉ *?Q1*› ‹(*getC state'*, *getC stateMin*) ∈ *multLess ?M*› ‹*getC stateMin* = *Cmin*›
have *getC state'* ∉ *?Q1*
by *simp*
with ‹*getM state'* = *getM stateMin*› ‹*getM stateMin* = *?M*›
show *state'* ∉ *Q*
by *auto*
qed
qed
with ‹*stateMin* ∈ *Q*›
have ∃ *stateMin* ∈ *Q*. (∀ *state'*. (*state'*, *stateMin*) ∈ *multLessState* ⟶ *state'* ∉ *Q*)
by *auto*
}
thus *?thesis*
by *auto*
qed
qed

next
case (*2 state'*)
note *ih* = *this*
show *?case*
proof (*cases getCn state'* = *1*)
case *True*
show *?thesis*
apply (*rule applyExplainUIP-dom.intros*)
using *True*
by *simp*
next
case *False*
let *?state''* = *applyExplain* (*getCl state'*) *state'*
have *InvariantGetReasonIsReason* (*getReason ?state''*) (*getF ?state''*) (*getM ?state''*) (*set* (*getQ ?state''*))
InvariantUniq (*getM ?state''*)
InvariantEquivalentZL (*getF ?state''*) (*getM ?state''*) *F0*
getConflictFlag ?state''
currentLevel (*getM ?state''*) > *0*
using *ih*
unfolding *applyExplain-def*
unfolding *setConflictAnalysisClause-def*
by (*auto split*: *option.split simp add*: *findLastAssertedLiteral-def countCurrentLevelLiterals-def Let-def*)
moreover
have *InvariantCFalse* (*getConflictFlag ?state''*) (*getM ?state''*) (*getC ?state''*)
InvariantClCharacterization (*getCl ?state''*) (*getC ?state''*) (*getM ?state''*)
InvariantCnCharacterization (*getCn ?state''*) (*getC ?state''*) (*getM ?state''*)
InvariantClCurrentLevel (*getCl ?state''*) (*getM ?state''*)
InvariantCEntailed (*getConflictFlag ?state''*) *F0* (*getC ?state''*)
using *InvariantsClAfterApplyExplain*[*of state' F0*]
using *ih*
using *False*
by (*auto simp add*:*Let-def*)
moreover
have (*?state''*, *state'*) ∈ *multLessState*
proof−
have *getM ?state''* = *getM state'*
unfolding *applyExplain-def*
unfolding *setConflictAnalysisClause-def*
by (*auto split*: *option.split simp add*: *findLastAssertedLiteral-def countCurrentLevelLiterals-def Let-def*)

let *?Cl* = *getCl state'*
let *?oppM0* = *oppositeLiteralList* (*elements* (*prefixToLevel 0* (*getM state'*)))

have *isLastAssertedLiteral ?Cl* (*oppositeLiteralList* (*getC state'*)) (*elements* (*getM state'*))
using *ih*
unfolding *InvariantClCharacterization-def*
by *simp*
hence *literalTrue ?Cl* (*elements* (*getM state'*)) *?Cl el* (*oppositeLiteralList* (*getC state'*))
unfolding *isLastAssertedLiteral-def*
by *auto*
hence *opposite ?Cl el getC state'*
using *literalElListIffOppositeLiteralElOppositeLiteralList*[*of opposite ?Cl getC state'*]
by *simp*

have *clauseFalse* (*getC state'*) (*elements* (*getM state'*))
using *ih*
unfolding *InvariantCFalse-def*
by *simp*

have ¬ *?Cl el* (*decisions* (*getM state'*))
proof−
{
assume ¬ *?thesis*
hence *isUIP* (*opposite ?Cl*) (*getC state'*) (*getM state'*)
using *ih*
using ‹*isLastAssertedLiteral ?Cl* (*oppositeLiteralList* (*getC state'*)) (*elements* (*getM state'*))›
using ‹*clauseFalse* (*getC state'*) (*elements* (*getM state'*))›
using *lastDecisionThenUIP*[*of getM state' opposite ?Cl getC state'*]
unfolding *InvariantUniq-def*
unfolding *isUIP-def*
by *simp*
with ‹*getCn state'* ≠ *1*›
have *False*
using *CnEqual1IffUIP*[*of state'*]
using *ih*
by *simp*
} **thus** *?thesis*
by *auto*
qed

have *elementLevel ?Cl* (*getM state'*) = *currentLevel* (*getM state'*)
using *ih*
unfolding *InvariantClCurrentLevel-def*
by *simp*
hence *elementLevel ?Cl* (*getM state'*) > *0*
using *ih*

by *simp*

obtain *reason*
where *isReason* (*nth* (*getF state′*) *reason*) *?Cl* (*elements* (*getM state′*))
getReason state′ ?Cl = *Some reason* *0* ≤ *reason* ∧ *reason* < *length* (*getF state′*)
using *ih*
unfolding *InvariantGetReasonIsReason-def*
using ‹*literalTrue ?Cl* (*elements* (*getM state′*))›
using ‹¬ *?Cl el* (*decisions* (*getM state′*))›
using ‹*elementLevel ?Cl* (*getM state′*) > *0*›
by *auto*

let *?res* = *resolve* (*getC state′*) (*getF state′* ! *reason*) (*opposite ?Cl*)

have *getC ?state′′* = (*remdups* (*list-diff ?res ?oppM0*))
unfolding *applyExplain-def*
unfolding *setConflictAnalysisClause-def*
using ‹*getReason state′ ?Cl* = *Some reason*›
by (*simp add*: *Let-def findLastAssertedLiteral-def countCurrentLevelLiterals-def*)

have (*?res*, *getC state′*) ∈ *multLess* (*getM state′*)
using *multLessResolve*[*of ?Cl getC state′ nth* (*getF state′*) *reason getM state′*]
using ‹*opposite ?Cl el* (*getC state′*)›
using ‹*isReason* (*nth* (*getF state′*) *reason*) *?Cl* (*elements* (*getM state′*))›
by *simp*
hence (*list-diff ?res ?oppM0*, *getC state′*) ∈ *multLess* (*getM state′*)
by (*simp add*: *multLessListDiff*)

have (*remdups* (*list-diff ?res ?oppM0*), *getC state′*) ∈ *multLess* (*getM state′*)
using ‹(*list-diff ?res ?oppM0*, *getC state′*) ∈ *multLess* (*getM state′*)›
by (*simp add*: *multLessRemdups*)
thus *?thesis*
using ‹*getC ?state′′* = (*remdups* (*list-diff ?res ?oppM0*))›
using ‹*getM ?state′′* = *getM state′*›
unfolding *multLessState-def*
by *simp*
qed
ultimately
have *applyExplainUIP-dom ?state′′*
using *ih*
by *auto*

thus *?thesis*
using *applyExplainUIP-dom.intros*[*of state'*]
using *False*
by *simp*
qed
qed

lemma *ApplyExplainUIPPreservedVariables*:
assumes
applyExplainUIP-dom state
shows
let state' = applyExplainUIP state in
(*getM state' = getM state*) ∧
(*getF state' = getF state*) ∧
(*getQ state' = getQ state*) ∧
(*getWatch1 state' = getWatch1 state*) ∧
(*getWatch2 state' = getWatch2 state*) ∧
(*getWatchList state' = getWatchList state*) ∧
(*getConflictFlag state' = getConflictFlag state*) ∧
(*getConflictClause state' = getConflictClause state*) ∧
(*getSATFlag state' = getSATFlag state*) ∧
(*getReason state' = getReason state*)
(**is** *let state' = applyExplainUIP state in ?p state state'*)
using *assms*
proof(*induct state rule*: *applyExplainUIP-dom.induct*)
case (*step state'*)
note *ih = this*
show *?case*
proof (*cases getCn state' = 1*)
case *True*
with *applyExplainUIP.simps*[*of state'*]
have *applyExplainUIP state' = state'*
by *simp*
thus *?thesis*
by (*auto simp only*: *Let-def*)
next
case *False*
let *?state' = applyExplainUIP* (*applyExplain* (*getCl state'*) *state'*)
from *applyExplainUIP.simps*[*of state'*] *False*
have *applyExplainUIP state' = ?state'*
by (*simp add*: *Let-def*)
have *?p state'* (*applyExplain* (*getCl state'*) *state'*)
unfolding *applyExplain-def*
unfolding *setConflictAnalysisClause-def*
by (*auto split*: *option.split simp add*: *findLastAssertedLiteral-def countCurrentLevelLiterals-def Let-def*)
thus *?thesis*
using *ih*

using *False*
using ‹*applyExplainUIP state′ = ?state′*›
by (*simp add*: *Let-def*)
qed
qed

lemma *isUIPApplyExplainUIP*:
assumes *applyExplainUIP-dom state*
InvariantUniq (*getM state*)
InvariantCFalse (*getConflictFlag state*) (*getM state*) (*getC state*)
InvariantCEntailed (*getConflictFlag state*) *F0* (*getC state*)
InvariantClCharacterization (*getCl state*) (*getC state*) (*getM state*)
InvariantCnCharacterization (*getCn state*) (*getC state*) (*getM state*)
InvariantClCurrentLevel (*getCl state*) (*getM state*)
InvariantGetReasonIsReason (*getReason state*) (*getF state*) (*getM state*) (*set* (*getQ state*))
InvariantEquivalentZL (*getF state*) (*getM state*) *F0*
getConflictFlag state
currentLevel (*getM state*) > *0*
shows *let state′* = (*applyExplainUIP state*) *in*
isUIP (*opposite* (*getCl state′*)) (*getC state′*) (*getM state′*)
using *assms*
proof(*induct state rule*: *applyExplainUIP-dom.induct*)
case (*step state′*)
note *ih* = *this*
show *?case*
proof (*cases getCn state′* = *1*)
case *True*
with *applyExplainUIP.simps*[*of state′*]
have *applyExplainUIP state′* = *state′*
by *simp*
thus *?thesis*
using *ih*
using *CnEqual1IffUIP*[*of state′*]
using *True*
by (*simp add*: *Let-def*)
next
case *False*
let *?state′′* = *applyExplain* (*getCl state′*) *state′*
let *?state′* = *applyExplainUIP ?state′′*
from *applyExplainUIP.simps*[*of state′*] *False*
have *applyExplainUIP state′* = *?state′*
by (*simp add*: *Let-def*)
moreover
have *InvariantUniq* (*getM ?state′′*)
InvariantGetReasonIsReason (*getReason ?state′′*) (*getF ?state′′*) (*getM ?state′′*) (*set* (*getQ ?state′′*))
InvariantEquivalentZL (*getF ?state′′*) (*getM ?state′′*) *F0*
getConflictFlag ?state′′

currentLevel (*getM ?state''*) > *0*
using *ih*
unfolding *applyExplain-def*
unfolding *setConflictAnalysisClause-def*
by (*auto split*: *option.split simp add*: *findLastAssertedLiteral-def countCurrentLevelLiterals-def Let-def*)
moreover
have *InvariantCFalse* (*getConflictFlag ?state''*) (*getM ?state''*) (*getC ?state''*)
InvariantCEntailed (*getConflictFlag ?state''*) *F0* (*getC ?state''*)
InvariantClCharacterization (*getCl ?state''*) (*getC ?state''*) (*getM ?state''*)
InvariantCnCharacterization (*getCn ?state''*) (*getC ?state''*) (*getM ?state''*)
InvariantClCurrentLevel (*getCl ?state''*) (*getM ?state''*)
using *False*
using *ih*
using *InvariantsClAfterApplyExplain*[*of state' F0*]
by (*auto simp add*: *Let-def*)
ultimately
show *?thesis*
using *ih*(*2*)
using *False*
by (*simp add*: *Let-def*)
qed
qed

lemma *InvariantsClAfterExplainUIP*:
assumes
applyExplainUIP-dom state
InvariantUniq (*getM state*)
InvariantCFalse (*getConflictFlag state*) (*getM state*) (*getC state*)
InvariantCEntailed (*getConflictFlag state*) *F0* (*getC state*)
InvariantClCharacterization (*getCl state*) (*getC state*) (*getM state*)
InvariantCnCharacterization (*getCn state*) (*getC state*) (*getM state*)
InvariantClCurrentLevel (*getCl state*) (*getM state*)
InvariantUniqC (*getC state*)
InvariantGetReasonIsReason (*getReason state*) (*getF state*) (*getM state*) (*set* (*getQ state*))
InvariantEquivalentZL (*getF state*) (*getM state*) *F0*
getConflictFlag state
currentLevel (*getM state*) > *0*
shows
let state' = applyExplainUIP state in
InvariantCFalse (*getConflictFlag state'*) (*getM state'*) (*getC state'*) ∧
InvariantCEntailed (*getConflictFlag state'*) *F0* (*getC state'*) ∧
InvariantClCharacterization (*getCl state'*) (*getC state'*) (*getM*

```
state') ∧
       InvariantCnCharacterization (getCn state') (getC state') (getM
state') ∧
      InvariantClCurrentLevel (getCl state') (getM state') ∧
      InvariantUniqC (getC state')
using assms
proof(induct state rule: applyExplainUIP-dom.induct)
  case (step state')
  note ih = this
  show ?case
  proof (cases getCn state' = 1)
    case True
    with applyExplainUIP.simps[of state']
    have applyExplainUIP state' = state'
      by simp
    thus ?thesis
      using assms
      using ih
      by (auto simp only: Let-def)
  next
    case False
    let ?state'' = applyExplain (getCl state') state'
    let ?state' = applyExplainUIP ?state''
    from applyExplainUIP.simps[of state'] False
    have applyExplainUIP state' = ?state'
      by (simp add: Let-def)
    moreover
    have InvariantUniq (getM ?state'')
       InvariantGetReasonIsReason (getReason ?state'') (getF ?state'')
(getM ?state'') (set (getQ ?state''))
      InvariantEquivalentZL (getF ?state'') (getM ?state'') F0
      getConflictFlag ?state''
      currentLevel (getM ?state'') > 0
      using ih
      unfolding applyExplain-def
      unfolding setConflictAnalysisClause-def
       by (auto split: option.split simp add: findLastAssertedLiteral-def
countCurrentLevelLiterals-def Let-def)
    moreover
     have InvariantCFalse (getConflictFlag ?state'') (getM ?state'')
(getC ?state'')
      InvariantCEntailed (getConflictFlag ?state'') F0 (getC ?state'')
     InvariantClCharacterization (getCl ?state'') (getC ?state'') (getM
?state'')
    InvariantCnCharacterization (getCn ?state'') (getC ?state'') (getM
?state'')
      InvariantClCurrentLevel (getCl ?state'') (getM ?state'')
      InvariantUniqC (getC ?state'')
      using False
```

```
      using ih
      using InvariantsClAfterApplyExplain[of state' F0]
      by (auto simp add: Let-def)
    ultimately
    show ?thesis
      using False
      using ih(2)
      by simp
  qed
qed
```

```
lemma oneElementSetCharacterization:
shows
(set l = {a}) = ((remdups l) = [a])
proof (induct l)
  case Nil
  thus ?case
    by simp
next
  case (Cons a' l')
  show ?case
  proof (cases l' = [])
    case True
    thus ?thesis
      by simp
  next
    case False
    then obtain b
      where b ∈ set l'
      by force
    show ?thesis
    proof
      assume set (a' # l') = {a}
      hence a' = a set l' ⊆ {a}
        by auto
      hence b = a
        using ‹b ∈ set l'›
        by auto
      hence {a} ⊆ set l'
        using ‹b ∈ set l'›
        by auto
      hence set l' = {a}
        using ‹set l' ⊆ {a}›
        by auto
      thus remdups (a' # l') = [a]
```

```
      using ‹a' = a›
      using Cons
      by simp
  next
    assume remdups (a' # l') = [a]
    thus set (a' # l') = {a}
      using set-remdups[of a' # l']
      by auto
  qed
 qed
qed

lemma uniqOneElementCharacterization:
assumes
  uniq l
shows
  (l = [a]) = (set l = {a})
using assms
using uniqDistinct[of l]
using oneElementSetCharacterization[of l a]
using distinct-remdups-id[of l]
by auto

lemma isMinimalBackjumpLevelGetBackjumpLevel:
assumes
  InvariantUniq (getM state)
  InvariantCFalse (getConflictFlag state) (getM state) (getC state)
  InvariantClCharacterization (getCl state) (getC state) (getM state)
  InvariantCllCharacterization (getCl state) (getCll state) (getC state)
(getM state)
  InvariantClCurrentLevel (getCl state) (getM state)
  InvariantUniqC (getC state)

  getConflictFlag state
  isUIP (opposite (getCl state)) (getC state) (getM state)
  currentLevel (getM state) > 0
shows
  isMinimalBackjumpLevel (getBackjumpLevel state) (opposite (getCl
state)) (getC state) (getM state)
proof−
  let ?oppC = oppositeLiteralList (getC state)
  let ?Cl = getCl state

  have isLastAssertedLiteral ?Cl ?oppC (elements (getM state))
   using ‹InvariantClCharacterization (getCl state) (getC state) (getM
state)›
    unfolding InvariantClCharacterization-def
    by simp
```

have *elementLevel ?Cl* (*getM state*) > *0*
using ‹*InvariantClCurrentLevel* (*getCl state*) (*getM state*)›
using ‹*currentLevel* (*getM state*) > *0*›
unfolding *InvariantClCurrentLevel-def*
by *simp*

have *clauseFalse* (*getC state*) (*elements* (*getM state*))
using ‹*getConflictFlag state*›
using ‹*InvariantCFalse* (*getConflictFlag state*) (*getM state*) (*getC state*)›
unfolding *InvariantCFalse-def*
by *simp*

show *?thesis*
proof (*cases getC state* = [*opposite ?Cl*])
case *True*
thus *?thesis*
using *backjumpLevelZero*[*of opposite ?Cl oppositeLiteralList ?oppC getM state*]
using ‹*isLastAssertedLiteral ?Cl ?oppC* (*elements* (*getM state*))›
using *True*
using ‹*elementLevel ?Cl* (*getM state*) > *0*›
unfolding *getBackjumpLevel-def*
unfolding *isMinimalBackjumpLevel-def*
by (*simp add*: *Let-def*)
next
let *?Cll* = *getCll state*
case *False*
with ‹*InvariantCllCharacterization* (*getCl state*) (*getCll state*) (*getC state*) (*getM state*)›
‹*InvariantUniqC* (*getC state*)›
have *isLastAssertedLiteral ?Cll* (*removeAll ?Cl ?oppC*) (*elements* (*getM state*))
unfolding *InvariantCllCharacterization-def*
unfolding *InvariantUniqC-def*
using *uniqOneElementCharacterization*[*of getC state opposite ?Cl*]
by *simp*
hence *?Cll el ?oppC ?Cll* ≠ *?Cl*
unfolding *isLastAssertedLiteral-def*
by *auto*
hence *opposite ?Cll el* (*getC state*)
using *literalElListIffOppositeLiteralElOppositeLiteralList*[*of ?Cll ?oppC*]
by *auto*

show *?thesis*
using *backjumpLevelLastLast*[*of opposite ?Cl getC state getM state opposite ?Cll*]

using ‹*isUIP* (*opposite* (*getCl state*)) (*getC state*) (*getM state*)›
using ‹*clauseFalse* (*getC state*) (*elements* (*getM state*))›
using ‹*isLastAssertedLiteral ?Cll* (*removeAll ?Cl ?oppC*) (*elements* (*getM state*))›
using ‹*InvariantUniq* (*getM state*)›
using ‹*InvariantUniqC* (*getC state*)›
using *uniqOneElementCharacterization*[*of getC state opposite ?Cl*]
unfolding *InvariantUniqC-def*
unfolding *InvariantUniq-def*
using *False*
using ‹*opposite ?Cll el* (*getC state*)›
unfolding *getBackjumpLevel-def*
unfolding *isMinimalBackjumpLevel-def*
by (*auto simp add*: *Let-def*)
qed
qed

lemma *applyLearnPreservedVariables*:
let state′ = *applyLearn state in*
getM state′ = *getM state* ∧
getQ state′ = *getQ state* ∧
getC state′ = *getC state* ∧
getCl state′ = *getCl state* ∧
getConflictFlag state′ = *getConflictFlag state* ∧
getConflictClause state′ = *getConflictClause state* ∧
getF state′ = (*if getC state* = [*opposite* (*getCl state*)] *then*
getF state
else
(*getF state* @ [*getC state*])
)
proof (*cases getC state* = [*opposite* (*getCl state*)])
case *True*
thus *?thesis*
unfolding *applyLearn-def*
unfolding *setWatch1-def*
unfolding *setWatch2-def*
by (*simp add*:*Let-def*)
next
case *False*
thus *?thesis*
unfolding *applyLearn-def*
unfolding *setWatch1-def*
unfolding *setWatch2-def*

by (*simp add*:*Let-def*)
qed

lemma *WatchInvariantsAfterApplyLearn*:
assumes
InvariantUniq (*getM state*) **and**
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
and
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*) **and**
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantClCharacterization (*getCl state*) (*getC state*) (*getM state*)
and

getConflictFlag state
InvariantCFalse (*getConflictFlag state*) (*getM state*) (*getC state*)
InvariantUniqC (*getC state*)
shows
let state′ = (*applyLearn state*) *in*
InvariantWatchesEl (*getF state′*) (*getWatch1 state′*) (*getWatch2 state′*) ∧
InvariantWatchesDiffer (*getF state′*) (*getWatch1 state′*) (*getWatch2 state′*) ∧
InvariantWatchCharacterization (*getF state′*) (*getWatch1 state′*) (*getWatch2 state′*) (*getM state′*) ∧
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state′*) (*getF state′*) ∧
InvariantWatchListsUniq (*getWatchList state′*) ∧
InvariantWatchListsCharacterization (*getWatchList state′*) (*getWatch1 state′*) (*getWatch2 state′*)
proof (*cases getC state* ≠ [*opposite* (*getCl state*)])
case *False*
thus *?thesis*
using *assms*
unfolding *applyLearn-def*
unfolding *InvariantCllCharacterization-def*
by (*simp add*: *Let-def*)
next
case *True*

let *?oppC* = *oppositeLiteralList* (*getC state*)
let *?l* = *getCl state*
let *?ll* = *getLastAssertedLiteral* (*removeAll ?l ?oppC*) (*elements*

```
(getM state))

  have clauseFalse (getC state) (elements (getM state))
    using ‹getConflictFlag state›
    using ‹InvariantCFalse (getConflictFlag state) (getM state) (getC
state)›
    unfolding InvariantCFalse-def
    by simp


  from True
  have set (getC state) ≠ {opposite ?l}
    using ‹InvariantUniqC (getC state)›
    using uniqOneElementCharacterization[of getC state opposite ?l]
    unfolding InvariantUniqC-def
    by (simp add: Let-def)


  have isLastAssertedLiteral ?l ?oppC (elements (getM state))
   using ‹InvariantClCharacterization (getCl state) (getC state) (getM
state)›
    unfolding InvariantClCharacterization-def
    by simp

  have opposite ?l el (getC state)
    using ‹isLastAssertedLiteral ?l ?oppC (elements (getM state))›
    unfolding isLastAssertedLiteral-def
   using literalElListIffOppositeLiteralElOppositeLiteralList[of ?l ?oppC]
    by simp

  have removeAll ?l ?oppC ≠ []
  proof−
    {
      assume ¬ ?thesis
      hence set ?oppC ⊆ {?l}
        using set-removeAll[of ?l ?oppC]
        by auto
      have set (getC state) ⊆ {opposite ?l}
      proof
        fix x
        assume x ∈ set (getC state)
        hence opposite x ∈ set ?oppC
          using literalElListIffOppositeLiteralElOppositeLiteralList[of x
getC state]
          by simp
        hence opposite x ∈ {?l}
          using ‹set ?oppC ⊆ {?l}›
          by auto
        thus x ∈ {opposite ?l}
```

using *oppositeSymmetry*[*of x ?l*]
by *force*
qed
hence *False*
using ‹*set* (*getC state*) ≠ {*opposite ?l*}›
using ‹*opposite ?l el getC state*›
by (*auto simp add*: *Let-def*)
} **thus** *?thesis*
by *auto*
qed

have *clauseFalse* (*oppositeLiteralList* (*removeAll ?l ?oppC*)) (*elements* (*getM state*))
using ‹*clauseFalse* (*getC state*) (*elements* (*getM state*))›
using *oppositeLiteralListRemove*[*of ?l ?oppC*]
by (*simp add*: *clauseFalseIffAllLiteralsAreFalse*)
moreover
have *oppositeLiteralList* (*removeAll ?l ?oppC*) ≠ []
using ‹*removeAll ?l ?oppC* ≠ []›
using *oppositeLiteralListNonempty*
by *simp*
ultimately
have *isLastAssertedLiteral ?ll* (*removeAll ?l ?oppC*) (*elements* (*getM state*))
using ‹*InvariantUniq* (*getM state*)›
unfolding *InvariantUniq-def*
using *getLastAssertedLiteralCharacterization*[*of oppositeLiteralList* (*removeAll ?l ?oppC*) *elements* (*getM state*)]
by *auto*
hence *?ll el* (*removeAll ?l ?oppC*)
unfolding *isLastAssertedLiteral-def*
by *auto*
hence *?ll el ?oppC ?ll* ≠ *?l*
by *auto*
hence *opposite ?ll el* (*getC state*)
using *literalElListIffOppositeLiteralElOppositeLiteralList*[*of ?ll ?oppC*]
by *auto*

let *?state′* = *applyLearn state*

have *InvariantWatchesEl* (*getF ?state′*) (*getWatch1 ?state′*) (*getWatch2 ?state′*)
proof−
{
fix *clause*::*nat*
assume *0* ≤ *clause* ∧ *clause* < *length* (*getF ?state′*)
have ∃ *w1 w2*. *getWatch1 ?state′ clause* = *Some w1* ∧
getWatch2 ?state′ clause = *Some w2* ∧
w1 el (*getF ?state′* ! *clause*) ∧ *w2 el* (*getF ?state′* !

clause)
proof (*cases clause* < *length* (*getF state*))
case *True*
thus *?thesis*
using ‹*InvariantWatchesEl* (*getF state*) (*getWatch1 state*) (*getWatch2 state*)›
unfolding *InvariantWatchesEl-def*
using ‹*set* (*getC state*) ≠ {*opposite ?l*}›
unfolding *applyLearn-def*
unfolding *setWatch1-def*
unfolding *setWatch2-def*
by (*auto simp add*:*Let-def nth-append*)
next
case *False*
with ‹*0* ≤ *clause* ∧ *clause* < *length* (*getF ?state'*)›
have *clause* = *length* (*getF state*)
using ‹*getC state* ≠ [*opposite ?l*]›
unfolding *applyLearn-def*
unfolding *setWatch1-def*
unfolding *setWatch2-def*
by (*auto simp add*: *Let-def*)
moreover
have *getWatch1 ?state' clause* = *Some* (*opposite ?l*) *getWatch2 ?state' clause* = *Some* (*opposite ?ll*)
using ‹*clause* = *length* (*getF state*)›
using ‹*set* (*getC state*) ≠ {*opposite ?l*}›
unfolding *applyLearn-def*
unfolding *setWatch1-def*
unfolding *setWatch2-def*
by (*auto simp add*: *Let-def*)
moreover
have *getF ?state'* ! *clause* = (*getC state*)
using ‹*clause* = *length* (*getF state*)›
using ‹*set* (*getC state*) ≠ {*opposite ?l*}›
unfolding *applyLearn-def*
unfolding *setWatch1-def*
unfolding *setWatch2-def*
by (*auto simp add*: *Let-def*)
ultimately
show *?thesis*
using ‹*opposite ?l el* (*getC state*)› ‹*opposite ?ll el* (*getC state*)›
by *force*
qed
} **thus** *?thesis*
unfolding *InvariantWatchesEl-def*
by *auto*
qed
moreover
have *InvariantWatchesDiffer* (*getF ?state'*) (*getWatch1 ?state'*) (*getWatch2*

```
?state')
  proof−
    {
      fix clause::nat
      assume 0 ≤ clause ∧ clause < length (getF ?state')
      have  getWatch1 ?state' clause ≠ getWatch2 ?state' clause
      proof (cases clause < length (getF state))
        case True
        thus ?thesis
          using ‹InvariantWatchesDiffer (getF state) (getWatch1 state)
(getWatch2 state)›
          unfolding InvariantWatchesDiffer-def
          using ‹set (getC state) ≠ {opposite ?l}›
          unfolding applyLearn-def
          unfolding setWatch1-def
          unfolding setWatch2-def
          by (auto simp add:Let-def nth-append)
      next
        case False
        with  ‹0 ≤ clause ∧ clause < length (getF ?state')›
        have clause = length (getF state)
          using ‹getC state ≠ [opposite ?l]›
          unfolding applyLearn-def
          unfolding setWatch1-def
          unfolding setWatch2-def
          by (auto simp add: Let-def)
        moreover
        have getWatch1 ?state' clause = Some (opposite ?l) getWatch2
?state' clause = Some (opposite ?ll)
          using ‹clause = length (getF state)›
          using ‹set (getC state) ≠ {opposite ?l}›
          unfolding applyLearn-def
          unfolding setWatch1-def
          unfolding setWatch2-def
          by (auto simp add: Let-def)
        moreover
        have getF ?state' ! clause = (getC state)
          using ‹clause = length (getF state)›
          using ‹set (getC state) ≠ {opposite ?l}›
          unfolding applyLearn-def
          unfolding setWatch1-def
          unfolding setWatch2-def
          by (auto simp add: Let-def)
        ultimately
        show ?thesis
          using ‹?ll ≠ ?l›
          by force
      qed
    } thus ?thesis
```

unfolding *InvariantWatchesDiffer-def*
by *auto*
qed
moreover
have *InvariantWatchCharacterization* (*getF ?state′*) (*getWatch1 ?state′*) (*getWatch2 ?state′*) (*getM ?state′*)
proof−
{
fix *clause*::*nat* **and** *w1*::*Literal* **and** *w2*::*Literal*
assume ∗: *0* ≤ *clause* ∧ *clause* < *length* (*getF ?state′*)
assume ∗∗: *Some w1* = *getWatch1 ?state′ clause Some w2* = *getWatch2 ?state′ clause*
have *watchCharacterizationCondition w1 w2* (*getM ?state′*) (*getF ?state′* ! *clause*) ∧
watchCharacterizationCondition w2 w1 (*getM ?state′*) (*getF ?state′* ! *clause*)
proof (*cases clause* < *length* (*getF state*))
case *True*
thus *?thesis*
using ‹*InvariantWatchCharacterization* (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*)›
unfolding *InvariantWatchCharacterization-def*
using ‹*set* (*getC state*) ≠ {*opposite ?l*}›
using ∗∗
unfolding *applyLearn-def*
unfolding *setWatch1-def*
unfolding *setWatch2-def*
by (*auto simp add*:*Let-def nth-append*)
next
case *False*
with ‹*0* ≤ *clause* ∧ *clause* < *length* (*getF ?state′*)›
have *clause* = *length* (*getF state*)
using ‹*getC state* ≠ [*opposite ?l*]›
unfolding *applyLearn-def*
unfolding *setWatch1-def*
unfolding *setWatch2-def*
by (*auto simp add*: *Let-def*)
moreover
have *getWatch1 ?state′ clause* = *Some* (*opposite ?l*) *getWatch2 ?state′ clause* = *Some* (*opposite ?ll*)
using ‹*clause* = *length* (*getF state*)›
using ‹*set* (*getC state*) ≠ {*opposite ?l*}›
unfolding *applyLearn-def*
unfolding *setWatch1-def*
unfolding *setWatch2-def*
by (*auto simp add*: *Let-def*)
moreover
have ∀ *l*. *l el* (*getC state*) ∧ *l* ≠ *opposite ?l* ∧ *l* ≠ *opposite ?ll* ⟶

elementLevel (*opposite l*) (*getM state*) ≤ *elementLevel*
?l (*getM state*) ∧
elementLevel (*opposite l*) (*getM state*) ≤ *elementLevel*
?ll (*getM state*)
proof−
{
fix *l*
assume *l el* (*getC state*) *l* ≠ *opposite ?l l* ≠ *opposite ?ll*
hence *opposite l el ?oppC*
using *literalElListIffOppositeLiteralElOppositeLiteralList*[*of*
l getC state]
by *simp*
moreover
from ‹*l* ≠ *opposite ?l*›
have *opposite l* ≠ *?l*
using *oppositeSymmetry*[*of l ?l*]
by *blast*
ultimately
have *opposite l el* (*removeAll ?l ?oppC*)
by *simp*

from ‹*clauseFalse* (*getC state*) (*elements* (*getM state*))›
have *literalFalse l* (*elements* (*getM state*))
using ‹*l el* (*getC state*)›
by (*simp add*: *clauseFalseIffAllLiteralsAreFalse*)
hence *elementLevel* (*opposite l*) (*getM state*) ≤ *elementLevel*
?l (*getM state*) ∧
elementLevel (*opposite l*) (*getM state*) ≤ *elementLevel ?ll*
(*getM state*)
using ‹*InvariantUniq* (*getM state*)›
unfolding *InvariantUniq-def*
using ‹*isLastAssertedLiteral ?l ?oppC* (*elements* (*getM*
state))›
using *lastAssertedLiteralHasHighestElementLevel*[*of ?l*
?oppC getM state]
using ‹*isLastAssertedLiteral ?ll* (*removeAll ?l ?oppC*)
(*elements* (*getM state*))›
using *lastAssertedLiteralHasHighestElementLevel*[*of ?ll*
(*removeAll ?l ?oppC*) *getM state*]
using ‹*opposite l el ?oppC*› ‹*opposite l el* (*removeAll ?l*
?oppC)›
by *simp*
}
thus *?thesis*
by *simp*
qed
moreover
have *getF ?state′* ! *clause* = (*getC state*)
using ‹*clause* = *length* (*getF state*)›

```
        using ‹set (getC state) ≠ {opposite ?l}›
        unfolding applyLearn-def
        unfolding setWatch1-def
        unfolding setWatch2-def
        by (auto simp add: Let-def)
      moreover
      have getM ?state' = getM state
        using ‹set (getC state) ≠ {opposite ?l}›
        unfolding applyLearn-def
        unfolding setWatch1-def
        unfolding setWatch2-def
        by (auto simp add: Let-def)
      ultimately
      show ?thesis
        using ‹clauseFalse (getC state) (elements (getM state))›
        using **
        unfolding watchCharacterizationCondition-def
        by (auto simp add: clauseFalseIffAllLiteralsAreFalse)
    qed
  } thus ?thesis
    unfolding InvariantWatchCharacterization-def
    by auto
qed
moreover
have InvariantWatchListsContainOnlyClausesFromF (getWatchList
?state') (getF ?state')
proof−
  {
    fix clause::nat and literal::Literal
    assume clause ∈ set (getWatchList ?state' literal)
    have clause < length (getF ?state')
    proof(cases clause ∈ set (getWatchList state literal))
      case True
      thus ?thesis
     using ‹InvariantWatchListsContainOnlyClausesFromF (getWatchList
state) (getF state)›
       unfolding InvariantWatchListsContainOnlyClausesFromF-def
        using ‹set (getC state) ≠ {opposite ?l}›
        unfolding applyLearn-def
        unfolding setWatch1-def
        unfolding setWatch2-def
        by (auto simp add:Let-def nth-append) (force)+
    next
      case False
      with ‹clause ∈ set (getWatchList ?state' literal)›
      have clause = length (getF state)
        using ‹set (getC state) ≠ {opposite ?l}›
        unfolding applyLearn-def
        unfolding setWatch1-def
```

unfolding *setWatch2-def*
by (*auto simp add*:*Let-def nth-append split*: *if-split-asm*)
thus *?thesis*
using ‹*set* (*getC state*) ≠ {*opposite ?l*}›
unfolding *applyLearn-def*
unfolding *setWatch1-def*
unfolding *setWatch2-def*
by (*auto simp add*:*Let-def nth-append*)
qed
} **thus** *?thesis*
unfolding *InvariantWatchListsContainOnlyClausesFromF-def*
by *simp*
qed
moreover
have *InvariantWatchListsUniq* (*getWatchList ?state′*)
unfolding *InvariantWatchListsUniq-def*
proof
fix *l*::*Literal*
show *uniq* (*getWatchList ?state′ l*)
proof(*cases l* = *opposite ?l* ∨ *l* = *opposite ?ll*)
case *True*
hence *getWatchList ?state′ l* = (*length* (*getF state*)) # *getWatchList state l*
using ‹*set* (*getC state*) ≠ {*opposite ?l*}›
unfolding *applyLearn-def*
unfolding *setWatch1-def*
unfolding *setWatch2-def*
using ‹*?ll* ≠ *?l*›
by (*auto simp add*:*Let-def nth-append*)
moreover
have *length* (*getF state*) ∉ *set* (*getWatchList state l*)
using ‹*InvariantWatchListsContainOnlyClausesFromF* (*getWatchList state*) (*getF state*)›
unfolding *InvariantWatchListsContainOnlyClausesFromF-def*
by *auto*
ultimately
show *?thesis*
using ‹*InvariantWatchListsUniq* (*getWatchList state*)›
unfolding *InvariantWatchListsUniq-def*
by (*simp add*: *uniqAppendIff*)
next
case *False*
hence *getWatchList ?state′ l* = *getWatchList state l*
using ‹*set* (*getC state*) ≠ {*opposite ?l*}›
unfolding *applyLearn-def*
unfolding *setWatch1-def*
unfolding *setWatch2-def*
by (*auto simp add*:*Let-def nth-append*)
thus *?thesis*

```
        using ‹InvariantWatchListsUniq (getWatchList state)›
        unfolding InvariantWatchListsUniq-def
        by simp
    qed
  qed
  moreover
  have InvariantWatchListsCharacterization (getWatchList ?state') (getWatch1 ?state') (getWatch2 ?state')
  proof−
    {
      fix c::nat and l::Literal
      have (c ∈ set (getWatchList ?state' l)) = (Some l = getWatch1 ?state' c ∨ Some l = getWatch2 ?state' c)
      proof (cases c = length (getF state))
        case False
        thus ?thesis
            using ‹InvariantWatchListsCharacterization (getWatchList state) (getWatch1 state) (getWatch2 state)›
          unfolding InvariantWatchListsCharacterization-def
          using ‹set (getC state) ≠ {opposite ?l}›
          unfolding applyLearn-def
          unfolding setWatch1-def
          unfolding setWatch2-def
          by (auto simp add:Let-def nth-append)
      next
        case True
        have length (getF state) ∉ set (getWatchList state l)
        using ‹InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state)›
          unfolding InvariantWatchListsContainOnlyClausesFromF-def
          by auto
        thus ?thesis
          using ‹c = length (getF state)›
      using ‹InvariantWatchListsCharacterization (getWatchList state) (getWatch1 state) (getWatch2 state)›
        unfolding InvariantWatchListsCharacterization-def
        using ‹set (getC state) ≠ {opposite ?l}›
        unfolding applyLearn-def
        unfolding setWatch1-def
        unfolding setWatch2-def
        by (auto simp add:Let-def nth-append)
    qed
  } thus ?thesis
    unfolding InvariantWatchListsCharacterization-def
    by simp
  qed
  moreover
  have InvariantClCharacterization (getCl ?state') (getC ?state') (getM ?state')
```

using ‹*InvariantClCharacterization* (*getCl state*) (*getC state*) (*getM state*)›
using ‹*set* (*getC state*) ≠ {*opposite ?l*}›
unfolding *applyLearn-def*
unfolding *setWatch1-def*
unfolding *setWatch2-def*
by (*auto simp add*:*Let-def*)
moreover
have *InvariantCllCharacterization* (*getCl ?state′*) (*getCll ?state′*) (*getC ?state′*) (*getM ?state′*)
unfolding *InvariantCllCharacterization-def*
using ‹*isLastAssertedLiteral ?ll* (*removeAll ?l ?oppC*) (*elements* (*getM state*))›
using ‹*set* (*getC state*) ≠ {*opposite ?l*}›
unfolding *applyLearn-def*
unfolding *setWatch1-def*
unfolding *setWatch2-def*
by (*auto simp add*:*Let-def*)
ultimately
show *?thesis*
by *simp*
qed

lemma *InvariantCllCharacterizationAfterApplyLearn*:
assumes
InvariantUniq (*getM state*)
InvariantClCharacterization (*getCl state*) (*getC state*) (*getM state*)
InvariantCFalse (*getConflictFlag state*) (*getM state*) (*getC state*)
InvariantUniqC (*getC state*)
getConflictFlag state
shows
let state′ = *applyLearn state in*
InvariantCllCharacterization (*getCl state′*) (*getCll state′*) (*getC state′*) (*getM state′*)
proof (*cases getC state* ≠ [*opposite* (*getCl state*)])
case *False*
thus *?thesis*
using *assms*
unfolding *applyLearn-def*
unfolding *InvariantCllCharacterization-def*
by (*simp add*: *Let-def*)
next
case *True*

let *?oppC* = *oppositeLiteralList* (*getC state*)
let *?l* = *getCl state*
let *?ll* = *getLastAssertedLiteral* (*removeAll ?l ?oppC*) (*elements* (*getM state*))

have *clauseFalse* (*getC state*) (*elements* (*getM state*))
using ‹*getConflictFlag state*›
using ‹*InvariantCFalse* (*getConflictFlag state*) (*getM state*) (*getC state*)›
unfolding *InvariantCFalse-def*
by *simp*

from *True*
have *set* (*getC state*) ≠ {*opposite ?l*}
using ‹*InvariantUniqC* (*getC state*)›
using *uniqOneElementCharacterization*[*of getC state opposite ?l*]
unfolding *InvariantUniqC-def*
by (*simp add*: *Let-def*)

have *isLastAssertedLiteral ?l ?oppC* (*elements* (*getM state*))
using ‹*InvariantClCharacterization* (*getCl state*) (*getC state*) (*getM state*)›
unfolding *InvariantClCharacterization-def*
by *simp*

have *opposite ?l el* (*getC state*)
using ‹*isLastAssertedLiteral ?l ?oppC* (*elements* (*getM state*))›
unfolding *isLastAssertedLiteral-def*
using *literalElListIffOppositeLiteralElOppositeLiteralList*[*of ?l ?oppC*]
by *simp*

have *removeAll ?l ?oppC* ≠ []
proof−
{
assume ¬ *?thesis*
hence *set ?oppC* ⊆ {*?l*}
using *set-removeAll*[*of ?l ?oppC*]
by *auto*
have *set* (*getC state*) ⊆ {*opposite ?l*}
proof
fix *x*
assume *x* ∈ *set* (*getC state*)
hence *opposite x* ∈ *set ?oppC*
using *literalElListIffOppositeLiteralElOppositeLiteralList*[*of x getC state*]
by *simp*
hence *opposite x* ∈ {*?l*}
using ‹*set ?oppC* ⊆ {*?l*}›
by *auto*
thus *x* ∈ {*opposite ?l*}
using *oppositeSymmetry*[*of x ?l*]
by *force*
qed

hence *False*
using ‹*set* (*getC state*) ≠ {*opposite ?l*}›
using ‹*opposite ?l el getC state*›
by (*auto simp add*: *Let-def*)
} **thus** *?thesis*
by *auto*
qed

have *clauseFalse* (*oppositeLiteralList* (*removeAll ?l ?oppC*)) (*elements* (*getM state*))
using ‹*clauseFalse* (*getC state*) (*elements* (*getM state*))›
using *oppositeLiteralListRemove*[*of ?l ?oppC*]
by (*simp add*: *clauseFalseIffAllLiteralsAreFalse*)
moreover
have *oppositeLiteralList* (*removeAll ?l ?oppC*) ≠ []
using ‹*removeAll ?l ?oppC* ≠ []›
using *oppositeLiteralListNonempty*
by *simp*
ultimately
have *isLastAssertedLiteral ?ll* (*removeAll ?l ?oppC*) (*elements* (*getM state*))
using *getLastAssertedLiteralCharacterization*[*of oppositeLiteralList* (*removeAll ?l ?oppC*) *elements* (*getM state*)]
using ‹*InvariantUniq* (*getM state*)›
unfolding *InvariantUniq-def*
by *auto*
thus *?thesis*
using ‹*set* (*getC state*) ≠ {*opposite ?l*}›
unfolding *applyLearn-def*
unfolding *setWatch1-def*
unfolding *setWatch2-def*
unfolding *InvariantCllCharacterization-def*
by (*auto simp add*:*Let-def*)
qed

lemma *InvariantConflictClauseCharacterizationAfterApplyLearn*:
assumes
getConflictFlag state
InvariantConflictClauseCharacterization (*getConflictFlag state*) (*getConflictClause state*) (*getF state*) (*getM state*)
shows
let state′ = applyLearn state in
InvariantConflictClauseCharacterization (*getConflictFlag state′*) (*getConflictClause state′*) (*getF state′*) (*getM state′*)
proof−
have *getConflictClause state* < *length* (*getF state*)
using *assms*
unfolding *InvariantConflictClauseCharacterization-def*

by (*auto simp add*: *Let-def*)
hence *nth* ((*getF state*) @ [*getC state*]) (*getConflictClause state*) =
nth (*getF state*) (*getConflictClause state*)
by (*simp add*: *nth-append*)
thus *?thesis*
using ‹*InvariantConflictClauseCharacterization* (*getConflictFlag state*) (*getConflictClause state*) (*getF state*) (*getM state*)›
unfolding *InvariantConflictClauseCharacterization-def*
unfolding *applyLearn-def*
unfolding *setWatch1-def*
unfolding *setWatch2-def*
by (*auto simp add*: *Let-def clauseFalseAppendValuation*)
qed

lemma *InvariantGetReasonIsReasonAfterApplyLearn*:
assumes
InvariantGetReasonIsReason (*getReason state*) (*getF state*) (*getM state*) (*set* (*getQ state*))
shows
let state′ = *applyLearn state in*
InvariantGetReasonIsReason (*getReason state′*) (*getF state′*) (*getM state′*) (*set* (*getQ state′*))

proof (*cases getC state* = [*opposite* (*getCl state*)])
case *True*
thus *?thesis*
unfolding *applyLearn-def*
using *assms*
by (*simp add*: *Let-def*)
next
case *False*
have *InvariantGetReasonIsReason* (*getReason state*) ((*getF state*) @ [*getC state*]) (*getM state*) (*set* (*getQ state*))
using *assms*
using *nth-append*[*of getF state* [*getC state*]]
unfolding *InvariantGetReasonIsReason-def*
by *auto*
thus *?thesis*
using *False*
unfolding *applyLearn-def*
unfolding *setWatch1-def*
unfolding *setWatch2-def*
by (*simp add*: *Let-def*)
qed

lemma *InvariantQCharacterizationAfterApplyLearn*:
assumes
getConflictFlag state
InvariantQCharacterization (*getConflictFlag state*) (*getQ state*) (*getF*

state) (*getM state*)
shows
let state′ = applyLearn state in
InvariantQCharacterization (*getConflictFlag state′*) (*getQ state′*) (*getF state′*) (*getM state′*)
using *assms*
unfolding *InvariantQCharacterization-def*
unfolding *applyLearn-def*
unfolding *setWatch1-def*
unfolding *setWatch2-def*
by (*simp add*: *Let-def*)

lemma *InvariantUniqQAfterApplyLearn*:
assumes
InvariantUniqQ (*getQ state*)
shows
let state′ = applyLearn state in
InvariantUniqQ (*getQ state′*)
using *assms*
unfolding *applyLearn-def*
unfolding *setWatch1-def*
unfolding *setWatch2-def*
by (*simp add*: *Let-def*)

lemma *InvariantConflictFlagCharacterizationAfterApplyLearn*:
assumes
getConflictFlag state
InvariantConflictFlagCharacterization (*getConflictFlag state*) (*getF state*) (*getM state*)
shows
let state′ = applyLearn state in
InvariantConflictFlagCharacterization (*getConflictFlag state′*) (*getF state′*) (*getM state′*)
using *assms*
unfolding *InvariantConflictFlagCharacterization-def*
unfolding *applyLearn-def*
unfolding *setWatch1-def*
unfolding *setWatch2-def*
by (*auto simp add*: *Let-def formulaFalseIffContainsFalseClause*)

lemma *InvariantNoDecisionsWhenConflictNorUnitAfterApplyLearn*:
assumes
InvariantUniq (*getM state*)
InvariantConsistent (*getM state*)
InvariantNoDecisionsWhenConflict (*getF state*) (*getM state*) (*currentLevel* (*getM state*))
InvariantNoDecisionsWhenUnit (*getF state*) (*getM state*) (*currentLevel* (*getM state*))
InvariantCFalse (*getConflictFlag state*) (*getM state*) (*getC state*) **and**

InvariantClCharacterization (*getCl state*) (*getC state*) (*getM state*)
InvariantClCurrentLevel (*getCl state*) (*getM state*)
InvariantUniqC (*getC state*)

getConflictFlag state
isUIP (*opposite* (*getCl state*)) (*getC state*) (*getM state*)
currentLevel (*getM state*) > *0*
shows
let state' = *applyLearn state in*
InvariantNoDecisionsWhenConflict (*getF state*) (*getM state'*) (*currentLevel* (*getM state'*)) ∧
InvariantNoDecisionsWhenUnit (*getF state*) (*getM state'*) (*currentLevel* (*getM state'*)) ∧
InvariantNoDecisionsWhenConflict [*getC state*] (*getM state'*) (*getBackjumpLevel state'*) ∧
InvariantNoDecisionsWhenUnit [*getC state*] (*getM state'*) (*getBackjumpLevel state'*)
proof−
let *?state'* = *applyLearn state*
let *?l* = *getCl state*

have *clauseFalse* (*getC state*) (*elements* (*getM state*))
using ‹*getConflictFlag state*›
using ‹*InvariantCFalse* (*getConflictFlag state*) (*getM state*) (*getC state*)›
unfolding *InvariantCFalse-def*
by *simp*

have *getM ?state'* = *getM state getC ?state'* = *getC state*
getCl ?state' = *getCl state getConflictFlag ?state'* = *getConflictFlag state*
unfolding *applyLearn-def*
unfolding *setWatch2-def*
unfolding *setWatch1-def*
by (*auto simp add*: *Let-def*)

hence *InvariantNoDecisionsWhenConflict* (*getF state*) (*getM ?state'*) (*currentLevel* (*getM ?state'*)) ∧
InvariantNoDecisionsWhenUnit (*getF state*) (*getM ?state'*) (*currentLevel* (*getM ?state'*))
using ‹*InvariantNoDecisionsWhenConflict* (*getF state*) (*getM state*) (*currentLevel* (*getM state*))›
using ‹*InvariantNoDecisionsWhenUnit* (*getF state*) (*getM state*) (*currentLevel* (*getM state*))›
by *simp*
moreover
have *InvariantCllCharacterization* (*getCl ?state'*) (*getCll ?state'*) (*getC ?state'*) (*getM ?state'*)
using *assms*

using *InvariantCllCharacterizationAfterApplyLearn*[*of state*]
by (*simp add*: *Let-def*)
hence *isMinimalBackjumpLevel* (*getBackjumpLevel ?state'*) (*opposite ?l*) (*getC ?state'*) (*getM ?state'*)
using *assms*
using ‹*getM ?state' = getM state*› ‹*getC ?state' = getC state*›
‹*getCl ?state' = getCl state*› ‹*getConflictFlag ?state' = getConflictFlag state*›
using *isMinimalBackjumpLevelGetBackjumpLevel*[*of ?state'*]
unfolding *isUIP-def*
unfolding *SatSolverVerification.isUIP-def*
by (*simp add*: *Let-def*)
hence *getBackjumpLevel ?state' < elementLevel ?l* (*getM ?state'*)
unfolding *isMinimalBackjumpLevel-def*
unfolding *isBackjumpLevel-def*
by *simp*
hence *getBackjumpLevel ?state' < currentLevel* (*getM ?state'*)
using *elementLevelLeqCurrentLevel*[*of ?l getM ?state'*]
by *simp*

have *InvariantNoDecisionsWhenConflict* [*getC state*] (*getM ?state'*) (*getBackjumpLevel ?state'*) ∧
InvariantNoDecisionsWhenUnit [*getC state*] (*getM ?state'*) (*getBackjumpLevel ?state'*)
proof−
{
fix *clause*::*Clause*
assume *clause el* [*getC state*]
hence *clause = getC state*
by *simp*

have (∀ *level'*. *level'* < (*getBackjumpLevel ?state'*) ⟶
¬ *clauseFalse clause* (*elements* (*prefixToLevel level'* (*getM ?state'*)))) ∧
(∀ *level'*. *level'* < (*getBackjumpLevel ?state'*) ⟶
¬ (∃ *l*. *isUnitClause clause l* (*elements* (*prefixToLevel level'* (*getM ?state'*))))) (**is** *?false* ∧ *?unit*)
proof(*cases getC state* = [*opposite ?l*])
case *True*
thus *?thesis*
using ‹*getM ?state' = getM state*› ‹*getC ?state' = getC state*› ‹*getCl ?state' = getCl state*›
unfolding *getBackjumpLevel-def*
by (*simp add*: *Let-def*)
next
case *False*
hence *getF ?state' = getF state* @ [*getC state*]
unfolding *applyLearn-def*
unfolding *setWatch2-def*

unfolding *setWatch1-def*
by (*auto simp add*: *Let-def*)

show *?thesis*
proof−
have *?unit*
using ‹*clause* = *getC state*›
using ‹*InvariantUniq* (*getM state*)›
using ‹*InvariantConsistent* (*getM state*)›
using ‹*getM ?state′* = *getM state*› ‹*getC ?state′* = *getC state*›
using ‹*clauseFalse* (*getC state*) (*elements* (*getM state*))›
using ‹*isMinimalBackjumpLevel* (*getBackjumpLevel ?state′*) (*opposite ?l*) (*getC ?state′*) (*getM ?state′*)›
using *isMinimalBackjumpLevelEnsuresIsNotUnitBeforePrefix*[*of getM ?state′ getC ?state′ getBackjumpLevel ?state′ opposite ?l*]
unfolding *InvariantUniq-def*
unfolding *InvariantConsistent-def*
by *simp*
moreover
have *isUnitClause* (*getC state*) (*opposite ?l*) (*elements* (*prefixToLevel* (*getBackjumpLevel ?state′*) (*getM state*)))
using ‹*InvariantUniq* (*getM state*)›
using ‹*InvariantConsistent* (*getM state*)›
using ‹*isMinimalBackjumpLevel* (*getBackjumpLevel ?state′*) (*opposite ?l*) (*getC ?state′*) (*getM ?state′*)›
using ‹*getM ?state′* = *getM state*› ‹*getC ?state′* = *getC state*›
using ‹*clauseFalse* (*getC state*) (*elements* (*getM state*))›
using *isBackjumpLevelEnsuresIsUnitInPrefix*[*of getM ?state′ getC ?state′ getBackjumpLevel ?state′ opposite ?l*]
unfolding *isMinimalBackjumpLevel-def*
unfolding *InvariantUniq-def*
unfolding *InvariantConsistent-def*
by *simp*
hence ¬ *clauseFalse* (*getC state*) (*elements* (*prefixToLevel* (*getBackjumpLevel ?state′*) (*getM state*)))
unfolding *isUnitClause-def*
by (*auto simp add*: *clauseFalseIffAllLiteralsAreFalse*)
have *?false*
proof
fix *level′*
show *level′* < *getBackjumpLevel ?state′* ⟶ ¬ *clauseFalse clause* (*elements* (*prefixToLevel level′* (*getM ?state′*)))
proof
assume *level′* < *getBackjumpLevel ?state′*
show ¬ *clauseFalse clause* (*elements* (*prefixToLevel level′* (*getM ?state′*)))
proof−

have *isPrefix* (*prefixToLevel level′* (*getM state*)) (*prefixToLevel* (*getBackjumpLevel ?state′*) (*getM state*))
using ‹*level′* < *getBackjumpLevel ?state′*›
using *isPrefixPrefixToLevelLowerLevel*[*of level′ getBackjumpLevel ?state′ getM state*]
by *simp*
then obtain *s*
where *prefixToLevel level′* (*getM state*) @ *s* = *prefixToLevel* (*getBackjumpLevel ?state′*) (*getM state*)
unfolding *isPrefix-def*
by *auto*
hence *prefixToLevel* (*getBackjumpLevel ?state′*) (*getM state*) = *prefixToLevel level′* (*getM state*) @ *s*
by (*rule sym*)
thus *?thesis*
using ‹*getM ?state′* = *getM state*›
using ‹*clause* = *getC state*›
using ‹¬ *clauseFalse* (*getC state*) (*elements* (*prefixToLevel* (*getBackjumpLevel ?state′*) (*getM state*)))›
unfolding *isPrefix-def*
by (*auto simp add*: *clauseFalseIffAllLiteralsAreFalse*)
qed
qed
qed
ultimately
show *?thesis*
by *simp*
qed
qed
} **thus** *?thesis*
unfolding *InvariantNoDecisionsWhenConflict-def*
unfolding *InvariantNoDecisionsWhenUnit-def*
by (*auto simp add*: *formulaFalseIffContainsFalseClause*)
qed
ultimately
show *?thesis*
by (*simp add*: *Let-def*)
qed

lemma *InvariantEquivalentZLAfterApplyLearn*:
assumes
InvariantEquivalentZL (*getF state*) (*getM state*) *F0* **and**
InvariantCEntailed (*getConflictFlag state*) *F0* (*getC state*) **and**
getConflictFlag state
shows
let state′ = *applyLearn state in*
InvariantEquivalentZL (*getF state′*) (*getM state′*) *F0*
proof−
let *?M0* = *val2form* (*elements* (*prefixToLevel 0* (*getM state*)))

have *equivalentFormulae F0* (*getF state* @ *?M0*)
using ‹*InvariantEquivalentZL* (*getF state*) (*getM state*) *F0*›
using *equivalentFormulaeSymmetry*[*of F0 getF state* @ *?M0*]
unfolding *InvariantEquivalentZL-def*
by *simp*
moreover
have *formulaEntailsClause* (*getF state* @ *?M0*) (*getC state*)
using *assms*
unfolding *InvariantEquivalentZL-def*
unfolding *InvariantCEntailed-def*
unfolding *equivalentFormulae-def*
unfolding *formulaEntailsClause-def*
by *auto*
ultimately
have *equivalentFormulae F0* ((*getF state* @ *?M0*) @ [*getC state*])
using *extendEquivalentFormulaWithEntailedClause*[*of F0 getF state* @ *?M0 getC state*]
by *simp*
hence *equivalentFormulae* ((*getF state* @ *?M0*) @ [*getC state*]) *F0*
by (*simp add*: *equivalentFormulaeSymmetry*)
have *equivalentFormulae* ((*getF state*) @ [*getC state*] @ *?M0*) *F0*
proof−
{
fix *valuation*:: *Valuation*
have *formulaTrue* ((*getF state* @ *?M0*) @ [*getC state*]) *valuation* = *formulaTrue* ((*getF state*) @ [*getC state*] @ *?M0*) *valuation*
by (*simp add*: *formulaTrueIffAllClausesAreTrue*)
}
thus *?thesis*
using ‹*equivalentFormulae* ((*getF state* @ *?M0*) @ [*getC state*]) *F0*›
unfolding *equivalentFormulae-def*
by *auto*
qed
thus *?thesis*
using *assms*
unfolding *InvariantEquivalentZL-def*
unfolding *applyLearn-def*
unfolding *setWatch1-def*
unfolding *setWatch2-def*
by (*auto simp add*: *Let-def*)
qed

lemma *InvariantVarsFAfterApplyLearn*:
assumes
InvariantCFalse (*getConflictFlag state*) (*getM state*) (*getC state*)
getConflictFlag state
InvariantVarsF (*getF state*) *F0 Vbl*

InvariantVarsM (*getM state*) *F0 Vbl*
shows
let state′ = applyLearn state in
InvariantVarsF (*getF state′*) *F0 Vbl*

proof−
from *assms*
have *clauseFalse* (*getC state*) (*elements* (*getM state*))
unfolding *InvariantCFalse-def*
by *simp*
hence *vars* (*getC state*) ⊆ *vars* (*elements* (*getM state*))
using *valuationContainsItsFalseClausesVariables*[*of getC state elements* (*getM state*)]
by *simp*
thus *?thesis*
using *applyLearnPreservedVariables*[*of state*]
using *assms*
using *varsAppendFormulae*[*of getF state* [*getC state*]]
unfolding *InvariantVarsF-def*
unfolding *InvariantVarsM-def*
by (*auto simp add*: *Let-def*)
qed

lemma *applyBackjumpEffect*:
assumes
InvariantConsistent (*getM state*)
InvariantUniq (*getM state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**

getConflictFlag state
InvariantCFalse (*getConflictFlag state*) (*getM state*) (*getC state*) **and**
InvariantCEntailed (*getConflictFlag state*) *F0* (*getC state*) **and**
InvariantClCharacterization (*getCl state*) (*getC state*) (*getM state*) **and**
InvariantCllCharacterization (*getCl state*) (*getCll state*) (*getC state*) (*getM state*) **and**
InvariantClCurrentLevel (*getCl state*) (*getM state*)
InvariantUniqC (*getC state*)

isUIP (*opposite* (*getCl state*)) (*getC state*) (*getM state*)
currentLevel (*getM state*) > *0*

shows
let l = (getCl state) in
let bClause = (getC state) in
let bLiteral = opposite l in
let level = getBackjumpLevel state in
let prefix = prefixToLevel level (getM state) in
let state'' = applyBackjump state in
(formulaEntailsClause F0 bClause ∧
isUnitClause bClause bLiteral (elements prefix) ∧
(getM state'') = prefix @ [(bLiteral, False)]) ∧
getF state'' = getF state
proof−
let *?l = getCl state*
let *?level = getBackjumpLevel state*
let *?prefix = prefixToLevel ?level (getM state)*
let *?state' = state(| getConflictFlag := False, getQ := [], getM := ?prefix |)*
let *?state'' = applyBackjump state*

have *clauseFalse (getC state) (elements (getM state))*
using ‹*getConflictFlag state*›
using ‹*InvariantCFalse (getConflictFlag state) (getM state) (getC state)*›
unfolding *InvariantCFalse-def*
by *simp*

have *formulaEntailsClause F0 (getC state)*
using ‹*getConflictFlag state*›
using ‹*InvariantCEntailed (getConflictFlag state) F0 (getC state)*›
unfolding *InvariantCEntailed-def*
by *simp*

have *isBackjumpLevel ?level (opposite ?l) (getC state) (getM state)*
using *assms*
using *isMinimalBackjumpLevelGetBackjumpLevel[of state]*
unfolding *isMinimalBackjumpLevel-def*
by *(simp add: Let-def)*
then have *isUnitClause (getC state) (opposite ?l) (elements ?prefix)*
using *assms*
using ‹*clauseFalse (getC state) (elements (getM state))*›
using *isBackjumpLevelEnsuresIsUnitInPrefix[of getM state getC state ?level opposite ?l]*
unfolding *InvariantConsistent-def*
unfolding *InvariantUniq-def*
by *simp*
moreover
have *getM ?state'' = ?prefix @ [(opposite ?l, False)] getF ?state'' = getF state*
unfolding *applyBackjump-def*

using *assms*
using *assertLiteralEffect*
unfolding *setReason-def*
by (*auto simp add*: *Let-def*)
ultimately
show *?thesis*
using ‹*formulaEntailsClause F0* (*getC state*)›
by (*simp add*: *Let-def*)
qed

lemma *applyBackjumpPreservedVariables*:
assumes
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
shows
let state′ = applyBackjump state in
getSATFlag state′ = getSATFlag state
using *assms*
unfolding *applyBackjump-def*
unfolding *setReason-def*
by (*auto simp add*: *Let-def assertLiteralEffect*)

lemma *InvariantWatchCharacterizationInBackjumpPrefix*:
assumes
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*)

shows
let l = getCl state in
let level = getBackjumpLevel state in
let prefix = prefixToLevel level (*getM state*) *in*
let state′ = state⦇ *getConflictFlag := False*, *getQ :=* [], *getM := prefix* ⦈ *in*
InvariantWatchCharacterization (*getF state′*) (*getWatch1 state′*) (*getWatch2 state′*) (*getM state′*)
proof−
let *?l = getCl state*
let *?level = getBackjumpLevel state*
let *?prefix = prefixToLevel ?level* (*getM state*)
let *?state′ = state*⦇ *getConflictFlag := False*, *getQ :=* [], *getM := ?prefix* ⦈

{
fix *c w1 w2*
assume *c < length* (*getF state*) *Some w1 = getWatch1 state c Some w2 = getWatch2 state c*
with ‹*InvariantWatchCharacterization* (*getF state*) (*getWatch1*

state) (*getWatch2 state*) (*getM state*)›
have *watchCharacterizationCondition w1 w2* (*getM state*) (*nth* (*getF state*) *c*)
watchCharacterizationCondition w2 w1 (*getM state*) (*nth* (*getF state*) *c*)
unfolding *InvariantWatchCharacterization-def*
by *auto*

let *?clause* = *nth* (*getF state*) *c*
let *?a state w1 w2* = ∃ *l*. *l el ?clause* ∧ *literalTrue l* (*elements* (*getM state*)) ∧
elementLevel l (*getM state*) ≤ *elementLevel* (*opposite w1*) (*getM state*)
let *?b state w1 w2* = ∀ *l*. *l el ?clause* ∧ *l* ≠ *w1* ∧ *l* ≠ *w2* ⟶
literalFalse l (*elements* (*getM state*)) ∧
elementLevel (*opposite l*) (*getM state*) ≤ *elementLevel* (*opposite w1*) (*getM state*)

have *watchCharacterizationCondition w1 w2* (*getM ?state′*) *?clause* ∧
watchCharacterizationCondition w2 w1 (*getM ?state′*) *?clause*
proof−
{
assume *literalFalse w1* (*elements* (*getM ?state′*))
hence *literalFalse w1* (*elements* (*getM state*))
using *isPrefixPrefixToLevel*[*of ?level getM state*]
using *isPrefixElements*[*of prefixToLevel ?level* (*getM state*) *getM state*]
using *prefixIsSubset*[*of elements* (*prefixToLevel ?level* (*getM state*)) *elements* (*getM state*)]
by *auto*

from ‹*literalFalse w1* (*elements* (*getM ?state′*))›
have *elementLevel* (*opposite w1*) (*getM state*) ≤ *?level*
using *prefixToLevelElementsElementLevel*[*of opposite w1 ?level getM state*]
by *simp*

from ‹*literalFalse w1* (*elements* (*getM ?state′*))›
have *elementLevel* (*opposite w1*) (*getM ?state′*) = *elementLevel* (*opposite w1*) (*getM state*)
using *elementLevelPrefixElement*
by *simp*

have *?a ?state′ w1 w2* ∨ *?b ?state′ w1 w2*
proof (*cases ?a state w1 w2*)
case *True*
then obtain *l*

where *l el ?clause literalTrue l* (*elements* (*getM state*))
elementLevel l (*getM state*) ≤ *elementLevel* (*opposite w1*) (*getM state*)
by *auto*

have *literalTrue l* (*elements* (*getM ?state'*))
using ‹*elementLevel* (*opposite w1*) (*getM state*) ≤ *?level*›
using *elementLevelLtLevelImpliesMemberPrefixToLevel*[*of l getM state ?level*]
using ‹*elementLevel l* (*getM state*) ≤ *elementLevel* (*opposite w1*) (*getM state*)›
using ‹*literalTrue l* (*elements* (*getM state*))›
by *simp*
moreover
from ‹*literalTrue l* (*elements* (*getM ?state'*))›
have *elementLevel l* (*getM ?state'*) = *elementLevel l* (*getM state*)
using *elementLevelPrefixElement*
by *simp*
ultimately
show *?thesis*
using ‹*elementLevel* (*opposite w1*) (*getM ?state'*) = *elementLevel* (*opposite w1*) (*getM state*)›
using ‹*elementLevel l* (*getM state*) ≤ *elementLevel* (*opposite w1*) (*getM state*)›
using ‹*l el ?clause*›
by *auto*
next
case *False*
{
fix *l*
assume *l el ?clause l* ≠ *w1 l* ≠ *w2*
hence *literalFalse l* (*elements* (*getM state*))
elementLevel (*opposite l*) (*getM state*) ≤ *elementLevel* (*opposite w1*) (*getM state*)
using ‹*literalFalse w1* (*elements* (*getM state*))›
using *False*
using ‹*watchCharacterizationCondition w1 w2* (*getM state*) *?clause*›
unfolding *watchCharacterizationCondition-def*
by *auto*

have *literalFalse l* (*elements* (*getM ?state'*)) ∧
elementLevel (*opposite l*) (*getM ?state'*) ≤ *elementLevel* (*opposite w1*) (*getM ?state'*)
proof−
have *literalFalse l* (*elements* (*getM ?state'*))
using ‹*elementLevel* (*opposite w1*) (*getM state*) ≤ *?level*›
using *elementLevelLtLevelImpliesMemberPrefixToLevel*[*of*

opposite l getM state ?level]
using ‹*elementLevel* (*opposite l*) (*getM state*) ≤ *elementLevel* (*opposite w1*) (*getM state*)›
using ‹*literalFalse l* (*elements* (*getM state*))›
by *simp*
moreover
from ‹*literalFalse l* (*elements* (*getM ?state'*))›
have *elementLevel* (*opposite l*) (*getM ?state'*) = *elementLevel* (*opposite l*) (*getM state*)
using *elementLevelPrefixElement*
by *simp*
ultimately
show *?thesis*
using ‹*elementLevel* (*opposite w1*) (*getM ?state'*) = *elementLevel* (*opposite w1*) (*getM state*)›
using ‹*elementLevel* (*opposite l*) (*getM state*) ≤ *elementLevel* (*opposite w1*) (*getM state*)›
using ‹*l el ?clause*›
by *auto*
qed
}
thus *?thesis*
by *auto*
qed
}
moreover
{
assume *literalFalse w2* (*elements* (*getM ?state'*))
hence *literalFalse w2* (*elements* (*getM state*))
using *isPrefixPrefixToLevel*[*of ?level getM state*]
using *isPrefixElements*[*of prefixToLevel ?level* (*getM state*) *getM state*]
using *prefixIsSubset*[*of elements* (*prefixToLevel ?level* (*getM state*)) *elements* (*getM state*)]
by *auto*

from ‹*literalFalse w2* (*elements* (*getM ?state'*))›
have *elementLevel* (*opposite w2*) (*getM state*) ≤ *?level*
using *prefixToLevelElementsElementLevel*[*of opposite w2 ?level getM state*]
by *simp*

from ‹*literalFalse w2* (*elements* (*getM ?state'*))›
have *elementLevel* (*opposite w2*) (*getM ?state'*) = *elementLevel* (*opposite w2*) (*getM state*)
using *elementLevelPrefixElement*
by *simp*

have *?a ?state' w2 w1* ∨ *?b ?state' w2 w1*

proof (*cases ?a state w2 w1*)
case *True*
then obtain *l*
where *l el ?clause literalTrue l* (*elements* (*getM state*))
elementLevel l (*getM state*) ≤ *elementLevel* (*opposite w2*) (*getM state*)
by *auto*

have *literalTrue l* (*elements* (*getM ?state′*))
using ‹*elementLevel* (*opposite w2*) (*getM state*) ≤ *?level*›
using *elementLevelLtLevelImpliesMemberPrefixToLevel*[*of l getM state ?level*]
using ‹*elementLevel l* (*getM state*) ≤ *elementLevel* (*opposite w2*) (*getM state*)›
using ‹*literalTrue l* (*elements* (*getM state*))›
by *simp*
moreover
from ‹*literalTrue l* (*elements* (*getM ?state′*))›
have *elementLevel l* (*getM ?state′*) = *elementLevel l* (*getM state*)
using *elementLevelPrefixElement*
by *simp*
ultimately
show *?thesis*
using ‹*elementLevel* (*opposite w2*) (*getM ?state′*) = *elementLevel* (*opposite w2*) (*getM state*)›
using ‹*elementLevel l* (*getM state*) ≤ *elementLevel* (*opposite w2*) (*getM state*)›
using ‹*l el ?clause*›
by *auto*
next
case *False*
{
fix *l*
assume *l el ?clause l* ≠ *w1 l* ≠ *w2*
hence *literalFalse l* (*elements* (*getM state*))
elementLevel (*opposite l*) (*getM state*) ≤ *elementLevel* (*opposite w2*) (*getM state*)
using ‹*literalFalse w2* (*elements* (*getM state*))›
using *False*
using ‹*watchCharacterizationCondition w2 w1* (*getM state*) *?clause*›
unfolding *watchCharacterizationCondition-def*
by *auto*

have *literalFalse l* (*elements* (*getM ?state′*)) ∧
elementLevel (*opposite l*) (*getM ?state′*) ≤ *elementLevel* (*opposite w2*) (*getM ?state′*)
proof−

have *literalFalse l* (*elements* (*getM ?state'*))
using ‹*elementLevel* (*opposite w2*) (*getM state*) ≤ *?level*›
using *elementLevelLtLevelImpliesMemberPrefixToLevel*[*of opposite l getM state ?level*]
using ‹*elementLevel* (*opposite l*) (*getM state*) ≤ *elementLevel* (*opposite w2*) (*getM state*)›
using ‹*literalFalse l* (*elements* (*getM state*))›
by *simp*
moreover
from ‹*literalFalse l* (*elements* (*getM ?state'*))›
have *elementLevel* (*opposite l*) (*getM ?state'*) = *elementLevel* (*opposite l*) (*getM state*)
using *elementLevelPrefixElement*
by *simp*
ultimately
show *?thesis*
using ‹*elementLevel* (*opposite w2*) (*getM ?state'*) = *elementLevel* (*opposite w2*) (*getM state*)›
using ‹*elementLevel* (*opposite l*) (*getM state*) ≤ *elementLevel* (*opposite w2*) (*getM state*)›
using ‹*l el ?clause*›
by *auto*
qed
}
thus *?thesis*
by *auto*
qed
}
ultimately
show *?thesis*
unfolding *watchCharacterizationCondition-def*
by *auto*
qed
}
thus *?thesis*
unfolding *InvariantWatchCharacterization-def*
by *auto*
qed

lemma *InvariantConsistentAfterApplyBackjump*:
assumes
InvariantConsistent (*getM state*)
InvariantUniq (*getM state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
and
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**

getConflictFlag state

InvariantCFalse (*getConflictFlag state*) (*getM state*) (*getC state*) **and**
InvariantUniqC (*getC state*)
InvariantCEntailed (*getConflictFlag state*) *F0* (*getC state*) **and**
InvariantClCharacterization (*getCl state*) (*getC state*) (*getM state*)
and
InvariantCllCharacterization (*getCl state*) (*getCll state*) (*getC state*)
(*getM state*) **and**
InvariantClCurrentLevel (*getCl state*) (*getM state*)

currentLevel (*getM state*) > *0*
isUIP (*opposite* (*getCl state*)) (*getC state*) (*getM state*)
shows
let state′ = applyBackjump state in
InvariantConsistent (*getM state′*)
proof−
let *?l = getCl state*
let *?bClause = getC state*
let *?bLiteral = opposite ?l*
let *?level = getBackjumpLevel state*
let *?prefix = prefixToLevel ?level* (*getM state*)
let *?state′′ = applyBackjump state*

have *formulaEntailsClause F0 ?bClause* **and**
isUnitClause ?bClause ?bLiteral (*elements ?prefix*) **and**
getM ?state′′ = ?prefix @ [(*?bLiteral*, *False*)]
using *assms*
using *applyBackjumpEffect*[*of state*]
by (*auto simp add*: *Let-def*)
thus *?thesis*
using ‹*InvariantConsistent* (*getM state*)›
using *InvariantConsistentAfterBackjump*[*of getM state ?prefix ?bClause*
?bLiteral getM ?state′′]
using *isPrefixPrefixToLevel*
by (*auto simp add*: *Let-def*)
qed

lemma *InvariantUniqAfterApplyBackjump*:
assumes
InvariantConsistent (*getM state*)
InvariantUniq (*getM state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
and
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*)
(*getF state*) **and**

getConflictFlag state
InvariantCFalse (*getConflictFlag state*) (*getM state*) (*getC state*) **and**
InvariantUniqC (*getC state*)

InvariantCEntailed (*getConflictFlag state*) *F0* (*getC state*) **and**
InvariantClCharacterization (*getCl state*) (*getC state*) (*getM state*)
and
InvariantCllCharacterization (*getCl state*) (*getCll state*) (*getC state*)
(*getM state*) **and**
InvariantClCurrentLevel (*getCl state*) (*getM state*)

currentLevel (*getM state*) > *0*
isUIP (*opposite* (*getCl state*)) (*getC state*) (*getM state*)
shows
let state′ = applyBackjump state in
InvariantUniq (*getM state′*)
proof−
let *?l = getCl state*
let *?bClause = getC state*
let *?bLiteral = opposite ?l*
let *?level = getBackjumpLevel state*
let *?prefix = prefixToLevel ?level* (*getM state*)
let *?state′′ = applyBackjump state*

have *clauseFalse* (*getC state*) (*elements* (*getM state*))
using ‹*getConflictFlag state*›
using ‹*InvariantCFalse* (*getConflictFlag state*) (*getM state*) (*getC state*)›
unfolding *InvariantCFalse-def*
by *simp*

have *isUnitClause ?bClause ?bLiteral* (*elements ?prefix*) **and**
getM ?state′′ = ?prefix @ [(*?bLiteral*, *False*)]
using *assms*
using *applyBackjumpEffect*[*of state*]
by (*auto simp add*: *Let-def*)
thus *?thesis*
using ‹*InvariantUniq* (*getM state*)›
using *InvariantUniqAfterBackjump*[*of getM state ?prefix ?bClause ?bLiteral getM ?state′′*]
using *isPrefixPrefixToLevel*
by (*auto simp add*: *Let-def*)
qed

lemma *WatchInvariantsAfterApplyBackjump*:
assumes
InvariantConsistent (*getM state*)
InvariantUniq (*getM state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
and
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2

state) (*getM state*) **and**
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)

getConflictFlag state
InvariantUniqC (*getC state*)
InvariantCFalse (*getConflictFlag state*) (*getM state*) (*getC state*) **and**
InvariantCEntailed (*getConflictFlag state*) *F0* (*getC state*) **and**
InvariantClCharacterization (*getCl state*) (*getC state*) (*getM state*) **and**
InvariantCllCharacterization (*getCl state*) (*getCll state*) (*getC state*) (*getM state*) **and**
InvariantClCurrentLevel (*getCl state*) (*getM state*)

isUIP (*opposite* (*getCl state*)) (*getC state*) (*getM state*)
currentLevel (*getM state*) $>$ *0*
shows
let state' = (*applyBackjump state*) *in*
InvariantWatchesEl (*getF state'*) (*getWatch1 state'*) (*getWatch2 state'*) $\wedge$
InvariantWatchesDiffer (*getF state'*) (*getWatch1 state'*) (*getWatch2 state'*) $\wedge$
InvariantWatchCharacterization (*getF state'*) (*getWatch1 state'*) (*getWatch2 state'*) (*getM state'*) $\wedge$
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state'*) (*getF state'*) $\wedge$
InvariantWatchListsUniq (*getWatchList state'*) $\wedge$
InvariantWatchListsCharacterization (*getWatchList state'*) (*getWatch1 state'*) (*getWatch2 state'*)
(**is** *let state'* = (*applyBackjump state*) *in ?inv state'*)
proof−
let *?l* = *getCl state*
let *?level* = *getBackjumpLevel state*
let *?prefix* = *prefixToLevel ?level* (*getM state*)
let *?state'* = *state*(| *getConflictFlag* := *False*, *getQ* := [], *getM* := *?prefix* |)
let *?state''* = *setReason* (*opposite* (*getCl state*)) (*length* (*getF state*) − *1*) *?state'*
let *?state0* = *assertLiteral* (*opposite* (*getCl state*)) *False ?state''*

have *getF ?state'* = *getF state getWatchList ?state'* = *getWatchList state*
getWatch1 ?state' = *getWatch1 state getWatch2 ?state'* = *getWatch2 state*
unfolding *setReason-def*
by (*auto simp add*: *Let-def*)

moreover
have *InvariantWatchCharacterization* (*getF ?state′*) (*getWatch1 ?state′*) (*getWatch2 ?state′*) (*getM ?state′*)
using *assms*
using *InvariantWatchCharacterizationInBackjumpPrefix*[*of state*]
unfolding *setReason-def*
by (*simp add*: *Let-def*)
moreover
have *InvariantConsistent* (*?prefix* @ [(*opposite ?l*, *False*)])
using *assms*
using *InvariantConsistentAfterApplyBackjump*[*of state F0*]
using *assertLiteralEffect*
unfolding *applyBackjump-def*
unfolding *setReason-def*
by (*auto simp add*: *Let-def split*: *if-split-asm*)
moreover
have *InvariantUniq* (*?prefix* @ [(*opposite ?l*, *False*)])
using *assms*
using *InvariantUniqAfterApplyBackjump*[*of state F0*]
using *assertLiteralEffect*
unfolding *applyBackjump-def*
unfolding *setReason-def*
by (*auto simp add*: *Let-def split*: *if-split-asm*)
ultimately
show *?thesis*
using *assms*
using *WatchInvariantsAfterAssertLiteral*[*of ?state′′ opposite ?l False*]
using *WatchInvariantsAfterAssertLiteral*[*of ?state′ opposite ?l False*]
using *InvariantWatchCharacterizationAfterAssertLiteral*[*of ?state′′ opposite ?l False*]
using *InvariantWatchCharacterizationAfterAssertLiteral*[*of ?state′ opposite ?l False*]
unfolding *applyBackjump-def*
unfolding *setReason-def*
by (*auto simp add*: *Let-def*)
qed

lemma *InvariantUniqQAfterApplyBackjump*:
assumes
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
shows
let state′ = applyBackjump state in
InvariantUniqQ (*getQ state′*)
proof−
let *?l = getCl state*
let *?level = getBackjumpLevel state*

let *?prefix* = *prefixToLevel ?level* (*getM state*)
let *?state'* = *state*⦇ *getConflictFlag* := *False*, *getQ* := [], *getM* := *?prefix* ⦈
let *?state''* = *setReason* (*opposite* (*getCl state*)) (*length* (*getF state*) − *1*) *?state'*

show *?thesis*
using *assms*
unfolding *applyBackjump-def*
using *InvariantUniqQAfterAssertLiteral*[*of ?state' opposite ?l False*]
using *InvariantUniqQAfterAssertLiteral*[*of ?state'' opposite ?l False*]
unfolding *InvariantUniqQ-def*
unfolding *setReason-def*
by (*auto simp add*: *Let-def*)
qed

lemma *invariantQCharacterizationAfterApplyBackjump-1*:
assumes
InvariantConsistent (*getM state*)
InvariantUniq (*getM state*)
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*) **and**
InvariantConflictFlagCharacterization (*getConflictFlag state*) (*getF state*) (*getM state*) **and**
InvariantQCharacterization (*getConflictFlag state*) (*getQ state*) (*getF state*) (*getM state*) **and**

InvariantUniqC (*getC state*)
getC state = [*opposite* (*getCl state*)]
InvariantNoDecisionsWhenUnit (*getF state*) (*getM state*) (*currentLevel* (*getM state*))
InvariantNoDecisionsWhenConflict (*getF state*) (*getM state*) (*currentLevel* (*getM state*))

getConflictFlag state
InvariantCFalse (*getConflictFlag state*) (*getM state*) (*getC state*)
InvariantCEntailed (*getConflictFlag state*) *F0* (*getC state*) **and**
InvariantClCharacterization (*getCl state*) (*getC state*) (*getM state*) **and**

InvariantCllCharacterization (*getCl state*) (*getCll state*) (*getC state*) (*getM state*) **and**
InvariantClCurrentLevel (*getCl state*) (*getM state*)

currentLevel (*getM state*) > *0*
isUIP (*opposite* (*getCl state*)) (*getC state*) (*getM state*)
shows
let state'' = (*applyBackjump state*) *in*
InvariantQCharacterization (*getConflictFlag state''*) (*getQ state''*) (*getF state''*) (*getM state''*)
proof−
let *?l* = *getCl state*
let *?level* = *getBackjumpLevel state*
let *?prefix* = *prefixToLevel ?level* (*getM state*)
let *?state'* = *state*(| *getConflictFlag* := *False*, *getQ* := [], *getM* := *?prefix* |)
let *?state''* = *setReason* (*opposite* (*getCl state*)) (*length* (*getF state*) − *1*) *?state'*

let *?state'1* = *assertLiteral* (*opposite ?l*) *False ?state'*
let *?state''1* = *assertLiteral* (*opposite ?l*) *False ?state''*

have *?level* < *elementLevel ?l* (*getM state*)
using *assms*
using *isMinimalBackjumpLevelGetBackjumpLevel*[*of state*]
unfolding *isMinimalBackjumpLevel-def*
unfolding *isBackjumpLevel-def*
by (*simp add*: *Let-def*)
hence *?level* < *currentLevel* (*getM state*)
using *elementLevelLeqCurrentLevel*[*of ?l getM state*]
by *simp*
hence *InvariantQCharacterization* (*getConflictFlag ?state'*) (*getQ ?state'*) (*getF ?state'*) (*getM ?state'*)
InvariantConflictFlagCharacterization (*getConflictFlag ?state'*) (*getF ?state'*) (*getM ?state'*)
unfolding *InvariantQCharacterization-def*
unfolding *InvariantConflictFlagCharacterization-def*
using ‹*InvariantNoDecisionsWhenConflict* (*getF state*) (*getM state*) (*currentLevel* (*getM state*))›
using ‹*InvariantNoDecisionsWhenUnit* (*getF state*) (*getM state*) (*currentLevel* (*getM state*))›
unfolding *InvariantNoDecisionsWhenConflict-def*
unfolding *InvariantNoDecisionsWhenUnit-def*
unfolding *applyBackjump-def*
by (*auto simp add*: *Let-def set-conv-nth*)
moreover
have *InvariantConsistent* (*?prefix* @ [(*opposite ?l*, *False*)])
using *assms*
using *InvariantConsistentAfterApplyBackjump*[*of state F0*]

using *assertLiteralEffect*
unfolding *applyBackjump-def*
unfolding *setReason-def*
by (*auto simp add*: *Let-def split*: *if-split-asm*)
moreover
have *InvariantWatchCharacterization* (*getF ?state'*) (*getWatch1 ?state'*) (*getWatch2 ?state'*) (*getM ?state'*)
using *InvariantWatchCharacterizationInBackjumpPrefix*[*of state*]
using *assms*
by (*simp add*: *Let-def*)
moreover
have ¬ *opposite ?l el* (*getQ ?state'1*) ¬ *opposite ?l el* (*getQ ?state''1*)
using *assertedLiteralIsNotUnit*[*of ?state' opposite ?l False*]
using *assertedLiteralIsNotUnit*[*of ?state'' opposite ?l False*]
using ‹*InvariantQCharacterization* (*getConflictFlag ?state'*) (*getQ ?state'*) (*getF ?state'*) (*getM ?state'*)›
using ‹*InvariantConsistent* (*?prefix* @ [(*opposite ?l*, *False*)])›
using ‹*InvariantWatchCharacterization* (*getF ?state'*) (*getWatch1 ?state'*) (*getWatch2 ?state'*) (*getM ?state'*)›
unfolding *applyBackjump-def*
unfolding *setReason-def*
using *assms*
by (*auto simp add*: *Let-def split*: *if-split-asm*)
hence *removeAll* (*opposite ?l*) (*getQ ?state'1*) = *getQ ?state'1*
removeAll (*opposite ?l*) (*getQ ?state''1*) = *getQ ?state''1*
using *removeAll-id*[*of opposite ?l getQ ?state'1*]
using *removeAll-id*[*of opposite ?l getQ ?state''1*]
unfolding *setReason-def*
by *auto*
ultimately
show *?thesis*
using *assms*
using *InvariantWatchCharacterizationInBackjumpPrefix*[*of state*]
using *InvariantQCharacterizationAfterAssertLiteral*[*of ?state' opposite ?l False*]
using *InvariantQCharacterizationAfterAssertLiteral*[*of ?state'' opposite ?l False*]
unfolding *applyBackjump-def*
unfolding *setReason-def*
by (*auto simp add*: *Let-def*)
qed

lemma *invariantQCharacterizationAfterApplyBackjump-2*:
fixes *state*::*State*
assumes
InvariantConsistent (*getM state*)
InvariantUniq (*getM state*)
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*)

(*getF state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*) **and**
InvariantConflictFlagCharacterization (*getConflictFlag state*) (*getF state*) (*getM state*) **and**
InvariantQCharacterization (*getConflictFlag state*) (*getQ state*) (*getF state*) (*getM state*) **and**

InvariantUniqC (*getC state*)
getC state ≠ [*opposite* (*getCl state*)]
InvariantNoDecisionsWhenUnit (*butlast* (*getF state*)) (*getM state*) (*currentLevel* (*getM state*))
InvariantNoDecisionsWhenConflict (*butlast* (*getF state*)) (*getM state*) (*currentLevel* (*getM state*))
getF state ≠ []
last (*getF state*) = *getC state*

getConflictFlag state
InvariantCFalse (*getConflictFlag state*) (*getM state*) (*getC state*) **and**
InvariantCEntailed (*getConflictFlag state*) *F0* (*getC state*) **and**
InvariantClCharacterization (*getCl state*) (*getC state*) (*getM state*) **and**
InvariantCllCharacterization (*getCl state*) (*getCll state*) (*getC state*) (*getM state*) **and**
InvariantClCurrentLevel (*getCl state*) (*getM state*)

currentLevel (*getM state*) > *0*
isUIP (*opposite* (*getCl state*)) (*getC state*) (*getM state*)
shows
let state'' = (*applyBackjump state*) *in*
InvariantQCharacterization (*getConflictFlag state''*) (*getQ state''*) (*getF state''*) (*getM state''*)
proof−
let *?l* = *getCl state*
let *?level* = *getBackjumpLevel state*
let *?prefix* = *prefixToLevel ?level* (*getM state*)

let *?state'* = *state*⦇ *getConflictFlag* := *False*, *getQ* := [], *getM* := *?prefix* ⦈
let *?state''* = *setReason* (*opposite* (*getCl state*)) (*length* (*getF state*) − *1*) *?state'*

have *?level* < *elementLevel ?l* (*getM state*)
using *assms*
using *isMinimalBackjumpLevelGetBackjumpLevel*[*of state*]
unfolding *isMinimalBackjumpLevel-def*
unfolding *isBackjumpLevel-def*
by (*simp add*: *Let-def*)
hence *?level* < *currentLevel* (*getM state*)
using *elementLevelLeqCurrentLevel*[*of ?l getM state*]
by *simp*

have *isUnitClause* (*last* (*getF state*)) (*opposite ?l*) (*elements ?prefix*)
using ‹*last* (*getF state*) = *getC state*›
using *isMinimalBackjumpLevelGetBackjumpLevel*[*of state*]
using ‹*InvariantUniq* (*getM state*)›
using ‹*InvariantConsistent* (*getM state*)›
using ‹*getConflictFlag state*›
using ‹*InvariantUniqC* (*getC state*)›
using ‹*InvariantCFalse* (*getConflictFlag state*) (*getM state*) (*getC state*)›
using *isBackjumpLevelEnsuresIsUnitInPrefix*[*of getM state getC state getBackjumpLevel state opposite ?l*]
using ‹*InvariantClCharacterization* (*getCl state*) (*getC state*) (*getM state*)›
using ‹*InvariantCllCharacterization* (*getCl state*) (*getCll state*) (*getC state*) (*getM state*)›
using ‹*InvariantClCurrentLevel* (*getCl state*) (*getM state*)›
using ‹*currentLevel* (*getM state*) > *0*›
using ‹*isUIP* (*opposite* (*getCl state*)) (*getC state*) (*getM state*)›
unfolding *isMinimalBackjumpLevel-def*
unfolding *InvariantUniq-def*
unfolding *InvariantConsistent-def*
unfolding *InvariantCFalse-def*
by (*simp add*: *Let-def*)
hence ¬ *clauseFalse* (*last* (*getF state*)) (*elements ?prefix*)
unfolding *isUnitClause-def*
by (*auto simp add*: *clauseFalseIffAllLiteralsAreFalse*)

have *InvariantConsistent* (*?prefix* @ [(*opposite ?l*, *False*)])
using *assms*
using *InvariantConsistentAfterApplyBackjump*[*of state F0*]
using *assertLiteralEffect*
unfolding *applyBackjump-def*
unfolding *setReason-def*
by (*auto simp add*: *Let-def split*: *if-split-asm*)

have *InvariantUniq* (*?prefix* @ [(*opposite ?l*, *False*)])
using *assms*
using *InvariantUniqAfterApplyBackjump*[*of state F0*]
using *assertLiteralEffect*

unfolding *applyBackjump-def*
unfolding *setReason-def*
by (*auto simp add*: *Let-def split*: *if-split-asm*)

let *?state'1* = *?state'* (| *getQ* := *getQ ?state'* @ [*opposite ?l*]|)
let *?state'2* = *assertLiteral* (*opposite ?l*) *False ?state'1*

let *?state''1* = *?state''* (| *getQ* := *getQ ?state''* @ [*opposite ?l*]|)
let *?state''2* = *assertLiteral* (*opposite ?l*) *False ?state''1*

have *InvariantQCharacterization* (*getConflictFlag ?state'*) ((*getQ ?state'*) @ [*opposite ?l*]) (*getF ?state'*) (*getM ?state'*)
proof−
have ∀ *l c*. *c el* (*butlast* (*getF state*)) ⟶ ¬ *isUnitClause c l* (*elements* (*getM ?state'*))
using ‹*InvariantNoDecisionsWhenUnit* (*butlast* (*getF state*)) (*getM state*) (*currentLevel* (*getM state*))›
using ‹*?level* < *currentLevel* (*getM state*)›
unfolding *InvariantNoDecisionsWhenUnit-def*
by *simp*

have ∀ *l*. ((∃ *c*. *c el* (*getF state*) ∧ *isUnitClause c l* (*elements* (*getM ?state'*))) = (*l* = *opposite ?l*))
proof
fix *l*
show (∃ *c*. *c el* (*getF state*) ∧ *isUnitClause c l* (*elements* (*getM ?state'*))) = (*l* = *opposite ?l*) (is *?lhs* = *?rhs*)
proof
assume *?lhs*
then obtain *c*::*Clause*
where *c el* (*getF state*) and *isUnitClause c l* (*elements ?prefix*)
by *auto*
show *?rhs*
proof (*cases c el* (*butlast* (*getF state*)))
case *True*
thus *?thesis*
using ‹∀ *l c*. *c el* (*butlast* (*getF state*)) ⟶ ¬ *isUnitClause c l* (*elements* (*getM ?state'*))›
using ‹*isUnitClause c l* (*elements ?prefix*)›
by *auto*
next
case *False*

from ‹*getF state* ≠ []›
have *butlast* (*getF state*) @ [*last* (*getF state*)] = *getF state*
using *append-butlast-last-id*[*of getF state*]
by *simp*
hence *getF state* = *butlast* (*getF state*) @ [*last* (*getF state*)]
by (*rule sym*)

```
          with ‹c el getF state›
          have c el butlast (getF state) ∨ c el [last (getF state)]
            using set-append[of butlast (getF state) [last (getF state)]]
            by auto
          hence c = last (getF state)
            using ‹¬ c el (butlast (getF state))›
            by simp
          thus ?thesis
          using ‹isUnitClause (last (getF state)) (opposite ?l) (elements
?prefix)›
            using ‹isUnitClause c l (elements ?prefix)›
            unfolding isUnitClause-def
            by auto
        qed
        next
          from ‹getF state ≠ []›
          have last (getF state) el (getF state)
            by auto
          assume ?rhs
          thus ?lhs
          using ‹isUnitClause (last (getF state)) (opposite ?l) (elements
?prefix)›
            using ‹last (getF state) el (getF state)›
            by auto
      qed
    qed
    thus ?thesis
      unfolding InvariantQCharacterization-def
      by simp
  qed
  hence InvariantQCharacterization (getConflictFlag ?state'1) (getQ
?state'1) (getF ?state'1) (getM ?state'1)
    by simp
  hence InvariantQCharacterization (getConflictFlag ?state''1) (getQ
?state''1) (getF ?state''1) (getM ?state''1)
    unfolding setReason-def
    by simp

   have InvariantWatchCharacterization (getF ?state'1) (getWatch1
?state'1) (getWatch2 ?state'1) (getM ?state'1)
    using InvariantWatchCharacterizationInBackjumpPrefix[of state]
    using assms
    by (simp add: Let-def)
  hence InvariantWatchCharacterization (getF ?state''1) (getWatch1
?state''1) (getWatch2 ?state''1) (getM ?state''1)
    unfolding setReason-def
    by simp

  have InvariantWatchCharacterization (getF ?state') (getWatch1 ?state')
```

(*getWatch2 ?state'*) (*getM ?state'*)
using *InvariantWatchCharacterizationInBackjumpPrefix*[*of state*]
using *assms*
by (*simp add*: *Let-def*)
hence *InvariantWatchCharacterization* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*) (*getM ?state''*)
unfolding *setReason-def*
by *simp*

have *InvariantConflictFlagCharacterization* (*getConflictFlag ?state'1*) (*getF ?state'1*) (*getM ?state'1*)
proof−
{
fix *c*::*Clause*
assume *c el* (*getF state*)
have ¬ *clauseFalse c* (*elements ?prefix*)
proof (*cases c el* (*butlast* (*getF state*)))
case *True*
thus *?thesis*
using ‹*InvariantNoDecisionsWhenConflict* (*butlast* (*getF state*)) (*getM state*) (*currentLevel* (*getM state*))›
using ‹*?level* < *currentLevel* (*getM state*)›
unfolding *InvariantNoDecisionsWhenConflict-def*
by (*simp add*: *formulaFalseIffContainsFalseClause*)
next
case *False*
from ‹*getF state* ≠ []›
have *butlast* (*getF state*) @ [*last* (*getF state*)] = *getF state*
using *append-butlast-last-id*[*of getF state*]
by *simp*
hence *getF state* = *butlast* (*getF state*) @ [*last* (*getF state*)]
by (*rule sym*)
with ‹*c el getF state*›
have *c el butlast* (*getF state*) ∨ *c el* [*last* (*getF state*)]
using *set-append*[*of butlast* (*getF state*) [*last* (*getF state*)]]
by *auto*
hence *c* = *last* (*getF state*)
using ‹¬ *c el* (*butlast* (*getF state*))›
by *simp*
thus *?thesis*
using ‹¬ *clauseFalse* (*last* (*getF state*)) (*elements ?prefix*)›
by *simp*
qed
} **thus** *?thesis*
unfolding *InvariantConflictFlagCharacterization-def*
by (*simp add*: *formulaFalseIffContainsFalseClause*)
qed
hence *InvariantConflictFlagCharacterization* (*getConflictFlag ?state''1*) (*getF ?state''1*) (*getM ?state''1*)

unfolding *setReason-def*
by *simp*

have *InvariantQCharacterization* (*getConflictFlag ?state'2*) (*removeAll* (*opposite ?l*) (*getQ ?state'2*)) (*getF ?state'2*) (*getM ?state'2*)
using *assms*
using ‹*InvariantConsistent* (*?prefix* @ [(*opposite ?l*, *False*)])›
using ‹*InvariantUniq* (*?prefix* @ [(*opposite ?l*, *False*)])›
using ‹*InvariantConflictFlagCharacterization* (*getConflictFlag ?state'1*) (*getF ?state'1*) (*getM ?state'1*)›
using ‹*InvariantWatchCharacterization* (*getF ?state'1*) (*getWatch1 ?state'1*) (*getWatch2 ?state'1*) (*getM ?state'1*)›
using ‹*InvariantQCharacterization* (*getConflictFlag ?state'1*) (*getQ ?state'1*) (*getF ?state'1*) (*getM ?state'1*)›
using *InvariantQCharacterizationAfterAssertLiteral*[*of ?state'1 opposite ?l False*]
by (*simp add*: *Let-def*)

have *InvariantQCharacterization* (*getConflictFlag ?state''2*) (*removeAll* (*opposite ?l*) (*getQ ?state''2*)) (*getF ?state''2*) (*getM ?state''2*)
using *assms*
using ‹*InvariantConsistent* (*?prefix* @ [(*opposite ?l*, *False*)])›
using ‹*InvariantUniq* (*?prefix* @ [(*opposite ?l*, *False*)])›
using ‹*InvariantConflictFlagCharacterization* (*getConflictFlag ?state''1*) (*getF ?state''1*) (*getM ?state''1*)›
using ‹*InvariantWatchCharacterization* (*getF ?state''1*) (*getWatch1 ?state''1*) (*getWatch2 ?state''1*) (*getM ?state''1*)›
using ‹*InvariantQCharacterization* (*getConflictFlag ?state''1*) (*getQ ?state''1*) (*getF ?state''1*) (*getM ?state''1*)›
using *InvariantQCharacterizationAfterAssertLiteral*[*of ?state''1 opposite ?l False*]
unfolding *setReason-def*
by (*simp add*: *Let-def*)

let *?stateB* = *applyBackjump state*
show *?thesis*
proof (*cases getBackjumpLevel state* > *0*)
case *False*
let *?state01* = *state*⦇*getConflictFlag* := *False*, *getM* := *?prefix*⦈
have *InvariantWatchesEl* (*getF ?state01*) (*getWatch1 ?state01*) (*getWatch2 ?state01*)
using ‹*InvariantWatchesEl* (*getF state*) (*getWatch1 state*) (*getWatch2 state*)›
unfolding *InvariantWatchesEl-def*
by *auto*

have *InvariantWatchListsContainOnlyClausesFromF* (*getWatchList ?state01*) (*getF ?state01*)

using ‹*InvariantWatchListsContainOnlyClausesFromF* (*getWatchList state*) (*getF state*)›
unfolding *InvariantWatchListsContainOnlyClausesFromF-def*
by *auto*

have *assertLiteral* (*opposite ?l*) *False* (*state* (|*getConflictFlag* := *False*, *getQ* := [], *getM* := *?prefix* |)) =
assertLiteral (*opposite ?l*) *False* (*state* (|*getConflictFlag* := *False*, *getM* := *?prefix*, *getQ* := [] |))
using *arg-cong*[*of state* (|*getConflictFlag* := *False*, *getQ* := [], *getM* := *?prefix* |)
state (|*getConflictFlag* := *False*, *getM* := *?prefix*, *getQ* := [] |)
λ *x*. *assertLiteral* (*opposite ?l*) *False x*]
by *simp*
hence *getConflictFlag ?stateB* = *getConflictFlag ?state'2*
getF ?stateB = *getF ?state'2*
getM ?stateB = *getM ?state'2*
unfolding *applyBackjump-def*
using *AssertLiteralStartQIreleveant*[*of ?state01 opposite ?l False* [] [*opposite ?l*]]
using ‹*InvariantWatchesEl* (*getF ?state01*) (*getWatch1 ?state01*) (*getWatch2 ?state01*)›
using ‹*InvariantWatchListsContainOnlyClausesFromF* (*getWatchList ?state01*) (*getF ?state01*)›
using ‹¬ *getBackjumpLevel state* > *0*›
by (*auto simp add*: *Let-def*)

have *set* (*getQ ?stateB*) = *set* (*removeAll* (*opposite ?l*) (*getQ ?state'2*))
proof−
have *set* (*getQ ?stateB*) = *set*(*getQ ?state'2*) − {*opposite ?l*}
proof−
let *?ulSet* = { *ul*. (∃ *uc*. *uc el* (*getF ?state'1*) ∧
?l el uc ∧
isUnitClause uc ul ((*elements* (*getM ?state'1*)) @ [*opposite ?l*])) }
have *set* (*getQ ?state'2*) = {*opposite ?l*} ∪ *?ulSet*
using *assertLiteralQEffect*[*of ?state'1 opposite ?l False*]
using *assms*
using ‹*InvariantConsistent* (*?prefix* @ [(*opposite ?l*, *False*)])›
using ‹*InvariantUniq* (*?prefix* @ [(*opposite ?l*, *False*)])›
using ‹*InvariantWatchCharacterization* (*getF ?state'1*) (*getWatch1 ?state'1*) (*getWatch2 ?state'1*) (*getM ?state'1*)›
by (*simp add*:*Let-def*)
moreover
have *set* (*getQ ?stateB*) = *?ulSet*
using *assertLiteralQEffect*[*of ?state' opposite ?l False*]
using *assms*

using ‹*InvariantConsistent* (*?prefix* @ [(*opposite ?l*, *False*)])›
using ‹*InvariantUniq* (*?prefix* @ [(*opposite ?l*, *False*)])›
using ‹*InvariantWatchCharacterization* (*getF ?state'*) (*getWatch1 ?state'*) (*getWatch2 ?state'*) (*getM ?state'*)›
using ‹¬ *getBackjumpLevel state* > *0*›
unfolding *applyBackjump-def*
by (*simp add*:*Let-def*)
moreover
have ¬ (*opposite ?l*) ∈ *?ulSet*
using *assertedLiteralIsNotUnit*[*of ?state' opposite ?l False*]
using *assms*
using ‹*InvariantConsistent* (*?prefix* @ [(*opposite ?l*, *False*)])›
using ‹*InvariantUniq* (*?prefix* @ [(*opposite ?l*, *False*)])›
using ‹*InvariantWatchCharacterization* (*getF ?state'*) (*getWatch1 ?state'*) (*getWatch2 ?state'*) (*getM ?state'*)›
using ‹*set* (*getQ ?stateB*) = *?ulSet*›
using ‹¬ *getBackjumpLevel state* > *0*›
unfolding *applyBackjump-def*
by (*simp add*: *Let-def*)
ultimately
show *?thesis*
by *simp*
qed
thus *?thesis*
by *simp*
qed

show *?thesis*
using ‹*InvariantQCharacterization* (*getConflictFlag ?state'2*) (*removeAll* (*opposite ?l*) (*getQ ?state'2*)) (*getF ?state'2*) (*getM ?state'2*)›
using ‹*set* (*getQ ?stateB*) = *set* (*removeAll* (*opposite ?l*) (*getQ ?state'2*))›
using ‹*getConflictFlag ?stateB* = *getConflictFlag ?state'2*›
using ‹*getF ?stateB* = *getF ?state'2*›
using ‹*getM ?stateB* = *getM ?state'2*›
unfolding *InvariantQCharacterization-def*
by (*simp add*: *Let-def*)
next
case *True*
let *?state02* = *setReason* (*opposite* (*getCl state*)) (*length* (*getF state*) − *1*)
state(|*getConflictFlag* := *False*, *getM* := *?prefix*|)
have *InvariantWatchesEl* (*getF ?state02*) (*getWatch1 ?state02*) (*getWatch2 ?state02*)
using ‹*InvariantWatchesEl* (*getF state*) (*getWatch1 state*) (*getWatch2 state*)›
unfolding *InvariantWatchesEl-def*
unfolding *setReason-def*
by *auto*

have *InvariantWatchListsContainOnlyClausesFromF* (*getWatchList* *?state02*) (*getF* *?state02*)
using ‹*InvariantWatchListsContainOnlyClausesFromF* (*getWatchList* *state*) (*getF* *state*)›
unfolding *InvariantWatchListsContainOnlyClausesFromF-def*
unfolding *setReason-def*
by *auto*

let *?stateTmp'* = *assertLiteral* (*opposite* (*getCl state*)) *False*
(*setReason* (*opposite* (*getCl state*)) (*length* (*getF state*) − *1*)
state (|*getConflictFlag* := *False*,
getM := *prefixToLevel* (*getBackjumpLevel state*) (*getM* *state*),
getQ := []|)
)
let *?stateTmp''* = *assertLiteral* (*opposite* (*getCl state*)) *False*
(*setReason* (*opposite* (*getCl state*)) (*length* (*getF state*) − *1*)
state (|*getConflictFlag* := *False*,
getM := *prefixToLevel* (*getBackjumpLevel state*) (*getM* *state*),
getQ := [*opposite* (*getCl state*)]|)
)

have *getM ?stateTmp'* = *getM ?stateTmp''*
getF ?stateTmp' = *getF ?stateTmp''*
getSATFlag ?stateTmp' = *getSATFlag ?stateTmp''*
getConflictFlag ?stateTmp' = *getConflictFlag ?stateTmp''*
using *AssertLiteralStartQIreleveant*[*of ?state02 opposite ?l False* [] [*opposite ?l*]]
using ‹*InvariantWatchesEl* (*getF ?state02*) (*getWatch1 ?state02*) (*getWatch2 ?state02*)›
using ‹*InvariantWatchListsContainOnlyClausesFromF* (*getWatchList* *?state02*) (*getF ?state02*)›
by (*auto simp add*: *Let-def*)
moreover
have *?stateB* = *?stateTmp'*
using ‹*getBackjumpLevel state* > *0*›
using *arg-cong*[*of state* (|
getConflictFlag := *False*,
getQ := [],
getM := *?prefix*,
getReason := (*getReason state*)(*opposite ?l* ↦ *length* (*getF state*) − *1*)
|)
state (|
getReason := (*getReason state*)(*opposite ?l* ↦ *length* (*getF state*) − *1*),

getConflictFlag := *False*,
getM := *prefixToLevel* (*getBackjumpLevel state*) (*getM state*),
getQ := []
⦈
λ *x*. *assertLiteral* (*opposite ?l*) *False x*]
unfolding *applyBackjump-def*
unfolding *setReason-def*
by (*auto simp add*: *Let-def*)
moreover
have *?stateTmp″* = *?state″2*
unfolding *setReason-def*
using *arg-cong*[*of state* ⦇*getReason* := (*getReason state*)(*opposite ?l* ↦ *length* (*getF state*) − *1*),
getConflictFlag := *False*,
getM := *?prefix*, *getQ* := [*opposite ?l*]⦈
state ⦇*getConflictFlag* := *False*,
getM := *prefixToLevel* (*getBackjumpLevel state*) (*getM state*),
getReason := (*getReason state*)(*opposite ?l* ↦ *length* (*getF state*) − *1*),
getQ := [*opposite ?l*]⦈
λ *x*. *assertLiteral* (*opposite ?l*) *False x*]
by *simp*
ultimately
have *getConflictFlag ?stateB* = *getConflictFlag ?state″2*
getF ?stateB = *getF ?state″2*
getM ?stateB = *getM ?state″2*
by *auto*

have *set* (*getQ ?stateB*) = *set* (*removeAll* (*opposite ?l*) (*getQ ?state″2*))
proof−
have *set* (*getQ ?stateB*) = *set*(*getQ ?state″2*) − {*opposite ?l*}
proof−
let *?ulSet* = { *ul*. (∃ *uc*. *uc el* (*getF ?state″1*) ∧
?l el uc ∧
isUnitClause uc ul ((*elements* (*getM ?state″1*)) @ [*opposite ?l*])) }
have *set* (*getQ ?state″2*) = {*opposite ?l*} ∪ *?ulSet*
using *assertLiteralQEffect*[*of ?state″1 opposite ?l False*]
using *assms*
using ‹*InvariantConsistent* (*?prefix* @ [(*opposite ?l*, *False*)])›
using ‹*InvariantUniq* (*?prefix* @ [(*opposite ?l*, *False*)])›
using ‹*InvariantWatchCharacterization* (*getF ?state″1*) (*getWatch1 ?state″1*) (*getWatch2 ?state″1*) (*getM ?state″1*)›
unfolding *setReason-def*
by (*simp add*:*Let-def*)
moreover

have *set* (*getQ ?stateB*) = *?ulSet*
using *assertLiteralQEffect*[*of ?state'' opposite ?l False*]
using *assms*
using ‹*InvariantConsistent* (*?prefix* @ [(*opposite ?l*, *False*)])›
using ‹*InvariantUniq* (*?prefix* @ [(*opposite ?l*, *False*)])›
using ‹*InvariantWatchCharacterization* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*) (*getM ?state''*)›
using ‹*getBackjumpLevel state* > *0*›
unfolding *applyBackjump-def*
unfolding *setReason-def*
by (*simp add*:*Let-def*)
moreover
have ¬ (*opposite ?l*) ∈ *?ulSet*
using *assertedLiteralIsNotUnit*[*of ?state'' opposite ?l False*]
using *assms*
using ‹*InvariantConsistent* (*?prefix* @ [(*opposite ?l*, *False*)])›
using ‹*InvariantUniq* (*?prefix* @ [(*opposite ?l*, *False*)])›
using ‹*InvariantWatchCharacterization* (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*) (*getM ?state''*)›
using ‹*set* (*getQ ?stateB*) = *?ulSet*›
using ‹*getBackjumpLevel state* > *0*›
unfolding *applyBackjump-def*
unfolding *setReason-def*
by (*simp add*: *Let-def*)
ultimately
show *?thesis*
by *simp*
qed
thus *?thesis*
by *simp*
qed

show *?thesis*
using ‹*InvariantQCharacterization* (*getConflictFlag ?state''2*) (*removeAll* (*opposite ?l*) (*getQ ?state''2*)) (*getF ?state''2*) (*getM ?state''2*)›
using ‹*set* (*getQ ?stateB*) = *set* (*removeAll* (*opposite ?l*) (*getQ ?state''2*))›
using ‹*getConflictFlag ?stateB* = *getConflictFlag ?state''2*›
using ‹*getF ?stateB* = *getF ?state''2*›
using ‹*getM ?stateB* = *getM ?state''2*›
unfolding *InvariantQCharacterization-def*
by (*simp add*: *Let-def*)
qed
qed

lemma *InvariantConflictFlagCharacterizationAfterApplyBackjump-1*:
assumes
InvariantConsistent (*getM state*)
InvariantUniq (*getM state*)

InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*) **and**

InvariantUniqC (*getC state*)
getC state = [*opposite* (*getCl state*)]
InvariantNoDecisionsWhenConflict (*getF state*) (*getM state*) (*currentLevel* (*getM state*))

getConflictFlag state
InvariantCFalse (*getConflictFlag state*) (*getM state*) (*getC state*) **and**
InvariantCEntailed (*getConflictFlag state*) *F0* (*getC state*) **and**
InvariantClCharacterization (*getCl state*) (*getC state*) (*getM state*) **and**
InvariantCllCharacterization (*getCl state*) (*getCll state*) (*getC state*) (*getM state*) **and**
InvariantClCurrentLevel (*getCl state*) (*getM state*)

currentLevel (*getM state*) > *0*
isUIP (*opposite* (*getCl state*)) (*getC state*) (*getM state*)
shows
let state′ = (*applyBackjump state*) *in*
InvariantConflictFlagCharacterization (*getConflictFlag state′*) (*getF state′*) (*getM state′*)
proof−
let *?l* = *getCl state*
let *?level* = *getBackjumpLevel state*
let *?prefix* = *prefixToLevel ?level* (*getM state*)
let *?state′* = *state*⦇ *getConflictFlag* := *False*, *getQ* := [], *getM* := *?prefix* ⦈
let *?state″* = *setReason* (*opposite ?l*) (*length* (*getF state*) − *1*) *?state′*
let *?stateB* = *applyBackjump state*

have *?level* < *elementLevel ?l* (*getM state*)
using *assms*
using *isMinimalBackjumpLevelGetBackjumpLevel*[*of state*]
unfolding *isMinimalBackjumpLevel-def*
unfolding *isBackjumpLevel-def*
by (*simp add*: *Let-def*)
hence *?level* < *currentLevel* (*getM state*)
using *elementLevelLeqCurrentLevel*[*of ?l getM state*]

by *simp*
hence *InvariantConflictFlagCharacterization* (*getConflictFlag ?state′*) (*getF ?state′*) (*getM ?state′*)
using ‹*InvariantNoDecisionsWhenConflict* (*getF state*) (*getM state*) (*currentLevel* (*getM state*))›
unfolding *InvariantNoDecisionsWhenConflict-def*
unfolding *InvariantConflictFlagCharacterization-def*
by *simp*
moreover
have *InvariantConsistent* (*?prefix* @ [(*opposite ?l*, *False*)])
using *assms*
using *InvariantConsistentAfterApplyBackjump*[*of state F0*]
using *assertLiteralEffect*
unfolding *applyBackjump-def*
unfolding *setReason-def*
by (*auto simp add*: *Let-def split*: *if-split-asm*)
ultimately
show *?thesis*
using *InvariantConflictFlagCharacterizationAfterAssertLiteral*[*of ?state′*]
using *InvariantConflictFlagCharacterizationAfterAssertLiteral*[*of ?state′′*]
using *InvariantWatchCharacterizationInBackjumpPrefix*[*of state*]
using *assms*
unfolding *applyBackjump-def*
unfolding *setReason-def*
using *assertLiteralEffect*
by (*auto simp add*: *Let-def*)
qed

lemma *InvariantConflictFlagCharacterizationAfterApplyBackjump-2*:
assumes
InvariantConsistent (*getM state*)
InvariantUniq (*getM state*)
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*) **and**

InvariantUniqC (*getC state*)
getC state ≠ [*opposite* (*getCl state*)]

InvariantNoDecisionsWhenConflict (*butlast* (*getF state*)) (*getM state*) (*currentLevel* (*getM state*))
getF state ≠ [] *last* (*getF state*) = *getC state*

getConflictFlag state
InvariantCFalse (*getConflictFlag state*) (*getM state*) (*getC state*) **and**
InvariantCEntailed (*getConflictFlag state*) *F0* (*getC state*) **and**
InvariantClCharacterization (*getCl state*) (*getC state*) (*getM state*) **and**
InvariantCllCharacterization (*getCl state*) (*getCll state*) (*getC state*) (*getM state*) **and**
InvariantClCurrentLevel (*getCl state*) (*getM state*)

currentLevel (*getM state*) > *0*
isUIP (*opposite* (*getCl state*)) (*getC state*) (*getM state*)
shows
let state′ = (*applyBackjump state*) *in*
InvariantConflictFlagCharacterization (*getConflictFlag state′*) (*getF state′*) (*getM state′*)
proof−
let *?l* = *getCl state*
let *?level* = *getBackjumpLevel state*
let *?prefix* = *prefixToLevel ?level* (*getM state*)
let *?state′* = *state*⦇ *getConflictFlag* := *False*, *getQ* := [], *getM* := *?prefix* ⦈
let *?state′′* = *setReason* (*opposite ?l*) (*length* (*getF state*) − *1*) *?state′*
let *?stateB* = *applyBackjump state*

have *?level* < *elementLevel ?l* (*getM state*)
using *assms*
using *isMinimalBackjumpLevelGetBackjumpLevel*[*of state*]
unfolding *isMinimalBackjumpLevel-def*
unfolding *isBackjumpLevel-def*
by (*simp add*: *Let-def*)
hence *?level* < *currentLevel* (*getM state*)
using *elementLevelLeqCurrentLevel*[*of ?l getM state*]
by *simp*

hence *InvariantConflictFlagCharacterization* (*getConflictFlag ?state′*) (*butlast* (*getF ?state′*)) (*getM ?state′*)
using ‹*InvariantNoDecisionsWhenConflict* (*butlast* (*getF state*)) (*getM state*) (*currentLevel* (*getM state*))›
unfolding *InvariantNoDecisionsWhenConflict-def*
unfolding *InvariantConflictFlagCharacterization-def*
by *simp*
moreover
have *isBackjumpLevel* (*getBackjumpLevel state*) (*opposite* (*getCl state*)) (*getC state*) (*getM state*)
using *assms*

using *isMinimalBackjumpLevelGetBackjumpLevel*[*of state*]
unfolding *isMinimalBackjumpLevel-def*
by (*simp add*: *Let-def*)
hence *isUnitClause* (*last* (*getF state*)) (*opposite ?l*) (*elements ?prefix*)
using *isBackjumpLevelEnsuresIsUnitInPrefix*[*of getM state getC state getBackjumpLevel state opposite ?l*]
using ‹*InvariantUniq* (*getM state*)›
using ‹*InvariantConsistent* (*getM state*)›
using ‹*getConflictFlag state*›
using ‹*InvariantCFalse* (*getConflictFlag state*) (*getM state*) (*getC state*)›
using ‹*last* (*getF state*) = *getC state*›
unfolding *InvariantUniq-def*
unfolding *InvariantConsistent-def*
unfolding *InvariantCFalse-def*
by (*simp add*: *Let-def*)
hence ¬ *clauseFalse* (*last* (*getF state*)) (*elements ?prefix*)
unfolding *isUnitClause-def*
by (*auto simp add*: *clauseFalseIffAllLiteralsAreFalse*)
moreover
from ‹*getF state* ≠ []›
have *butlast* (*getF state*) @ [*last* (*getF state*)] = *getF state*
using *append-butlast-last-id*[*of getF state*]
by *simp*
hence *getF state* = *butlast* (*getF state*) @ [*last* (*getF state*)]
by (*rule sym*)
ultimately
have *InvariantConflictFlagCharacterization* (*getConflictFlag ?state′*) (*getF ?state′*) (*getM ?state′*)
using *set-append*[*of butlast* (*getF state*) [*last* (*getF state*)]]
unfolding *InvariantConflictFlagCharacterization-def*
by (*auto simp add*: *formulaFalseIffContainsFalseClause*)
moreover
have *InvariantConsistent* (*?prefix* @ [(*opposite ?l*, *False*)])
using *assms*
using *InvariantConsistentAfterApplyBackjump*[*of state F0*]
using *assertLiteralEffect*
unfolding *applyBackjump-def*
unfolding *setReason-def*
by (*auto simp add*: *Let-def split*: *if-split-asm*)
ultimately
show *?thesis*
using *InvariantConflictFlagCharacterizationAfterAssertLiteral*[*of ?state′*]
using *InvariantConflictFlagCharacterizationAfterAssertLiteral*[*of ?state′′*]
using *InvariantWatchCharacterizationInBackjumpPrefix*[*of state*]
using *assms*
using *assertLiteralEffect*

unfolding *applyBackjump-def*
unfolding *setReason-def*
by (*auto simp add*: *Let-def*)
qed

lemma *InvariantConflictClauseCharacterizationAfterApplyBackjump*:
assumes
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
shows
let state′ = applyBackjump state in
InvariantConflictClauseCharacterization (*getConflictFlag state′*) (*getConflictClause state′*) (*getF state′*) (*getM state′*)
proof−
let *?l = getCl state*
let *?level = getBackjumpLevel state*
let *?prefix = prefixToLevel ?level* (*getM state*)
let *?state′ = state*⦇ *getConflictFlag* := *False*, *getQ* := [], *getM* := *?prefix* ⦈
let *?state′′ = if 0 < ?level then setReason* (*opposite ?l*) (*length* (*getF state*) − *1*) *?state′ else ?state′*

have ¬ *getConflictFlag ?state′*
by *simp*
hence *InvariantConflictClauseCharacterization* (*getConflictFlag ?state′′*) (*getConflictClause ?state′′*) (*getF ?state′′*) (*getM ?state′′*)
unfolding *InvariantConflictClauseCharacterization-def*
unfolding *setReason-def*
by *auto*
moreover
have *getF ?state′′ = getF state*
getWatchList ?state′′ = getWatchList state
getWatch1 ?state′′ = getWatch1 state
getWatch2 ?state′′ = getWatch2 state
unfolding *setReason-def*
by *auto*
ultimately
show *?thesis*
using *assms*
using *InvariantConflictClauseCharacterizationAfterAssertLiteral*[*of ?state′′*]
unfolding *applyBackjump-def*
by (*simp only*: *Let-def*)
qed

lemma *InvariantGetReasonIsReasonAfterApplyBackjump*:
assumes
InvariantConsistent (*getM state*)
InvariantUniq (*getM state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*) **and**
getConflictFlag state
InvariantUniqC (*getC state*)
InvariantCFalse (*getConflictFlag state*) (*getM state*) (*getC state*)
InvariantCEntailed (*getConflictFlag state*) *F0* (*getC state*)
InvariantClCharacterization (*getCl state*) (*getC state*) (*getM state*)
InvariantCllCharacterization (*getCl state*) (*getCll state*) (*getC state*) (*getM state*)
InvariantClCurrentLevel (*getCl state*) (*getM state*)
isUIP (*opposite* (*getCl state*)) (*getC state*) (*getM state*)
$0 <$ *currentLevel* (*getM state*)
InvariantGetReasonIsReason (*getReason state*) (*getF state*) (*getM state*) (*set* (*getQ state*))
getBackjumpLevel state $> 0 \longrightarrow$ *getF state* $\neq [] \wedge$ *last* (*getF state*) $=$ *getC state*
shows
let state′ = applyBackjump state in
InvariantGetReasonIsReason (*getReason state′*) (*getF state′*) (*getM state′*) (*set* (*getQ state′*))

proof−
let *?l* = *getCl state*
let *?level* = *getBackjumpLevel state*
let *?prefix* = *prefixToLevel ?level* (*getM state*)
let *?state′* = *state*⦇ *getConflictFlag* := *False*, *getQ* := [], *getM* := *?prefix* ⦈
let *?state″* = *if 0 < ?level then setReason* (*opposite ?l*) (*length* (*getF state*) − *1*) *?state′ else ?state′*
let *?stateB* = *applyBackjump state*
have *InvariantGetReasonIsReason* (*getReason ?state′*) (*getF ?state′*) (*getM ?state′*) (*set* (*getQ ?state′*))
proof−
{
fix *l*::*Literal*
assume ∗: *l el* (*elements ?prefix*) ∧ ¬ *l el* (*decisions ?prefix*) ∧ *elementLevel l ?prefix* > *0*
hence *l el* (*elements* (*getM state*)) ∧ ¬ *l el* (*decisions* (*getM state*)) ∧ *elementLevel l* (*getM state*) > *0*
using ‹*InvariantUniq* (*getM state*)›
unfolding *InvariantUniq-def*

using *isPrefixPrefixToLevel*[*of ?level* (*getM state*)]
using *isPrefixElements*[*of ?prefix getM state*]
using *prefixIsSubset*[*of elements ?prefix elements* (*getM state*)]
using *markedElementsTrailMemPrefixAreMarkedElementsPrefix*[*of getM state ?prefix l*]
using *elementLevelPrefixElement*[*of l getBackjumpLevel state getM state*]
by *auto*

with *assms*
obtain *reason*
where *reason* < *length* (*getF state*) *isReason* (*nth* (*getF state*) *reason*) *l* (*elements* (*getM state*))
getReason state l = *Some reason*
unfolding *InvariantGetReasonIsReason-def*
by *auto*
hence ∃ *reason*. *getReason state l* = *Some reason* ∧
reason < *length* (*getF state*) ∧
isReason (*nth* (*getF state*) *reason*) *l* (*elements ?prefix*)
using *isReasonHoldsInPrefix*[*of l elements ?prefix elements* (*getM state*) *nth* (*getF state*) *reason*]
using *isPrefixPrefixToLevel*[*of ?level* (*getM state*)]
using *isPrefixElements*[*of ?prefix getM state*]
using ∗
by *auto*
}
thus *?thesis*
unfolding *InvariantGetReasonIsReason-def*
by *auto*
qed

let *?stateM* = *?state''* ⦇ *getM* := *getM ?state''* @ [(*opposite ?l*, *False*)] ⦈

have ∗∗: *getM ?stateM* = *?prefix* @ [(*opposite ?l*, *False*)]
getF ?stateM = *getF state*
getQ ?stateM = []
getWatchList ?stateM = *getWatchList state*
getWatch1 ?stateM = *getWatch1 state*
getWatch2 ?stateM = *getWatch2 state*
unfolding *setReason-def*
by *auto*

have *InvariantGetReasonIsReason* (*getReason ?stateM*) (*getF ?stateM*) (*getM ?stateM*) (*set* (*getQ ?stateM*))
proof−
{

fix *l*::*Literal*
assume ∗: *l el* (*elements* (*getM ?stateM*)) ∧ ¬ *l el* (*decisions* (*getM ?stateM*)) ∧ *elementLevel l* (*getM ?stateM*) > *0*

have *isPrefix ?prefix* (*getM ?stateM*)
unfolding *setReason-def*
unfolding *isPrefix-def*
by *auto*

have ∃ *reason. getReason ?stateM l* = *Some reason* ∧
reason < *length* (*getF ?stateM*) ∧
isReason (*nth* (*getF ?stateM*) *reason*) *l* (*elements* (*getM ?stateM*))
proof (*cases l* = *opposite ?l*)
case *False*
hence *l el* (*elements ?prefix*)
using ∗
using ∗∗
by *auto*
moreover
hence ¬ *l el* (*decisions ?prefix*)
using *elementLevelAppend*[*of l ?prefix* [(*opposite ?l*, *False*)]]
using ‹*isPrefix ?prefix* (*getM ?stateM*)›
using *markedElementsPrefixAreMarkedElementsTrail*[*of ?prefix getM ?stateM l*]
using ∗
using ∗∗
by *auto*
moreover
have *elementLevel l ?prefix* = *elementLevel l* (*getM ?stateM*)
using ‹*l el* (*elements ?prefix*)›
using ∗
using ∗∗
using *elementLevelAppend*[*of l ?prefix* [(*opposite ?l*, *False*)]]
by *auto*
hence *elementLevel l ?prefix* > *0*
using ∗
by *simp*
ultimately
obtain *reason*
where *reason* < *length* (*getF state*)
isReason (*nth* (*getF state*) *reason*) *l* (*elements ?prefix*)
getReason state l = *Some reason*
using ‹*InvariantGetReasonIsReason* (*getReason ?state'*) (*getF ?state'*) (*getM ?state'*) (*set* (*getQ ?state'*))›
unfolding *InvariantGetReasonIsReason-def*
by *auto*
moreover
have *getReason ?stateM l* = *getReason ?state' l*

using *False*
unfolding *setReason-def*
by *auto*
ultimately
show *?thesis*
using *isReasonAppend*[*of nth* (*getF state*) *reason l elements ?prefix* [*opposite ?l*]]
using ∗∗
by *auto*
next
case *True*
show *?thesis*
proof (*cases ?level* = *0*)
case *True*
hence *currentLevel* (*getM ?stateM*) = *0*
using *currentLevelPrefixToLevel*[*of 0 getM state*]
using ∗
unfolding *currentLevel-def*
by (*simp add*: *markedElementsAppend*)
hence *elementLevel l* (*getM ?stateM*) = *0*
using ‹*?level* = *0*›
using *elementLevelLeqCurrentLevel*[*of l getM ?stateM*]
by *simp*
with ∗
have *False*
by *simp*
thus *?thesis*
by *simp*
next
case *False*
let *?reason* = *length* (*getF state*) − *1*

have *getReason ?stateM l* = *Some ?reason*
using ‹*?level* ≠ *0*›
using ‹*l* = *opposite ?l*›
unfolding *setReason-def*
by *auto*
moreover
have (*nth* (*getF state*) *?reason*) = (*getC state*)
using ‹*?level* ≠ *0*›
using ‹*getBackjumpLevel state* > *0* ⟶ *getF state* ≠ [] ∧ *last* (*getF state*) = *getC state*›
using *last-conv-nth*[*of getF state*]
by *simp*

hence *isUnitClause* (*nth* (*getF state*) *?reason*) *l* (*elements ?prefix*)
using *assms*
using *applyBackjumpEffect*[*of state F0*]

```
          using ‹l = opposite ?l›
          by (simp add: Let-def)
        hence isReason (nth (getF state) ?reason) l (elements (getM
?stateM))
          using **
            using isUnitClauseIsReason[of nth (getF state) ?reason l
elements ?prefix [opposite ?l]]
          using ‹l = opposite ?l›
          by simp
        moreover
        have ?reason < length (getF state)
          using ‹?level ≠ 0›
           using ‹getBackjumpLevel state > 0 ⟶ getF state ≠ [] ∧
last (getF state) = getC state›
          by simp
        ultimately
        show ?thesis
          using ‹?level ≠ 0›
          using ‹l = opposite ?l›
          using **
          by auto
      qed
    qed
  }
  thus ?thesis
    unfolding InvariantGetReasonIsReason-def
    unfolding setReason-def
    by auto
qed
thus ?thesis
  using InvariantGetReasonIsReasonAfterNotifyWatches[of ?stateM
getWatchList ?stateM ?l ?l ?prefix False {} []]
  unfolding applyBackjump-def
  unfolding Let-def
  unfolding assertLiteral-def
  unfolding Let-def
  unfolding notifyWatches-def
  using **
  using assms
  unfolding InvariantWatchListsCharacterization-def
  unfolding InvariantWatchListsUniq-def
  unfolding InvariantWatchListsContainOnlyClausesFromF-def
  by auto
qed
```

lemma *InvariantsNoDecisionsWhenConflictNorUnitAfterApplyBack-jump-1*:
assumes

InvariantConsistent (*getM state*)
InvariantUniq (*getM state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
and
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*)
(*getF state*) **and**

InvariantUniqC (*getC state*)
getC state = [*opposite* (*getCl state*)]

InvariantNoDecisionsWhenConflict (*getF state*) (*getM state*) (*currentLevel*
(*getM state*))
InvariantNoDecisionsWhenUnit (*getF state*) (*getM state*) (*currentLevel*
(*getM state*))
InvariantCFalse (*getConflictFlag state*) (*getM state*) (*getC state*) **and**
InvariantCEntailed (*getConflictFlag state*) *F0* (*getC state*) **and**
InvariantClCharacterization (*getCl state*) (*getC state*) (*getM state*)
and
InvariantCllCharacterization (*getCl state*) (*getCll state*) (*getC state*)
(*getM state*) **and**
InvariantClCurrentLevel (*getCl state*) (*getM state*)

getConflictFlag state
isUIP (*opposite* (*getCl state*)) (*getC state*) (*getM state*)
currentLevel (*getM state*) > *0*
shows
let state′ = *applyBackjump state in*
InvariantNoDecisionsWhenConflict (*getF state′*) (*getM state′*)
(*currentLevel* (*getM state′*)) ∧
InvariantNoDecisionsWhenUnit (*getF state′*) (*getM state′*)
(*currentLevel* (*getM state′*))
proof−
let *?l* = *getCl state*
let *?bClause* = *getC state*
let *?bLiteral* = *opposite ?l*
let *?level* = *getBackjumpLevel state*
let *?prefix* = *prefixToLevel ?level* (*getM state*)
let *?state′* = *applyBackjump state*
have *getM ?state′* = *?prefix* @ [(*?bLiteral*, *False*)] *getF ?state′* =
getF state
using *assms*
using *applyBackjumpEffect*[*of state*]
by (*auto simp add*: *Let-def*)
show *?thesis*
proof−

have *?level* < *elementLevel ?l* (*getM state*)
using *assms*
using *isMinimalBackjumpLevelGetBackjumpLevel*[*of state*]

unfolding *isMinimalBackjumpLevel-def*
unfolding *isBackjumpLevel-def*
by (*simp add*: *Let-def*)
hence *?level* < *currentLevel* (*getM state*)
using *elementLevelLeqCurrentLevel*[*of ?l getM state*]
by *simp*

have *currentLevel* (*getM ?state'*) = *currentLevel ?prefix*
using ‹*getM ?state'* = *?prefix* @ [(*?bLiteral*, *False*)]›
using *markedElementsAppend*[*of ?prefix* [(*?bLiteral*, *False*)]]
unfolding *currentLevel-def*
by *simp*

hence *currentLevel* (*getM ?state'*) ≤ *?level*
using *currentLevelPrefixToLevel*[*of ?level getM state*]
by *simp*

show *?thesis*
proof−
{
fix *level*
assume *level* < *currentLevel* (*getM ?state'*)
hence *level* < *currentLevel ?prefix*
using ‹*currentLevel* (*getM ?state'*) = *currentLevel ?prefix*›
by *simp*
hence *prefixToLevel level* (*getM* (*applyBackjump state*)) = *prefixToLevel level ?prefix*
using ‹*getM ?state'* = *?prefix* @ [(*?bLiteral*, *False*)]›
using *prefixToLevelAppend*[*of level ?prefix* [(*?bLiteral*, *False*)]]
by *simp*
have *level* < *?level*
using ‹*level* < *currentLevel ?prefix*›
using ‹*currentLevel* (*getM ?state'*) ≤ *?level*›
using ‹*currentLevel* (*getM ?state'*) = *currentLevel ?prefix*›
by *simp*
have *prefixToLevel level* (*getM ?state'*) = *prefixToLevel level ?prefix*
using ‹*getM ?state'* = *?prefix* @ [(*?bLiteral*, *False*)]›
using *prefixToLevelAppend*[*of level ?prefix* [(*?bLiteral*, *False*)]]
using ‹*level* < *currentLevel ?prefix*›
by *simp*

hence ¬ *formulaFalse* (*getF ?state'*) (*elements* (*prefixToLevel level* (*getM ?state'*))) (**is** *?false*)
using ‹*InvariantNoDecisionsWhenConflict* (*getF state*) (*getM state*) (*currentLevel* (*getM state*))›
unfolding *InvariantNoDecisionsWhenConflict-def*
using ‹*level* < *?level*›
using ‹*?level* < *currentLevel* (*getM state*)›

using *prefixToLevelPrefixToLevelHigherLevel*[*of level ?level getM state*, *THEN sym*]
using ‹*getF ?state′* = *getF state*›
using ‹*prefixToLevel level* (*getM ?state′*) = *prefixToLevel level ?prefix*›
using *prefixToLevelPrefixToLevelHigherLevel*[*of level ?level getM state*, *THEN sym*]
by (*auto simp add*: *formulaFalseIffContainsFalseClause*)
moreover
have ¬ (∃ *clause literal*.
clause el (*getF ?state′*) ∧
isUnitClause clause literal (*elements* (*prefixToLevel level* (*getM ?state′*)))) (**is** *?unit*)
using ‹*InvariantNoDecisionsWhenUnit* (*getF state*) (*getM state*) (*currentLevel* (*getM state*))›
unfolding *InvariantNoDecisionsWhenUnit-def*
using ‹*level* < *?level*›
using ‹*?level* < *currentLevel* (*getM state*)›
using ‹*getF ?state′* = *getF state*›
using ‹*prefixToLevel level* (*getM ?state′*) = *prefixToLevel level ?prefix*›
using *prefixToLevelPrefixToLevelHigherLevel*[*of level ?level getM state*, *THEN sym*]
by *simp*
ultimately
have *?false* ∧ *?unit*
by *simp*
}
thus *?thesis*
unfolding *InvariantNoDecisionsWhenConflict-def*
unfolding *InvariantNoDecisionsWhenUnit-def*
by (*auto simp add*: *Let-def*)
qed
qed
qed

lemma *InvariantsNoDecisionsWhenConflictNorUnitAfterApplyBackjump-2*:
assumes
InvariantConsistent (*getM state*)
InvariantUniq (*getM state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**

InvariantUniqC (*getC state*)
getC state ≠ [*opposite* (*getCl state*)]

InvariantNoDecisionsWhenConflict (*butlast* (*getF state*)) (*getM state*) (*currentLevel* (*getM state*))
InvariantNoDecisionsWhenUnit (*butlast* (*getF state*)) (*getM state*) (*currentLevel* (*getM state*))
getF state ≠ [] *last* (*getF state*) = *getC state*
InvariantNoDecisionsWhenConflict [*getC state*] (*getM state*) (*getBackjumpLevel state*)
InvariantNoDecisionsWhenUnit [*getC state*] (*getM state*) (*getBackjumpLevel state*)

getConflictFlag state
InvariantCFalse (*getConflictFlag state*) (*getM state*) (*getC state*) **and**
InvariantCEntailed (*getConflictFlag state*) *F0* (*getC state*) **and**
InvariantClCharacterization (*getCl state*) (*getC state*) (*getM state*) **and**
InvariantCllCharacterization (*getCl state*) (*getCll state*) (*getC state*) (*getM state*) **and**
InvariantClCurrentLevel (*getCl state*) (*getM state*)

isUIP (*opposite* (*getCl state*)) (*getC state*) (*getM state*)
currentLevel (*getM state*) > *0*
shows
let state′ = *applyBackjump state in*
InvariantNoDecisionsWhenConflict (*getF state′*) (*getM state′*) (*currentLevel* (*getM state′*)) ∧
InvariantNoDecisionsWhenUnit (*getF state′*) (*getM state′*) (*currentLevel* (*getM state′*))
proof−
let *?l* = *getCl state*
let *?bClause* = *getC state*
let *?bLiteral* = *opposite ?l*
let *?level* = *getBackjumpLevel state*
let *?prefix* = *prefixToLevel ?level* (*getM state*)
let *?state′* = *applyBackjump state*
have *getM ?state′* = *?prefix* @ [(*?bLiteral*, *False*)] *getF ?state′* = *getF state*
using *assms*
using *applyBackjumpEffect*[*of state*]
by (*auto simp add*: *Let-def*)
show *?thesis*
proof−
have *?level* < *elementLevel ?l* (*getM state*)
using *assms*
using *isMinimalBackjumpLevelGetBackjumpLevel*[*of state*]
unfolding *isMinimalBackjumpLevel-def*
unfolding *isBackjumpLevel-def*
by (*simp add*: *Let-def*)
hence *?level* < *currentLevel* (*getM state*)
using *elementLevelLeqCurrentLevel*[*of ?l getM state*]

by *simp*

have *currentLevel* (*getM ?state'*) = *currentLevel ?prefix*
using ‹*getM ?state'* = *?prefix* @ [(*?bLiteral*, *False*)]›
using *markedElementsAppend*[*of ?prefix* [(*?bLiteral*, *False*)]]
unfolding *currentLevel-def*
by *simp*

hence *currentLevel* (*getM ?state'*) ≤ *?level*
using *currentLevelPrefixToLevel*[*of ?level getM state*]
by *simp*

show *?thesis*
proof−
{
fix *level*
assume *level* < *currentLevel* (*getM ?state'*)
hence *level* < *currentLevel ?prefix*
using ‹*currentLevel* (*getM ?state'*) = *currentLevel ?prefix*›
by *simp*
hence *prefixToLevel level* (*getM* (*applyBackjump state*)) = *prefixToLevel level ?prefix*
using ‹*getM ?state'* = *?prefix* @ [(*?bLiteral*, *False*)]›
using *prefixToLevelAppend*[*of level ?prefix* [(*?bLiteral*, *False*)]]
by *simp*
have *level* < *?level*
using ‹*level* < *currentLevel ?prefix*›
using ‹*currentLevel* (*getM ?state'*) ≤ *?level*›
using ‹*currentLevel* (*getM ?state'*) = *currentLevel ?prefix*›
by *simp*
have *prefixToLevel level* (*getM ?state'*) = *prefixToLevel level ?prefix*
using ‹*getM ?state'* = *?prefix* @ [(*?bLiteral*, *False*)]›
using *prefixToLevelAppend*[*of level ?prefix* [(*?bLiteral*, *False*)]]
using ‹*level* < *currentLevel ?prefix*›
by *simp*

have ¬ *formulaFalse* (*butlast* (*getF ?state'*)) (*elements* (*prefixToLevel level* (*getM ?state'*)))
using ‹*getF ?state'* = *getF state*›
using ‹*InvariantNoDecisionsWhenConflict* (*butlast* (*getF state*)) (*getM state*) (*currentLevel* (*getM state*))›
using ‹*level* < *?level*›
using ‹*?level* < *currentLevel* (*getM state*)›
using ‹*prefixToLevel level* (*getM ?state'*) = *prefixToLevel level ?prefix*›
using *prefixToLevelPrefixToLevelHigherLevel*[*of level ?level getM state*, *THEN sym*]
unfolding *InvariantNoDecisionsWhenConflict-def*

by (*auto simp add*: *formulaFalseIffContainsFalseClause*)
moreover
have ¬ *clauseFalse* (*last* (*getF ?state′*)) (*elements* (*prefixToLevel level* (*getM ?state′*)))
using ‹*getF ?state′* = *getF state*›
using ‹*InvariantNoDecisionsWhenConflict* [*getC state*] (*getM state*) (*getBackjumpLevel state*)›
using ‹*last* (*getF state*) = *getC state*›
using ‹*level* < *?level*›
using ‹*prefixToLevel level* (*getM ?state′*) = *prefixToLevel level ?prefix*›
using *prefixToLevelPrefixToLevelHigherLevel*[*of level ?level getM state*, *THEN sym*]
unfolding *InvariantNoDecisionsWhenConflict-def*
by (*simp add*: *formulaFalseIffContainsFalseClause*)
moreover
from ‹*getF state* ≠ []›
have *butlast* (*getF state*) @ [*last* (*getF state*)] = *getF state*
using *append-butlast-last-id*[*of getF state*]
by *simp*
hence *getF state* = *butlast* (*getF state*) @ [*last* (*getF state*)]
by (*rule sym*)
ultimately
have ¬ *formulaFalse* (*getF ?state′*) (*elements* (*prefixToLevel level* (*getM ?state′*))) (**is** *?false*)
using ‹*getF ?state′* = *getF state*›
using *set-append*[*of butlast* (*getF state*) [*last* (*getF state*)]]
by (*auto simp add*: *formulaFalseIffContainsFalseClause*)

have ¬ (∃ *clause literal*.
clause el (*butlast* (*getF ?state′*)) ∧
isUnitClause clause literal (*elements* (*prefixToLevel level* (*getM ?state′*))))
using ‹*InvariantNoDecisionsWhenUnit* (*butlast* (*getF state*)) (*getM state*) (*currentLevel* (*getM state*))›
unfolding *InvariantNoDecisionsWhenUnit-def*
using ‹*level* < *?level*›
using ‹*?level* < *currentLevel* (*getM state*)›
using ‹*getF ?state′* = *getF state*›
using ‹*prefixToLevel level* (*getM ?state′*) = *prefixToLevel level ?prefix*›
using *prefixToLevelPrefixToLevelHigherLevel*[*of level ?level getM state*, *THEN sym*]
by *simp*
moreover
have ¬ (∃ *l*. *isUnitClause* (*last* (*getF ?state′*)) *l* (*elements* (*prefixToLevel level* (*getM ?state′*))))
using ‹*getF ?state′* = *getF state*›
using ‹*InvariantNoDecisionsWhenUnit* [*getC state*] (*getM*

state) (*getBackjumpLevel state*)›
using ‹*last* (*getF state*) = *getC state*›
using ‹*level* < *?level*›
using ‹*prefixToLevel level* (*getM ?state'*) = *prefixToLevel level ?prefix*›
using *prefixToLevelPrefixToLevelHigherLevel*[*of level ?level getM state*, *THEN sym*]
unfolding *InvariantNoDecisionsWhenUnit-def*
by *simp*
moreover
from ‹*getF state* ≠ []›
have *butlast* (*getF state*) @ [*last* (*getF state*)] = *getF state*
using *append-butlast-last-id*[*of getF state*]
by *simp*
hence *getF state* = *butlast* (*getF state*) @ [*last* (*getF state*)]
by (*rule sym*)
ultimately
have ¬ (∃ *clause literal*.
clause el (*getF ?state'*) ∧
isUnitClause clause literal (*elements* (*prefixToLevel level* (*getM ?state'*)))) (**is** *?unit*)
using ‹*getF ?state'* = *getF state*›
using *set-append*[*of butlast* (*getF state*) [*last* (*getF state*)]]
by *auto*

have *?false* ∧ *?unit*
using ‹*?false*› ‹*?unit*›
by *simp*
}
thus *?thesis*
unfolding *InvariantNoDecisionsWhenConflict-def*
unfolding *InvariantNoDecisionsWhenUnit-def*
by (*auto simp add*: *Let-def*)
qed
qed
qed

lemma *InvariantEquivalentZLAfterApplyBackjump*:
assumes
InvariantConsistent (*getM state*)
InvariantUniq (*getM state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**

getConflictFlag state
InvariantUniqC (*getC state*)
InvariantCFalse (*getConflictFlag state*) (*getM state*) (*getC state*) **and**

InvariantCEntailed (*getConflictFlag state*) *F0* (*getC state*) **and**
InvariantClCharacterization (*getCl state*) (*getC state*) (*getM state*) **and**
InvariantCllCharacterization (*getCl state*) (*getCll state*) (*getC state*) (*getM state*) **and**
InvariantClCurrentLevel (*getCl state*) (*getM state*)
InvariantEquivalentZL (*getF state*) (*getM state*) *F0*

isUIP (*opposite* (*getCl state*)) (*getC state*) (*getM state*)
currentLevel (*getM state*) > *0*
shows
let state′ = applyBackjump state in
InvariantEquivalentZL (*getF state′*) (*getM state′*) *F0*

proof−

let *?l = getCl state*
let *?bClause = getC state*
let *?bLiteral = opposite ?l*
let *?level = getBackjumpLevel state*
let *?prefix = prefixToLevel ?level* (*getM state*)
let *?state′ = applyBackjump state*

have *formulaEntailsClause F0 ?bClause*
isUnitClause ?bClause ?bLiteral (*elements ?prefix*)
getM ?state′ = ?prefix @ [(*?bLiteral*, *False*)]
getF ?state′ = getF state
using *assms*
using *applyBackjumpEffect*[*of state F0*]
by (*auto simp add*: *Let-def*)
note ∗ = *this*
show *?thesis*
proof (*cases ?level = 0*)
case *False*
have *?level < elementLevel ?l* (*getM state*)
using *assms*
using *isMinimalBackjumpLevelGetBackjumpLevel*[*of state*]
unfolding *isMinimalBackjumpLevel-def*
unfolding *isBackjumpLevel-def*
by (*simp add*: *Let-def*)
hence *?level < currentLevel* (*getM state*)
using *elementLevelLeqCurrentLevel*[*of ?l getM state*]
by *simp*
hence *prefixToLevel 0* (*getM ?state′*) = *prefixToLevel 0 ?prefix*
using ∗
using *prefixToLevelAppend*[*of 0 ?prefix* [(*?bLiteral*, *False*)]]
using ‹*?level ≠ 0*›
using *currentLevelPrefixToLevelEq*[*of ?level getM state*]
by *simp*

hence *prefixToLevel 0* (*getM ?state′*) = *prefixToLevel 0* (*getM state*)
using ‹*?level* ≠ *0*›
using *prefixToLevelPrefixToLevelHigherLevel*[*of 0 ?level getM state*]
by *simp*
thus *?thesis*
using ∗
using ‹*InvariantEquivalentZL* (*getF state*) (*getM state*) *F0*›
unfolding *InvariantEquivalentZL-def*
by (*simp add*: *Let-def*)
next
case *True*
hence *prefixToLevel 0* (*getM ?state′*) = *?prefix* @ [(*?bLiteral*, *False*)]
using ∗
using *prefixToLevelAppend*[*of 0 ?prefix* [(*?bLiteral*, *False*)]]
using *currentLevelPrefixToLevel*[*of 0 getM state*]
by *simp*

let *?FM* = *getF state* @ *val2form* (*elements* (*prefixToLevel 0* (*getM state*)))
let *?FM′* = *getF ?state′* @ *val2form* (*elements* (*prefixToLevel 0* (*getM ?state′*)))

have *formulaEntailsValuation F0* (*elements ?prefix*)
using ‹*?level* = *0*›
using *val2formIsEntailed*[*of getF state elements* (*prefixToLevel 0* (*getM state*)) []]
using ‹*InvariantEquivalentZL* (*getF state*) (*getM state*) *F0*›
unfolding *formulaEntailsValuation-def*
unfolding *InvariantEquivalentZL-def*
unfolding *equivalentFormulae-def*
unfolding *formulaEntailsLiteral-def*
by *auto*

have *formulaEntailsLiteral* (*F0* @ *val2form* (*elements ?prefix*)) *?bLiteral*
using ∗
using *unitLiteralIsEntailed* [*of ?bClause ?bLiteral elements ?prefix F0*]
by *simp*

have *formulaEntailsLiteral F0 ?bLiteral*
proof−
{
fix *valuation*::*Valuation*
assume *model valuation F0*

hence *formulaTrue* (*val2form* (*elements ?prefix*)) *valuation*
using ‹*formulaEntailsValuation F0* (*elements ?prefix*)›
using *val2formFormulaTrue*[*of elements ?prefix valuation*]
unfolding *formulaEntailsValuation-def*
unfolding *formulaEntailsLiteral-def*
by *simp*
hence *formulaTrue* (*F0* @ (*val2form* (*elements ?prefix*))) *valuation*
using ‹*model valuation F0*›
by (*simp add*: *formulaTrueAppend*)
hence *literalTrue ?bLiteral valuation*
using ‹*model valuation F0*›
using ‹*formulaEntailsLiteral* (*F0* @ *val2form* (*elements ?prefix*)) *?bLiteral*›
unfolding *formulaEntailsLiteral-def*
by *auto*
}
thus *?thesis*
unfolding *formulaEntailsLiteral-def*
by *simp*
qed

hence *formulaEntailsClause F0* [*?bLiteral*]
unfolding *formulaEntailsLiteral-def*
unfolding *formulaEntailsClause-def*
by (*auto simp add*: *clauseTrueIffContainsTrueLiteral*)

hence *formulaEntailsClause ?FM* [*?bLiteral*]
using ‹*InvariantEquivalentZL* (*getF state*) (*getM state*) *F0*›
unfolding *InvariantEquivalentZL-def*
unfolding *equivalentFormulae-def*
unfolding *formulaEntailsClause-def*
by *auto*

have *?FM′* = *?FM* @ [[*?bLiteral*]]
using ∗
using ‹*?level* = *0*›
using ‹*prefixToLevel 0* (*getM ?state′*) = *?prefix* @ [(*?bLiteral*, *False*)]›
by (*auto simp add*: *val2formAppend*)

show *?thesis*
using ‹*InvariantEquivalentZL* (*getF state*) (*getM state*) *F0*›
using ‹*?FM′* = *?FM* @ [[*?bLiteral*]]›
using ‹*formulaEntailsClause ?FM* [*?bLiteral*]›
unfolding *InvariantEquivalentZL-def*
using *extendEquivalentFormulaWithEntailedClause*[*of F0 ?FM* [*?bLiteral*]]
by (*simp add*: *equivalentFormulaeSymmetry*)

qed
qed

lemma *InvariantsVarsAfterApplyBackjump*:
assumes
InvariantConsistent (*getM state*)
InvariantUniq (*getM state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
and
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*)
(*getF state*) **and**

InvariantWatchListsUniq (*getWatchList state*)
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*) **and**

getConflictFlag state
InvariantCFalse (*getConflictFlag state*) (*getM state*) (*getC state*) **and**
InvariantUniqC (*getC state*) **and**
InvariantCEntailed (*getConflictFlag state*) *F0′* (*getC state*) **and**
InvariantClCharacterization (*getCl state*) (*getC state*) (*getM state*)
and
InvariantCllCharacterization (*getCl state*) (*getCll state*) (*getC state*)
(*getM state*) **and**
InvariantClCurrentLevel (*getCl state*) (*getM state*)
InvariantEquivalentZL (*getF state*) (*getM state*) *F0′*

isUIP (*opposite* (*getCl state*)) (*getC state*) (*getM state*)
currentLevel (*getM state*) > *0*

vars F0′ ⊆ *vars F0*

InvariantVarsM (*getM state*) *F0 Vbl*
InvariantVarsF (*getF state*) *F0 Vbl*
InvariantVarsQ (*getQ state*) *F0 Vbl*
shows
let state′ = *applyBackjump state in*
InvariantVarsM (*getM state′*) *F0 Vbl* ∧
InvariantVarsF (*getF state′*) *F0 Vbl* ∧
InvariantVarsQ (*getQ state′*) *F0 Vbl*

proof−

let *?l* = *getCl state*

let *?bClause* = *getC state*
let *?bLiteral* = *opposite ?l*
let *?level* = *getBackjumpLevel state*
let *?prefix* = *prefixToLevel ?level* (*getM state*)
let *?state'* = *state*(| *getConflictFlag* := *False*, *getQ* := [], *getM* := *?prefix* |)
let *?state''* = *setReason* (*opposite* (*getCl state*)) (*length* (*getF state*) − *1*) *?state'*
let *?stateB* = *applyBackjump state*

have *formulaEntailsClause F0' ?bClause*
isUnitClause ?bClause ?bLiteral (*elements ?prefix*)
getM ?stateB = *?prefix* @ [(*?bLiteral*, *False*)]
getF ?stateB = *getF state*
using *assms*
using *applyBackjumpEffect*[*of state F0'*]
by (*auto simp add*: *Let-def*)
note ∗ = *this*

have *var ?bLiteral* ∈ *vars F0* ∪ *Vbl*
proof−
have *vars* (*getC state*) ⊆ *vars* (*elements* (*getM state*))
using ‹*getConflictFlag state*›
using ‹*InvariantCFalse* (*getConflictFlag state*) (*getM state*) (*getC state*)›
using *valuationContainsItsFalseClausesVariables*[*of getC state elements* (*getM state*)]
unfolding *InvariantCFalse-def*
by *simp*
moreover
have *?bLiteral el* (*getC state*)
using ‹*InvariantClCharacterization* (*getCl state*) (*getC state*) (*getM state*)›
unfolding *InvariantClCharacterization-def*
unfolding *isLastAssertedLiteral-def*
using *literalElListIffOppositeLiteralElOppositeLiteralList*[*of ?bLiteral getC state*]
by *simp*
ultimately
show *?thesis*
using ‹*InvariantVarsM* (*getM state*) *F0 Vbl*›
using ‹*vars F0'* ⊆ *vars F0*›
unfolding *InvariantVarsM-def*
using *clauseContainsItsLiteralsVariable*[*of ?bLiteral getC state*]
by *auto*
qed

hence *InvariantVarsM* (*getM ?stateB*) *F0 Vbl*
using ‹*InvariantVarsM* (*getM state*) *F0 Vbl*›

using *InvariantVarsMAfterBackjump*[*of getM state F0 Vbl ?prefix ?bLiteral getM ?stateB*]
using ∗
by (*simp add*: *isPrefixPrefixToLevel*)
moreover
have *InvariantConsistent* (*prefixToLevel* (*getBackjumpLevel state*) (*getM state*) @ [(*opposite* (*getCl state*), *False*)])
InvariantUniq (*prefixToLevel* (*getBackjumpLevel state*) (*getM state*) @ [(*opposite* (*getCl state*), *False*)])
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*prefixToLevel* (*getBackjumpLevel state*) (*getM state*))
using *assms*
using *InvariantConsistentAfterApplyBackjump*[*of state F0'*]
using *InvariantUniqAfterApplyBackjump*[*of state F0'*]
using ∗
using *InvariantWatchCharacterizationInBackjumpPrefix*[*of state*]
by (*auto simp add*: *Let-def*)
hence *InvariantVarsQ* (*getQ ?stateB*) *F0 Vbl*
using ‹*InvariantVarsF* (*getF state*) *F0 Vbl*›
using ‹*InvariantWatchListsContainOnlyClausesFromF* (*getWatchList state*) (*getF state*)›
using ‹*InvariantWatchListsUniq* (*getWatchList state*)›
using ‹*InvariantWatchListsCharacterization* (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)›
using ‹*InvariantWatchesEl* (*getF state*) (*getWatch1 state*) (*getWatch2 state*)›
using ‹*InvariantWatchesDiffer* (*getF state*) (*getWatch1 state*) (*getWatch2 state*)›
using *InvariantVarsQAfterAssertLiteral*[*of if ?level > 0 then ?state'' else ?state' ?bLiteral False F0 Vbl*]
unfolding *applyBackjump-def*
unfolding *InvariantVarsQ-def*
unfolding *setReason-def*
by (*auto simp add*: *Let-def*)
moreover
have *InvariantVarsF* (*getF ?stateB*) *F0 Vbl*
using *assms*
using *assertLiteralEffect*[*of if ?level > 0 then ?state'' else ?state' ?bLiteral False*]
using ‹*InvariantVarsF* (*getF state*) *F0 Vbl*›
unfolding *applyBackjump-def*
unfolding *setReason-def*
by (*simp add*: *Let-def*)
ultimately
show *?thesis*
by (*simp add*: *Let-def*)
qed

end

theory *Decide*
imports *AssertLiteral*
begin

lemma *applyDecideEffect*:
assumes
$\neg$ *vars*(*elements* (*getM state*)) $\supseteq$ *Vbl* **and**
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
and
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*)
(*getF state*)
shows
let literal = *selectLiteral state Vbl in*
let state' = *applyDecide state Vbl in*
var literal $\notin$ *vars* (*elements* (*getM state*)) $\wedge$
var literal $\in$ *Vbl* $\wedge$
getM state' = *getM state* @ [(*literal*, *True*)] $\wedge$
getF state' = *getF state*
using *assms*
using *selectLiteral-def*[*of Vbl state*]
unfolding *applyDecide-def*
using *assertLiteralEffect*[*of state selectLiteral state Vbl True*]
by (*simp add*: *Let-def*)

lemma *InvariantConsistentAfterApplyDecide*:
assumes
$\neg$ *vars*(*elements* (*getM state*)) $\supseteq$ *Vbl* **and**
InvariantConsistent (*getM state*) **and**
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
and
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*)
(*getF state*)
shows
let state' = *applyDecide state Vbl in*
InvariantConsistent (*getM state'*)
using *assms*
using *applyDecideEffect*[*of Vbl state*]
using *InvariantConsistentAfterDecide*[*of getM state selectLiteral state Vbl getM* (*applyDecide state Vbl*)]
by (*simp add*: *Let-def*)

lemma *InvariantUniqAfterApplyDecide*:

assumes
¬ *vars*(*elements* (*getM state*)) ⊇ *Vbl* **and**
InvariantUniq (*getM state*) **and**
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
and
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*)
(*getF state*)
shows
let state′ = *applyDecide state Vbl in*
InvariantUniq (*getM state′*)
using *assms*
using *applyDecideEffect*[*of Vbl state*]
using *InvariantUniqAfterDecide*[*of getM state selectLiteral state Vbl*
getM (*applyDecide state Vbl*)]
by (*simp add*: *Let-def*)

lemma *InvariantQCharacterizationAfterApplyDecide*:
assumes
¬ *vars*(*elements* (*getM state*)) ⊇ *Vbl* **and**

InvariantConsistent (*getM state*) **and**
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*)
(*getF state*)
InvariantWatchListsUniq (*getWatchList state*)
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1*
state) (*getWatch2 state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2*
state)
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2*
state) (*getM state*)
InvariantConflictFlagCharacterization (*getConflictFlag state*) (*getF*
state) (*getM state*)
InvariantQCharacterization (*getConflictFlag state*) (*getQ state*) (*getF*
state) (*getM state*)

getQ state = []
shows
let state′ = *applyDecide state Vbl in*
InvariantQCharacterization (*getConflictFlag state′*) (*getQ state′*)
(*getF state′*) (*getM state′*)
proof−
let *?state′* = *applyDecide state Vbl*
let *?literal* = *selectLiteral state Vbl*
have *getM ?state′* = *getM state* @ [(*?literal*, *True*)]
using *assms*
using *applyDecideEffect*[*of Vbl state*]
by (*simp add*: *Let-def*)
hence *InvariantConsistent* (*getM state* @ [(*?literal*, *True*)])

using *InvariantConsistentAfterApplyDecide*[*of Vbl state*]
using *assms*
by (*simp add*: *Let-def*)
thus *?thesis*
using *assms*
using *InvariantQCharacterizationAfterAssertLiteralNotInQ*[*of state ?literal True*]
unfolding *applyDecide-def*
by *simp*
qed

lemma *InvariantEquivalentZLAfterApplyDecide*:
assumes
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantEquivalentZL (*getF state*) (*getM state*) *F0*
shows
let state′ = *applyDecide state Vbl in*
InvariantEquivalentZL (*getF state′*) (*getM state′*) *F0*
proof−
let *?state′* = *applyDecide state Vbl*
let *?l* = *selectLiteral state Vbl*

have *getM ?state′* = *getM state* @ [(*?l*, *True*)]
getF ?state′ = *getF state*
unfolding *applyDecide-def*
using *assertLiteralEffect*[*of state ?l True*]
using *assms*
by (*auto simp only*: *Let-def*)
have *prefixToLevel 0* (*getM ?state′*) = *prefixToLevel 0* (*getM state*)
proof (*cases currentLevel* (*getM state*) > *0*)
case *True*
thus *?thesis*
using *prefixToLevelAppend*[*of 0 getM state* [(*?l*, *True*)]]
using ‹*getM ?state′* = *getM state* @ [(*?l*, *True*)]›
by *auto*
next
case *False*
hence *prefixToLevel 0* (*getM state* @ [(*?l*, *True*)]) =
getM state @ (*prefixToLevel-aux* [(*?l*, *True*)] *0* (*currentLevel* (*getM state*)))
using *prefixToLevelAppend*[*of 0 getM state* [(*?l*, *True*)]]
by *simp*
hence *prefixToLevel 0* (*getM state* @ [(*?l*, *True*)]) = *getM state*
by *simp*
thus *?thesis*
using ‹*getM ?state′* = *getM state* @ [(*?l*, *True*)]›
using *currentLevelZeroTrailEqualsItsPrefixToLevelZero*[*of getM*

state]
using *False*
by *simp*
qed
thus *?thesis*
using ‹*InvariantEquivalentZL* (*getF state*) (*getM state*) *F0*›
unfolding *InvariantEquivalentZL-def*
using ‹*getF ?state′ = getF state*›
by *simp*
qed

lemma *InvariantGetReasonIsReasonAfterApplyDecide*:
assumes
¬ *vars* (*elements* (*getM state*)) ⊇ *Vbl*
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*)
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchListsUniq (*getWatchList state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantGetReasonIsReason (*getReason state*) (*getF state*) (*getM state*) (*set* (*getQ state*))
getQ state = []
shows
let state′ = applyDecide state Vbl in
InvariantGetReasonIsReason (*getReason state′*) (*getF state′*) (*getM state′*) (*set* (*getQ state′*))
proof−
let *?l = selectLiteral state Vbl*
let *?stateM = state* ⦇ *getM := getM state* @ [(*?l*, *True*)] ⦈
have *InvariantGetReasonIsReason* (*getReason ?stateM*) (*getF ?stateM*) (*getM ?stateM*) (*set* (*getQ ?stateM*))
proof−
{
fix *l*::*Literal*
assume ∗: *l el* (*elements* (*getM ?stateM*)) ¬ *l el* (*decisions* (*getM ?stateM*)) *elementLevel l* (*getM ?stateM*) > *0*
have ∃ *reason*. *getReason ?stateM l = Some reason* ∧
0 ≤ *reason* ∧ *reason* < *length* (*getF ?stateM*) ∧
isReason (*getF ?stateM* ! *reason*) *l* (*elements* (*getM ?stateM*))
proof (*cases l el* (*elements* (*getM state*)))
case *True*
moreover
hence ¬ *l el* (*decisions* (*getM state*))
using ∗
by (*simp add*: *markedElementsAppend*)
moreover
have *elementLevel l* (*getM state*) > *0*

proof−
{
assume ¬ *?thesis*
with ∗
have *l* = *?l*
using *True*
using *elementLevelAppend*[*of l getM state* [(*?l*, *True*)]]
by *simp*
hence *var ?l* ∈ *vars* (*elements* (*getM state*))
using *True*
using *valuationContainsItsLiteralsVariable*[*of l elements* (*getM state*)]
by *simp*
hence *False*
using ‹¬ *vars* (*elements* (*getM state*)) ⊇ *Vbl*›
using *selectLiteral-def*[*of Vbl state*]
by *auto*
} **thus** *?thesis*
by *auto*
qed
ultimately
obtain *reason*
where *getReason state l* = *Some reason* ∧
0 ≤ *reason* ∧ *reason* < *length* (*getF state*) ∧
isReason (*getF state* ! *reason*) *l* (*elements* (*getM state*))
using ‹*InvariantGetReasonIsReason* (*getReason state*) (*getF state*) (*getM state*) (*set* (*getQ state*))›
unfolding *InvariantGetReasonIsReason-def*
by *auto*
thus *?thesis*
using *isReasonAppend*[*of nth* (*getF ?stateM*) *reason l elements* (*getM state*) [*?l*]]
by *auto*
next
case *False*
hence *l* = *?l*
using ∗
by *auto*
hence *l el* (*decisions* (*getM ?stateM*))
using *markedElementIsMarkedTrue*[*of l getM ?stateM*]
by *auto*
with ∗
have *False*
by *auto*
thus *?thesis*
by *simp*
qed
}
thus *?thesis*

using ‹*getQ state* = []›
unfolding *InvariantGetReasonIsReason-def*
by *auto*
qed
thus *?thesis*
using *assms*
using *InvariantGetReasonIsReasonAfterNotifyWatches*[*of ?stateM getWatchList ?stateM* (*opposite ?l*)
opposite ?l getM state True {} []]
unfolding *applyDecide-def*
unfolding *assertLiteral-def*
unfolding *notifyWatches-def*
unfolding *InvariantWatchListsCharacterization-def*
unfolding *InvariantWatchListsContainOnlyClausesFromF-def*
unfolding *InvariantWatchListsUniq-def*
using ‹*getQ state* = []›
by (*simp add*: *Let-def*)
qed

lemma *InvariantsVarsAfterApplyDecide*:
assumes
¬ *vars* (*elements* (*getM state*)) ⊇ *Vbl*
InvariantConsistent (*getM state*)
InvariantUniq (*getM state*)
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*)
InvariantWatchListsUniq (*getWatchList state*)
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*)

InvariantVarsM (*getM state*) *F0 Vbl*
InvariantVarsF (*getF state*) *F0 Vbl*
getQ state = []
shows
let state′ = *applyDecide state Vbl in*
InvariantVarsM (*getM state′*) *F0 Vbl* ∧
InvariantVarsF (*getF state′*) *F0 Vbl* ∧
InvariantVarsQ (*getQ state′*) *F0 Vbl*
proof−
let *?state′* = *applyDecide state Vbl*
let *?l* = *selectLiteral state Vbl*

have *InvariantVarsM* (*getM ?state′*) *F0 Vbl InvariantVarsF* (*getF ?state′*) *F0 Vbl*

using *assms*
using *applyDecideEffect*[*of Vbl state*]
using *varsAppendValuation*[*of elements* (*getM state*) [*?l*]]
unfolding *InvariantVarsM-def*
by (*auto simp add*: *Let-def*)
moreover
have *InvariantVarsQ* (*getQ ?state′*) *F0 Vbl*
using *InvariantVarsQAfterAssertLiteral*[*of state ?l True F0 Vbl*]
using *assms*
using *InvariantConsistentAfterApplyDecide*[*of Vbl state*]
using *InvariantUniqAfterApplyDecide*[*of Vbl state*]
using *assertLiteralEffect*[*of state ?l True*]
unfolding *applyDecide-def*
unfolding *InvariantVarsQ-def*
by (*simp add*: *Let-def*)
ultimately
show *?thesis*
by (*simp add*: *Let-def*)
qed

end

theory *SolveLoop*
imports *UnitPropagate ConflictAnalysis Decide*
begin

lemma *soundnessForUNSAT*:
assumes
equivalentFormulae (*F* @ *val2form M*) *F0*
formulaFalse F M
shows
¬ *satisfiable F0*
proof−
have *formulaEntailsValuation* (*F* @ *val2form M*) *M*
using *val2formIsEntailed*[*of F M* []]
by *simp*
moreover
have *formulaFalse* (*F* @ *val2form M*) *M*
using ‹*formulaFalse F M*›
by (*simp add*: *formulaFalseAppend*)
ultimately
have ¬ *satisfiable* (*F* @ *val2form M*)
using *formulaFalseInEntailedValuationIsUnsatisfiable*[*of F* @ *val2form*

M M]
  by *simp*
 thus *?thesis*
  using ‹*equivalentFormulae* (*F* @ *val2form M*) *F0*›
  by (*simp add*: *satisfiableEquivalent*)
qed

lemma *soundnessForSat*:
 fixes *F0* :: *Formula* **and** *F* :: *Formula* **and** *M*::*LiteralTrail*
 assumes *vars F0* ⊆ *Vbl* **and** *InvariantVarsF F F0 Vbl* **and** *InvariantConsistent M* **and** *InvariantEquivalentZL F M F0* **and**
 ¬ *formulaFalse F* (*elements M*) **and** *vars* (*elements M*) ⊇ *Vbl*
 shows *model* (*elements M*) *F0*
proof−
 from ‹*InvariantConsistent M*›
 have *consistent* (*elements M*)
  unfolding *InvariantConsistent-def*
  .
 moreover
 from ‹*InvariantVarsF F F0 Vbl*›
 have *vars F* ⊆ *vars F0* ∪ *Vbl*
  unfolding *InvariantVarsF-def*
  .
 with ‹*vars F0* ⊆ *Vbl*›
 have *vars F* ⊆ *Vbl*
  by *auto*
 with ‹*vars* (*elements M*) ⊇ *Vbl*›
 have *vars F* ⊆ *vars* (*elements M*)
  by *simp*
 hence *formulaTrue F* (*elements M*) ∨ *formulaFalse F* (*elements M*)
  by (*simp add*:*totalValuationForFormulaDefinesItsValue*)
 with ‹¬ *formulaFalse F* (*elements M*)›
 have *formulaTrue F* (*elements M*)
  by *simp*
 ultimately
 have *model* (*elements M*) *F*
  by *simp*
 moreover
 obtain *s*
  where *elements* (*prefixToLevel 0 M*) @ *s* = *elements M*
  using *isPrefixPrefixToLevel*[*of 0 M*]
  using *isPrefixElements*[*of prefixToLevel 0 M M*]
  unfolding *isPrefix-def*
  by *auto*
 hence *elements M* = *elements* (*prefixToLevel 0 M*) @ *s*
  by (*rule sym*)
 hence *formulaTrue* (*val2form* (*elements* (*prefixToLevel 0 M*))) (*elements M*)
  using *val2formFormulaTrue*[*of elements* (*prefixToLevel 0 M*) *ele-*

ments M]
by *auto*
hence *model* (*elements M*) (*val2form* (*elements* (*prefixToLevel 0 M*)))
using ‹*consistent* (*elements M*)›
by *simp*
ultimately
show *?thesis*
using ‹*InvariantEquivalentZL F M F0*›
unfolding *InvariantEquivalentZL-def*
unfolding *equivalentFormulae-def*
using *formulaTrueAppend*[*of F val2form* (*elements* (*prefixToLevel 0 M*)) *elements M*]
by *auto*
qed

definition
satFlagLessState = {(*state1*::*State*, *state2*::*State*). (*getSATFlag state1*) $\neq$ *UNDEF* $\wedge$ (*getSATFlag state2*) = *UNDEF*}

lemma *wellFoundedSatFlagLessState*:
shows *wf satFlagLessState*
unfolding *wf-eq-minimal*
proof−
show $\forall$ *Q state*. *state* $\in$ *Q* $\longrightarrow$ ($\exists$ *stateMin*$\in$*Q*. $\forall$ *state'*. (*state'*, *stateMin*) $\in$ *satFlagLessState* $\longrightarrow$ *state'* $\notin$ *Q*)
proof−
{
fix *state*::*State* **and** *Q*::*State set*
assume *state* $\in$ *Q*
have $\exists$ *stateMin*$\in$*Q*. $\forall$ *state'*. (*state'*, *stateMin*) $\in$ *satFlagLessState* $\longrightarrow$ *state'* $\notin$ *Q*
proof (*cases* $\exists$ *stateDef* $\in$ *Q*. (*getSATFlag stateDef*) $\neq$ *UNDEF*)
case *True*
then obtain *stateDef* **where** *stateDef* $\in$ *Q* (*getSATFlag stateDef*) $\neq$ *UNDEF*
by *auto*
have $\forall$ *state'*. (*state'*, *stateDef*) $\in$ *satFlagLessState* $\longrightarrow$ *state'* $\notin$ *Q*
proof
fix *state'*
show (*state'*, *stateDef*) $\in$ *satFlagLessState* $\longrightarrow$ *state'* $\notin$ *Q*
proof
assume (*state'*, *stateDef*) $\in$ *satFlagLessState*
hence *getSATFlag stateDef* = *UNDEF*
unfolding *satFlagLessState-def*
by *auto*
with ‹*getSATFlag stateDef* $\neq$ *UNDEF*› **have** *False*

by *simp*
thus *state′* ∉ *Q*
by *simp*
qed
qed
with ‹*stateDef* ∈ *Q*›
show *?thesis*
by *auto*
next
case *False*
have ∀ *state′*. (*state′*, *state*) ∈ *satFlagLessState* ⟶ *state′* ∉ *Q*
proof
fix *state′*
show (*state′*, *state*) ∈ *satFlagLessState* ⟶ *state′* ∉ *Q*
proof
assume (*state′*, *state*) ∈ *satFlagLessState*
hence *getSATFlag state′* ≠ *UNDEF*
unfolding *satFlagLessState-def*
by *simp*
with *False*
show *state′* ∉ *Q*
by *auto*
qed
qed
with ‹*state* ∈ *Q*›
show *?thesis*
by *auto*
qed
}
thus *?thesis*
by *auto*
qed
qed

definition
lexLessState1 Vbl = {(*state1*::*State*, *state2*::*State*).
getSATFlag state1 = *UNDEF* ∧ *getSATFlag state2* = *UNDEF* ∧
(*getM state1*, *getM state2*) ∈ *lexLessRestricted Vbl*
}

lemma *wellFoundedLexLessState1*:
assumes
finite Vbl
shows
wf (*lexLessState1 Vbl*)
unfolding *wf-eq-minimal*
proof−
show ∀ *Q state*. *state* ∈ *Q* ⟶ (∃ *stateMin*∈*Q*. ∀ *state′*. (*state′*, *stateMin*) ∈ *lexLessState1 Vbl* ⟶ *state′* ∉ *Q*)

proof−
{
fix *Q* :: *State set* **and** *state* :: *State*
assume *state* ∈ *Q*
let *?Q1* = {*M*::*LiteralTrail*. ∃ *state*. *state* ∈ *Q* ∧ *getSATFlag* *state* = *UNDEF* ∧ (*getM state*) = *M*}
have ∃ *stateMin* ∈ *Q*. (∀ *state′*. (*state′*, *stateMin*) ∈ *lexLessState1* *Vbl* ⟶ *state′* ∉ *Q*)
proof (*cases ?Q1* ≠ {})
case *True*
then obtain *M*::*LiteralTrail*
where *M* ∈ *?Q1*
by *auto*
then obtain *MMin*::*LiteralTrail*
where *MMin* ∈ *?Q1* ∀ *M′*. (*M′*, *MMin*) ∈ *lexLessRestricted* *Vbl* ⟶ *M′* ∉ *?Q1*
using *wfLexLessRestricted*[*of Vbl*] ‹*finite Vbl*›
unfolding *wf-eq-minimal*
apply *simp*
apply (*erule-tac x*=*?Q1* **in** *allE*)
by *auto*
from ‹*MMin* ∈ *?Q1*› **obtain** *stateMin*
where *stateMin* ∈ *Q* (*getM stateMin*) = *MMin* *getSATFlag* *stateMin* = *UNDEF*
by *auto*
have ∀ *state′*. (*state′*, *stateMin*) ∈ *lexLessState1 Vbl* ⟶ *state′* ∉ *Q*
proof
fix *state′*
show (*state′*, *stateMin*) ∈ *lexLessState1 Vbl* ⟶ *state′* ∉ *Q*
proof
assume (*state′*, *stateMin*) ∈ *lexLessState1 Vbl*
hence *getSATFlag state′* = *UNDEF* (*getM state′*, *getM* *stateMin*) ∈ *lexLessRestricted Vbl*
unfolding *lexLessState1-def*
by *auto*
hence *getM state′* ∉ *?Q1*
using ‹∀ *M′*. (*M′*, *MMin*) ∈ *lexLessRestricted Vbl* ⟶ *M′* ∉ *?Q1*›
using ‹(*getM stateMin*) = *MMin*›
by *auto*
thus *state′* ∉ *Q*
using ‹*getSATFlag state′* = *UNDEF*›
by *auto*
qed
qed
thus *?thesis*
using ‹*stateMin* ∈ *Q*›
by *auto*

```
    next
      case False
     have ∀ state'. (state', state) ∈ lexLessState1 Vbl ⟶ state' ∉ Q
      proof
        fix state'
        show (state', state) ∈ lexLessState1 Vbl ⟶ state' ∉ Q
        proof
          assume (state', state) ∈ lexLessState1 Vbl
          hence getSATFlag state = UNDEF
            unfolding lexLessState1-def
            by simp
          hence (getM state) ∈ ?Q1
            using ‹state ∈ Q›
            by auto
          hence False
            using False
            by auto
          thus state' ∉ Q
            by simp
        qed
      qed
      thus ?thesis
        using ‹state ∈ Q›
        by auto
    qed
  }
  thus ?thesis
    by auto
 qed
qed

definition
terminationLessState1 Vbl = {(state1::State, state2::State).
  (state1, state2) ∈ satFlagLessState ∨
  (state1, state2) ∈ lexLessState1 Vbl}

lemma wellFoundedTerminationLessState1:
  assumes finite Vbl
  shows wf (terminationLessState1 Vbl)
unfolding wf-eq-minimal
proof−
  show ∀ Q state. state ∈ Q ⟶ (∃ stateMin ∈ Q. ∀ state'. (state',
stateMin) ∈ terminationLessState1 Vbl ⟶ state' ∉ Q)
  proof−
    {
      fix Q::State set
      fix state::State
      assume state ∈ Q
       have ∃ stateMin∈Q. ∀ state'. (state', stateMin) ∈ termination-
```

LessState1 Vbl ⟶ *state'* ∉ *Q*
proof−
obtain *state0*
where *state0* ∈ *Q* ∀ *state'*. (*state'*, *state0*) ∈ *satFlagLessState* ⟶ *state'* ∉ *Q*
using *wellFoundedSatFlagLessState*
unfolding *wf-eq-minimal*
using ‹*state* ∈ *Q*›
by *auto*
show *?thesis*
proof (*cases getSATFlag state0* = *UNDEF*)
case *False*
hence ∀ *state'*. (*state'*, *state0*) ∈ *terminationLessState1 Vbl* ⟶ *state'* ∉ *Q*
using ‹∀ *state'*. (*state'*, *state0*) ∈ *satFlagLessState* ⟶ *state'* ∉ *Q*›
unfolding *terminationLessState1-def*
unfolding *lexLessState1-def*
by *simp*
thus *?thesis*
using ‹*state0* ∈ *Q*›
by *auto*
next
case *True*
then obtain *state1*
where *state1* ∈ *Q* ∀ *state'*. (*state'*, *state1*) ∈ *lexLessState1 Vbl* ⟶ *state'* ∉ *Q*
using ‹*finite Vbl*›
using ‹*state* ∈ *Q*›
using *wellFoundedLexLessState1*[*of Vbl*]
unfolding *wf-eq-minimal*
by *auto*

have ∀ *state'*. (*state'*, *state1*) ∈ *terminationLessState1 Vbl* ⟶ *state'* ∉ *Q*
using ‹∀ *state'*. (*state'*, *state1*) ∈ *lexLessState1 Vbl* ⟶ *state'* ∉ *Q*›
unfolding *terminationLessState1-def*
using ‹∀ *state'*. (*state'*, *state0*) ∈ *satFlagLessState* ⟶ *state'* ∉ *Q*›
using *True*
unfolding *satFlagLessState-def*
by *simp*
thus *?thesis*
using ‹*state1* ∈ *Q*›
by *auto*
qed
qed
}

thus *?thesis*
by *auto*
qed
qed

lemma *transTerminationLessState1*:
trans (*terminationLessState1 Vbl*)
proof−
{
fix *x*::*State* **and** *y*::*State* **and** *z*::*State*
assume (*x*, *y*) ∈ *terminationLessState1 Vbl* (*y*, *z*) ∈ *termination-LessState1 Vbl*
have (*x*, *z*) ∈ *terminationLessState1 Vbl*
proof (*cases* (*x*, *y*) ∈ *satFlagLessState*)
case *True*
hence *getSATFlag x* ≠ *UNDEF getSATFlag y* = *UNDEF*
unfolding *satFlagLessState-def*
by *auto*
hence *getSATFlag z* = *UNDEF*
using ‹(*y*, *z*) ∈ *terminationLessState1 Vbl*›
unfolding *terminationLessState1-def*
unfolding *satFlagLessState-def*
unfolding *lexLessState1-def*
by *auto*
thus *?thesis*
using ‹*getSATFlag x* ≠ *UNDEF*›
unfolding *terminationLessState1-def*
unfolding *satFlagLessState-def*
by *simp*
next
case *False*
with ‹(*x*, *y*) ∈ *terminationLessState1 Vbl*›
have *getSATFlag x* = *UNDEF getSATFlag y* = *UNDEF* (*getM x*, *getM y*) ∈ *lexLessRestricted Vbl*
unfolding *terminationLessState1-def*
unfolding *lexLessState1-def*
by *auto*
hence *getSATFlag z* = *UNDEF* (*getM y*, *getM z*) ∈ *lexLessRestricted Vbl*
using ‹(*y*, *z*) ∈ *terminationLessState1 Vbl*›
unfolding *terminationLessState1-def*
unfolding *satFlagLessState-def*
unfolding *lexLessState1-def*
by *auto*
thus *?thesis*
using ‹*getSATFlag x* = *UNDEF*›
using ‹(*getM x*, *getM y*) ∈ *lexLessRestricted Vbl*›
using *transLexLessRestricted*[*of Vbl*]
unfolding *trans-def*

unfolding *terminationLessState1-def*
unfolding *satFlagLessState-def*
unfolding *lexLessState1-def*
by *blast*
qed
}
thus *?thesis*
unfolding *trans-def*
by *blast*
qed

lemma *transTerminationLessState1I*:
assumes
$(x, y) \in$ *terminationLessState1 Vbl*
$(y, z) \in$ *terminationLessState1 Vbl*
shows
$(x, z) \in$ *terminationLessState1 Vbl*
using *assms*
using *transTerminationLessState1* [*of Vbl*]
unfolding *trans-def*
by *blast*

lemma *TerminationLessAfterExhaustiveUnitPropagate*:
assumes
exhaustiveUnitPropagate-dom state
InvariantUniq (*getM state*)
InvariantConsistent (*getM state*)
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*)
InvariantConflictFlagCharacterization (*getConflictFlag state*) (*getF state*) (*getM state*)
InvariantQCharacterization (*getConflictFlag state*) (*getQ state*) (*getF state*) (*getM state*)
InvariantUniqQ (*getQ state*)
InvariantVarsM (*getM state*) *F0 Vbl*
InvariantVarsQ (*getQ state*) *F0 Vbl*
InvariantVarsF (*getF state*) *F0 Vbl*
finite Vbl
getSATFlag state = UNDEF

shows
let state′ = exhaustiveUnitPropagate state in
state′ = state ∨ (*state′*, *state*) ∈ *terminationLessState1* (*vars F0* ∪ *Vbl*)
using *assms*
proof (*induct state rule*: *exhaustiveUnitPropagate-dom.induct*)
case (*step state′*)
note *ih* = *this*
show *?case*
proof (*cases* (*getConflictFlag state′*) ∨ (*getQ state′*) = [])
case *True*
with *exhaustiveUnitPropagate.simps*[*of state′*]
have *exhaustiveUnitPropagate state′* = *state′*
by *simp*
thus *?thesis*
using *True*
by (*simp add*: *Let-def*)
next
case *False*
let *?state′′* = *applyUnitPropagate state′*

have *exhaustiveUnitPropagate state′* = *exhaustiveUnitPropagate ?state′′*
using *exhaustiveUnitPropagate.simps*[*of state′*]
using *False*
by *simp*
have *InvariantWatchListsContainOnlyClausesFromF* (*getWatchList ?state′′*) (*getF ?state′′*) **and**
InvariantWatchListsUniq (*getWatchList ?state′′*) **and**
InvariantWatchListsCharacterization (*getWatchList ?state′′*) (*getWatch1 ?state′′*) (*getWatch2 ?state′′*)
InvariantWatchesEl (*getF ?state′′*) (*getWatch1 ?state′′*) (*getWatch2 ?state′′*) **and**
InvariantWatchesDiffer (*getF ?state′′*) (*getWatch1 ?state′′*) (*getWatch2 ?state′′*)
using *ih*
using *WatchInvariantsAfterAssertLiteral*[*of state′ hd* (*getQ state′*) *False*]
unfolding *applyUnitPropagate-def*
by (*auto simp add*: *Let-def*)
moreover
have *InvariantWatchCharacterization* (*getF ?state′′*) (*getWatch1 ?state′′*) (*getWatch2 ?state′′*) (*getM ?state′′*)
using *ih*
using *InvariantWatchCharacterizationAfterApplyUnitPropagate*[*of state′*]
unfolding *InvariantQCharacterization-def*
using *False*
by (*simp add*: *Let-def*)

moreover
have *InvariantQCharacterization* (*getConflictFlag ?state''*) (*getQ ?state''*) (*getF ?state''*) (*getM ?state''*)
using *ih*
using *InvariantQCharacterizationAfterApplyUnitPropagate*[*of state'*]
using *False*
by (*simp add*: *Let-def*)
moreover
have *InvariantConflictFlagCharacterization* (*getConflictFlag ?state''*) (*getF ?state''*) (*getM ?state''*)
using *ih*
using *InvariantConflictFlagCharacterizationAfterApplyUnitPropagate*[*of state'*]
using *False*
by (*simp add*: *Let-def*)
moreover
have *InvariantUniqQ* (*getQ ?state''*)
using *ih*
using *InvariantUniqQAfterApplyUnitPropagate*[*of state'*]
using *False*
by (*simp add*: *Let-def*)
moreover
have *InvariantConsistent* (*getM ?state''*)
using *ih*
using *InvariantConsistentAfterApplyUnitPropagate*[*of state'*]
using *False*
by (*simp add*: *Let-def*)
moreover
have *InvariantUniq* (*getM ?state''*)
using *ih*
using *InvariantUniqAfterApplyUnitPropagate*[*of state'*]
using *False*
by (*simp add*: *Let-def*)
moreover
have *InvariantVarsM* (*getM ?state''*) *F0 Vbl InvariantVarsQ* (*getQ ?state''*) *F0 Vbl*
using *ih*
using *False*
using *InvariantsVarsAfterApplyUnitPropagate*[*of state' F0 Vbl*]
by (*auto simp add*: *Let-def*)
moreover
have *InvariantVarsF* (*getF ?state''*) *F0 Vbl*
unfolding *applyUnitPropagate-def*
using *assertLiteralEffect*[*of state' hd* (*getQ state'*) *False*]
using *ih*
by (*simp add*: *Let-def*)
moreover
have *getSATFlag ?state''* = *UNDEF*

unfolding *applyUnitPropagate-def*
using ‹*InvariantWatchListsContainOnlyClausesFromF* (*getWatchList state'*) (*getF state'*)›
using ‹*InvariantWatchesEl* (*getF state'*) (*getWatch1 state'*) (*getWatch2 state'*)›
using ‹*getSATFlag state'* = *UNDEF*›
using *assertLiteralEffect*[*of state' hd* (*getQ state'*) *False*]
by (*simp add*: *Let-def*)
ultimately
have ∗: *exhaustiveUnitPropagate state'* = *applyUnitPropagate state'* ∨
(*exhaustiveUnitPropagate state'*, *applyUnitPropagate state'*) ∈ *terminationLessState1* (*vars F0* ∪ *Vbl*)
using *ih*
using *False*
using ‹*exhaustiveUnitPropagate state'* = *exhaustiveUnitPropagate ?state''*›
by (*simp add*: *Let-def*)
moreover
have (*?state''*, *state'*) ∈ *terminationLessState1* (*vars F0* ∪ *Vbl*)
using *applyUnitPropagateEffect*[*of state'*]
using *lexLessAppend*[*of* [(*hd* (*getQ state'*), *False*)] *getM state'*]
using *False*
using ‹*InvariantUniq* (*getM state'*)›
using ‹*InvariantConsistent* (*getM state'*)›
using ‹*InvariantVarsM* (*getM state'*) *F0 Vbl*›
using ‹*InvariantWatchesEl* (*getF state'*) (*getWatch1 state'*) (*getWatch2 state'*)›
using ‹*InvariantWatchListsContainOnlyClausesFromF* (*getWatchList state'*) (*getF state'*)›
using ‹*InvariantQCharacterization* (*getConflictFlag state'*) (*getQ state'*) (*getF state'*) (*getM state'*)›
using ‹*InvariantUniq* (*getM ?state''*)›
using ‹*InvariantConsistent* (*getM ?state''*)›
using ‹*InvariantVarsM* (*getM ?state''*) *F0 Vbl*›
using ‹*getSATFlag state'* = *UNDEF*›
using ‹*getSATFlag ?state''* = *UNDEF*›
unfolding *terminationLessState1-def*
unfolding *lexLessState1-def*
unfolding *lexLessRestricted-def*
unfolding *InvariantUniq-def*
unfolding *InvariantConsistent-def*
unfolding *InvariantVarsM-def*
by (*auto simp add*: *Let-def*)
ultimately
show *?thesis*
using *transTerminationLessState1I*[*of exhaustiveUnitPropagate state' applyUnitPropagate state' vars F0* ∪ *Vbl state'*]
by (*auto simp add*: *Let-def*)

qed
qed

lemma *InvariantsAfterSolveLoopBody*:
assumes
getSATFlag state = UNDEF
InvariantConsistent (getM state)
InvariantUniq (getM state)
InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
and
InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 state) **and**
InvariantWatchCharacterization (getF state) (getWatch1 state) (getWatch2 state) (getM state) **and**
InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state) **and**
InvariantWatchListsUniq (getWatchList state) **and**
InvariantWatchListsCharacterization (getWatchList state) (getWatch1 state) (getWatch2 state) **and**
InvariantUniqQ (getQ state) **and**
InvariantQCharacterization (getConflictFlag state) (getQ state) (getF state) (getM state) **and**
InvariantConflictFlagCharacterization (getConflictFlag state) (getF state) (getM state) **and**
InvariantNoDecisionsWhenConflict (getF state) (getM state) (currentLevel (getM state)) **and**
InvariantNoDecisionsWhenUnit (getF state) (getM state) (currentLevel (getM state)) **and**
InvariantGetReasonIsReason (getReason state) (getF state) (getM state) (set (getQ state)) **and**
InvariantEquivalentZL (getF state) (getM state) F0′ **and**
InvariantConflictClauseCharacterization (getConflictFlag state) (getConflictClause state) (getF state) (getM state) **and**
finite Vbl
vars F0′ ⊆ vars F0
vars F0 ⊆ Vbl
InvariantVarsM (getM state) F0 Vbl
InvariantVarsQ (getQ state) F0 Vbl
InvariantVarsF (getF state) F0 Vbl
shows
let state′ = solve-loop-body state Vbl in
(InvariantConsistent (getM state′) ∧
InvariantUniq (getM state′) ∧
InvariantWatchesEl (getF state′) (getWatch1 state′) (getWatch2 state′) ∧
InvariantWatchesDiffer (getF state′) (getWatch1 state′) (getWatch2 state′) ∧
InvariantWatchCharacterization (getF state′) (getWatch1 state′)

(*getWatch2 state'*) (*getM state'*) ∧
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state'*) (*getF state'*) ∧
InvariantWatchListsUniq (*getWatchList state'*) ∧
InvariantWatchListsCharacterization (*getWatchList state'*) (*getWatch1 state'*) (*getWatch2 state'*) ∧
InvariantQCharacterization (*getConflictFlag state'*) (*getQ state'*) (*getF state'*) (*getM state'*) ∧
InvariantConflictFlagCharacterization (*getConflictFlag state'*) (*getF state'*) (*getM state'*) ∧
InvariantConflictClauseCharacterization (*getConflictFlag state'*) (*getConflictClause state'*) (*getF state'*) (*getM state'*) ∧
InvariantUniqQ (*getQ state'*)) ∧
(*InvariantNoDecisionsWhenConflict* (*getF state'*) (*getM state'*) (*currentLevel* (*getM state'*)) ∧
InvariantNoDecisionsWhenUnit (*getF state'*) (*getM state'*) (*currentLevel* (*getM state'*))) ∧
InvariantEquivalentZL (*getF state'*) (*getM state'*) *F0'* ∧
InvariantGetReasonIsReason (*getReason state'*) (*getF state'*) (*getM state'*) (*set* (*getQ state'*)) ∧
InvariantVarsM (*getM state'*) *F0 Vbl* ∧
InvariantVarsQ (*getQ state'*) *F0 Vbl* ∧
InvariantVarsF (*getF state'*) *F0 Vbl* ∧
(*state'*, *state*) ∈ *terminationLessState1* (*vars F0* ∪ *Vbl*) ∧
((*getSATFlag state'* = *FALSE* ⟶ ¬ *satisfiable F0'*) ∧
(*getSATFlag state'* = *TRUE* ⟶ *satisfiable F0'*))
(**is** *let state'* = *solve-loop-body state Vbl in ?inv' state'* ∧ *?inv'' state'* ∧ -)
proof−
let *?state-up* = *exhaustiveUnitPropagate state*

have *exhaustiveUnitPropagate-dom state*
using *exhaustiveUnitPropagateTermination*[*of state F0 Vbl*]
using *assms*
by *simp*

have *?inv' ?state-up*
using *assms*
using ‹*exhaustiveUnitPropagate-dom state*›
using *InvariantsAfterExhaustiveUnitPropagate*[*of state*]
using *InvariantConflictClauseCharacterizationAfterExhaustivePropagate*[*of state*]
by (*simp add*: *Let-def*)
have *?inv'' ?state-up*
using *assms*
using ‹*exhaustiveUnitPropagate-dom state*›
using *InvariantsNoDecisionsWhenConflictNorUnitAfterExhaustivePropagate*[*of state*]
by (*simp add*: *Let-def*)

have *InvariantEquivalentZL* (*getF ?state-up*) (*getM ?state-up*) *F0′*
using *assms*
using ‹*exhaustiveUnitPropagate-dom state*›
using *InvariantEquivalentZLAfterExhaustiveUnitPropagate*[*of state*]
by (*simp add*: *Let-def*)
have *InvariantGetReasonIsReason* (*getReason ?state-up*) (*getF ?state-up*) (*getM ?state-up*) (*set* (*getQ ?state-up*))
using *assms*
using ‹*exhaustiveUnitPropagate-dom state*›
using *InvariantGetReasonIsReasonAfterExhaustiveUnitPropagate*[*of state*]
by (*simp add*: *Let-def*)
have *getSATFlag ?state-up* = *getSATFlag state*
using *exhaustiveUnitPropagatePreservedVariables*[*of state*]
using *assms*
using ‹*exhaustiveUnitPropagate-dom state*›
by (*simp add*: *Let-def*)
have *getConflictFlag ?state-up* ∨ *getQ ?state-up* = []
using *conflictFlagOrQEmptyAfterExhaustiveUnitPropagate*[*of state*]
using ‹*exhaustiveUnitPropagate-dom state*›
by (*simp add*: *Let-def*)
have *InvariantVarsM* (*getM ?state-up*) *F0 Vbl*
InvariantVarsQ (*getQ ?state-up*) *F0 Vbl*
InvariantVarsF (*getF ?state-up*) *F0 Vbl*
using *assms*
using ‹*exhaustiveUnitPropagate-dom state*›
using *InvariantsAfterExhaustiveUnitPropagate*[*of state F0 Vbl*]
by (*auto simp add*: *Let-def*)

have *?state-up* = *state* ∨ (*?state-up*, *state*) ∈ *terminationLessState1* (*vars F0* ∪ *Vbl*)
using *assms*
using *TerminationLessAfterExhaustiveUnitPropagate*[*of state*]
using ‹*exhaustiveUnitPropagate-dom state*›
by (*simp add*: *Let-def*)

show *?thesis*
proof(*cases getConflictFlag ?state-up*)
case *True*
show *?thesis*
proof (*cases currentLevel* (*getM ?state-up*) = *0*)
case *True*
hence *prefixToLevel 0* (*getM ?state-up*) = (*getM ?state-up*)
using *currentLevelZeroTrailEqualsItsPrefixToLevelZero*[*of getM ?state-up*]
by *simp*
moreover
have *formulaFalse* (*getF ?state-up*) (*elements* (*getM ?state-up*))
using ‹*getConflictFlag ?state-up*›

using ‹*?inv′ ?state-up*›
unfolding *InvariantConflictFlagCharacterization-def*
by *simp*
ultimately
have ¬ *satisfiable F0′*
using ‹*InvariantEquivalentZL* (*getF ?state-up*) (*getM ?state-up*) *F0′*›
unfolding *InvariantEquivalentZL-def*
using *soundnessForUNSAT*[*of getF ?state-up elements* (*getM ?state-up*) *F0′*]
by *simp*
moreover
let *?state′* = *?state-up* (| *getSATFlag* := *FALSE* |)
have (*?state′*, *state*) ∈ *terminationLessState1* (*vars F0* ∪ *Vbl*)
unfolding *terminationLessState1-def*
unfolding *satFlagLessState-def*
using ‹*getSATFlag state* = *UNDEF*›
by *simp*
ultimately
show *?thesis*
using ‹*?inv′ ?state-up*›
using ‹*?inv″ ?state-up*›
using ‹*InvariantEquivalentZL* (*getF ?state-up*) (*getM ?state-up*) *F0′*›
using ‹*InvariantGetReasonIsReason* (*getReason ?state-up*) (*getF ?state-up*) (*getM ?state-up*) (*set* (*getQ ?state-up*))›
using ‹*InvariantVarsM* (*getM ?state-up*) *F0 Vbl*›
using ‹*InvariantVarsQ* (*getQ ?state-up*) *F0 Vbl*›
using ‹*InvariantVarsF* (*getF ?state-up*) *F0 Vbl*›
using ‹*getConflictFlag ?state-up*›
using ‹*currentLevel* (*getM ?state-up*) = *0*›
unfolding *solve-loop-body-def*
by (*simp add*: *Let-def*)
next
case *False*
show *?thesis*
proof−

let *?state-c* = *applyConflict ?state-up*

have *?inv′ ?state-c*
?inv″ ?state-c
getConflictFlag ?state-c
InvariantEquivalentZL (*getF ?state-c*) (*getM ?state-c*) *F0′*
currentLevel (*getM ?state-c*) > *0*
using ‹*?inv′ ?state-up*› ‹*?inv″ ?state-up*›
using ‹*getConflictFlag ?state-up*›
using ‹*InvariantEquivalentZL* (*getF ?state-up*) (*getM ?state-up*) *F0′*›

using ‹*currentLevel* (*getM ?state-up*) ≠ *0*›
unfolding *applyConflict-def*
unfolding *setConflictAnalysisClause-def*
by (*auto simp add*: *Let-def findLastAssertedLiteral-def countCurrentLevelLiterals-def*)

have *InvariantCFalse* (*getConflictFlag ?state-c*) (*getM ?state-c*) (*getC ?state-c*)
InvariantCEntailed (*getConflictFlag ?state-c*) *F0′* (*getC ?state-c*)
InvariantClCharacterization (*getCl ?state-c*) (*getC ?state-c*) (*getM ?state-c*)
InvariantCnCharacterization (*getCn ?state-c*) (*getC ?state-c*) (*getM ?state-c*)
InvariantClCurrentLevel (*getCl ?state-c*) (*getM ?state-c*)
InvariantUniqC (*getC ?state-c*)
using ‹*getConflictFlag ?state-up*›
using ‹*currentLevel* (*getM ?state-up*) ≠ *0*›
using ‹*?inv′ ?state-up*›
using ‹*?inv″ ?state-up*›
using ‹*InvariantEquivalentZL* (*getF ?state-up*) (*getM ?state-up*) *F0′*›
using *InvariantsClAfterApplyConflict*[*of ?state-up*]
by (*auto simp only*: *Let-def*)

have *getSATFlag ?state-c* = *getSATFlag state*
using ‹*getSATFlag ?state-up* = *getSATFlag state*›
unfolding *applyConflict-def*
unfolding *setConflictAnalysisClause-def*
by (*simp add*: *Let-def findLastAssertedLiteral-def countCurrentLevelLiterals-def*)

have *getReason ?state-c* = *getReason ?state-up*
getF ?state-c = *getF ?state-up*
getM ?state-c = *getM ?state-up*
getQ ?state-c = *getQ ?state-up*
unfolding *applyConflict-def*
unfolding *setConflictAnalysisClause-def*
by (*auto simp add*: *Let-def findLastAssertedLiteral-def countCurrentLevelLiterals-def*)
hence *InvariantGetReasonIsReason* (*getReason ?state-c*) (*getF ?state-c*) (*getM ?state-c*) (*set* (*getQ ?state-c*))
InvariantVarsM (*getM ?state-c*) *F0 Vbl*
InvariantVarsQ (*getQ ?state-c*) *F0 Vbl*
InvariantVarsF (*getF ?state-c*) *F0 Vbl*
using ‹*InvariantGetReasonIsReason* (*getReason ?state-up*) (*getF ?state-up*) (*getM ?state-up*) (*set* (*getQ ?state-up*))›
using ‹*InvariantVarsM* (*getM ?state-up*) *F0 Vbl*›
using ‹*InvariantVarsQ* (*getQ ?state-up*) *F0 Vbl*›

using ‹*InvariantVarsF* (*getF ?state-up*) *F0 Vbl*›
by *auto*

have *getM ?state-c* = *getM state* ∨ (*?state-c*, *state*) ∈ *terminationLessState1* (*vars F0* ∪ *Vbl*)
using ‹*?state-up* = *state* ∨ (*?state-up*, *state*) ∈ *terminationLessState1* (*vars F0* ∪ *Vbl*)›
using ‹*getM ?state-c* = *getM ?state-up*›
using ‹*getSATFlag ?state-c* = *getSATFlag state*›
using ‹*InvariantUniq* (*getM state*)›
using ‹*InvariantConsistent* (*getM state*)›
using ‹*InvariantVarsM* (*getM state*) *F0 Vbl*›
using ‹*?inv′ ?state-up*›
using ‹*InvariantVarsM* (*getM ?state-up*) *F0 Vbl*›
using ‹*getSATFlag ?state-up* = *getSATFlag state*›
using ‹*getSATFlag state* = *UNDEF*›
unfolding *InvariantConsistent-def*
unfolding *InvariantUniq-def*
unfolding *InvariantVarsM-def*
unfolding *terminationLessState1-def*
unfolding *satFlagLessState-def*
unfolding *lexLessState1-def*
unfolding *lexLessRestricted-def*
by *auto*

let *?state-euip* = *applyExplainUIP ?state-c*
let *?l′* = *getCl ?state-euip*

have *applyExplainUIP-dom ?state-c*
using *ApplyExplainUIPTermination*[*of ?state-c F0′*]
using ‹*getConflictFlag ?state-c*›
using ‹*InvariantEquivalentZL* (*getF ?state-c*) (*getM ?state-c*) *F0′*›
using ‹*currentLevel* (*getM ?state-c*) > *0*›
using ‹*?inv′ ?state-c*›
using ‹*InvariantCFalse* (*getConflictFlag ?state-c*) (*getM ?state-c*) (*getC ?state-c*)›
using ‹*InvariantCEntailed* (*getConflictFlag ?state-c*) *F0′* (*getC ?state-c*)›
using ‹*InvariantClCharacterization* (*getCl ?state-c*) (*getC ?state-c*) (*getM ?state-c*)›
using ‹*InvariantCnCharacterization* (*getCn ?state-c*) (*getC ?state-c*) (*getM ?state-c*)›
using ‹*InvariantClCurrentLevel* (*getCl ?state-c*) (*getM ?state-c*)›
using ‹*InvariantGetReasonIsReason* (*getReason ?state-c*) (*getF

?state-c) (*getM ?state-c*) (*set* (*getQ ?state-c*))›
by *simp*

have *?inv′ ?state-euip ?inv′′ ?state-euip*
using ‹*?inv′ ?state-c*› ‹*?inv′′ ?state-c*›
using ‹*applyExplainUIP-dom ?state-c*›
using *ApplyExplainUIPPreservedVariables*[*of ?state-c*]
by (*auto simp add*: *Let-def*)

have *InvariantCFalse* (*getConflictFlag ?state-euip*) (*getM ?state-euip*) (*getC ?state-euip*)
InvariantCEntailed (*getConflictFlag ?state-euip*) *F0′* (*getC ?state-euip*)
InvariantClCharacterization (*getCl ?state-euip*) (*getC ?state-euip*) (*getM ?state-euip*)
InvariantCnCharacterization (*getCn ?state-euip*) (*getC ?state-euip*) (*getM ?state-euip*)
InvariantClCurrentLevel (*getCl ?state-euip*) (*getM ?state-euip*)
InvariantUniqC (*getC ?state-euip*)
using ‹*?inv′ ?state-c*›
using ‹*InvariantCFalse* (*getConflictFlag ?state-c*) (*getM ?state-c*) (*getC ?state-c*)›
using ‹*InvariantCEntailed* (*getConflictFlag ?state-c*) *F0′* (*getC ?state-c*)›
using ‹*InvariantClCharacterization* (*getCl ?state-c*) (*getC ?state-c*) (*getM ?state-c*)›
using ‹*InvariantCnCharacterization* (*getCn ?state-c*) (*getC ?state-c*) (*getM ?state-c*)›
using ‹*InvariantClCurrentLevel* (*getCl ?state-c*) (*getM ?state-c*)›
using ‹*InvariantEquivalentZL* (*getF ?state-c*) (*getM ?state-c*) *F0′*›
using ‹*InvariantUniqC* (*getC ?state-c*)›
using ‹*getConflictFlag ?state-c*›
using ‹*currentLevel* (*getM ?state-c*) > *0*›
using ‹*InvariantGetReasonIsReason* (*getReason ?state-c*) (*getF ?state-c*) (*getM ?state-c*) (*set* (*getQ ?state-c*))›
using ‹*applyExplainUIP-dom ?state-c*›
using *InvariantsClAfterExplainUIP*[*of ?state-c F0′*]
by (*auto simp only*: *Let-def*)

have *InvariantEquivalentZL* (*getF ?state-euip*) (*getM ?state-euip*) *F0′*
using ‹*InvariantEquivalentZL* (*getF ?state-c*) (*getM ?state-c*) *F0′*›
using ‹*applyExplainUIP-dom ?state-c*›
using *ApplyExplainUIPPreservedVariables*[*of ?state-c*]
by (*simp only*: *Let-def*)

have *InvariantGetReasonIsReason* (*getReason ?state-euip*) (*getF ?state-euip*) (*getM ?state-euip*) (*set* (*getQ ?state-euip*))
using ‹*InvariantGetReasonIsReason* (*getReason ?state-c*) (*getF ?state-c*) (*getM ?state-c*) (*set* (*getQ ?state-c*))›
using ‹*applyExplainUIP-dom ?state-c*›
using *ApplyExplainUIPPreservedVariables*[*of ?state-c*]
by (*simp only*: *Let-def*)

have *getConflictFlag ?state-euip*
using ‹*getConflictFlag ?state-c*›
using ‹*applyExplainUIP-dom ?state-c*›
using *ApplyExplainUIPPreservedVariables*[*of ?state-c*]
by (*simp add*: *Let-def*)

hence *getSATFlag ?state-euip* = *getSATFlag state*
using ‹*getSATFlag ?state-c* = *getSATFlag state*›
using ‹*applyExplainUIP-dom ?state-c*›
using *ApplyExplainUIPPreservedVariables*[*of ?state-c*]
by (*simp add*: *Let-def*)

have *isUIP* (*opposite* (*getCl ?state-euip*)) (*getC ?state-euip*) (*getM ?state-euip*)
using ‹*applyExplainUIP-dom ?state-c*›
using ‹*?inv' ?state-c*›
using ‹*InvariantCFalse* (*getConflictFlag ?state-c*) (*getM ?state-c*) (*getC ?state-c*)›
using ‹*InvariantCEntailed* (*getConflictFlag ?state-c*) *F0'* (*getC ?state-c*)›
using ‹*InvariantClCharacterization* (*getCl ?state-c*) (*getC ?state-c*) (*getM ?state-c*)›
using ‹*InvariantCnCharacterization* (*getCn ?state-c*) (*getC ?state-c*) (*getM ?state-c*)›
using ‹*InvariantClCurrentLevel* (*getCl ?state-c*) (*getM ?state-c*)›
using ‹*InvariantGetReasonIsReason* (*getReason ?state-c*) (*getF ?state-c*) (*getM ?state-c*) (*set* (*getQ ?state-c*))›
using ‹*InvariantEquivalentZL* (*getF ?state-c*) (*getM ?state-c*) *F0'*›
using ‹*getConflictFlag ?state-c*›
using ‹*currentLevel* (*getM ?state-c*) > *0*›
using *isUIPApplyExplainUIP*[*of ?state-c*]
by (*simp add*: *Let-def*)

have *currentLevel* (*getM ?state-euip*) > *0*
using ‹*applyExplainUIP-dom ?state-c*›
using *ApplyExplainUIPPreservedVariables*[*of ?state-c*]
using ‹*currentLevel* (*getM ?state-c*) > *0*›
by (*simp add*: *Let-def*)

have *InvariantVarsM* (*getM ?state-euip*) *F0 Vbl*

InvariantVarsQ (*getQ ?state-euip*) *F0 Vbl*
InvariantVarsF (*getF ?state-euip*) *F0 Vbl*
using ‹*InvariantVarsM* (*getM ?state-c*) *F0 Vbl*›
using ‹*InvariantVarsQ* (*getQ ?state-c*) *F0 Vbl*›
using ‹*InvariantVarsF* (*getF ?state-c*) *F0 Vbl*›
using ‹*applyExplainUIP-dom ?state-c*›
using *ApplyExplainUIPPreservedVariables*[*of ?state-c*]
by (*auto simp add*: *Let-def*)

have *getM ?state-euip* = *getM state* ∨ (*?state-euip*, *state*) ∈ *terminationLessState1* (*vars F0* ∪ *Vbl*)
using ‹*getM ?state-c* = *getM state* ∨ (*?state-c*, *state*) ∈ *terminationLessState1* (*vars F0* ∪ *Vbl*)›
using ‹*applyExplainUIP-dom ?state-c*›
using *ApplyExplainUIPPreservedVariables*[*of ?state-c*]
unfolding *terminationLessState1-def*
unfolding *satFlagLessState-def*
unfolding *lexLessState1-def*
unfolding *lexLessRestricted-def*
by (*simp add*: *Let-def*)

let *?state-l* = *applyLearn ?state-euip*
let *?l''* = *getCl ?state-l*

have $: *getM ?state-l* = *getM ?state-euip* ∧
getQ ?state-l = *getQ ?state-euip* ∧
getC ?state-l = *getC ?state-euip* ∧
getCl ?state-l = *getCl ?state-euip* ∧
getConflictFlag ?state-l = *getConflictFlag ?state-euip* ∧
getConflictClause ?state-l = *getConflictClause ?state-euip* ∧
getF ?state-l = (*if getC ?state-euip* = [*opposite ?l'*] *then*
getF ?state-euip
else
(*getF ?state-euip* @ [*getC ?state-euip*])
)
using *applyLearnPreservedVariables*[*of ?state-euip*]
by (*simp add*: *Let-def*)

have *?inv' ?state-l*
proof−
have *InvariantConflictFlagCharacterization* (*getConflictFlag ?state-l*) (*getF ?state-l*) (*getM ?state-l*)
using ‹*?inv' ?state-euip*›
using ‹*getConflictFlag ?state-euip*›
using *InvariantConflictFlagCharacterizationAfterApplyLearn*[*of ?state-euip*]
by (*simp add*: *Let-def*)

moreover
hence *InvariantQCharacterization* (*getConflictFlag ?state-l*) (*getQ ?state-l*) (*getF ?state-l*) (*getM ?state-l*)
using ‹*?inv' ?state-euip*›
using ‹*getConflictFlag ?state-euip*›
using *InvariantQCharacterizationAfterApplyLearn*[*of ?state-euip*]
by (*simp add*: *Let-def*)
moreover
have *InvariantUniqQ* (*getQ ?state-l*)
using ‹*?inv' ?state-euip*›
using *InvariantUniqQAfterApplyLearn*[*of ?state-euip*]
by (*simp add*: *Let-def*)
moreover
have *InvariantConflictClauseCharacterization* (*getConflictFlag ?state-l*) (*getConflictClause ?state-l*) (*getF ?state-l*) (*getM ?state-l*)
using ‹*?inv' ?state-euip*›
using ‹*getConflictFlag ?state-euip*›
using *InvariantConflictClauseCharacterizationAfterApplyLearn*[*of ?state-euip*]
by (*simp only*: *Let-def*)
ultimately
show *?thesis*
using ‹*?inv' ?state-euip*›
using ‹*getConflictFlag ?state-euip*›
using ‹*InvariantUniqC* (*getC ?state-euip*)›
using ‹*InvariantCFalse* (*getConflictFlag ?state-euip*) (*getM ?state-euip*) (*getC ?state-euip*)›
using ‹*InvariantClCharacterization* (*getCl ?state-euip*) (*getC ?state-euip*) (*getM ?state-euip*)›
using ‹*isUIP* (*opposite* (*getCl ?state-euip*)) (*getC ?state-euip*) (*getM ?state-euip*)›
using *WatchInvariantsAfterApplyLearn*[*of ?state-euip*]
using $
by (*auto simp only*: *Let-def*)
qed

have *InvariantNoDecisionsWhenConflict* (*getF ?state-euip*) (*getM ?state-l*) (*currentLevel* (*getM ?state-l*))
InvariantNoDecisionsWhenUnit (*getF ?state-euip*) (*getM ?state-l*) (*currentLevel* (*getM ?state-l*))
InvariantNoDecisionsWhenConflict [*getC ?state-euip*] (*getM ?state-l*) (*getBackjumpLevel ?state-l*)
InvariantNoDecisionsWhenUnit [*getC ?state-euip*] (*getM ?state-l*) (*getBackjumpLevel ?state-l*)
using *InvariantNoDecisionsWhenConflictNorUnitAfterApplyLearn*[*of ?state-euip*]
using ‹*?inv' ?state-euip*›
using ‹*?inv'' ?state-euip*›
using ‹*getConflictFlag ?state-euip*›

using ‹*InvariantUniqC* (*getC ?state-euip*)›
using ‹*InvariantCFalse* (*getConflictFlag ?state-euip*) (*getM ?state-euip*) (*getC ?state-euip*)›
using ‹*InvariantClCharacterization* (*getCl ?state-euip*) (*getC ?state-euip*) (*getM ?state-euip*)›
using ‹*InvariantClCurrentLevel* (*getCl ?state-euip*) (*getM ?state-euip*)›
using ‹*isUIP* (*opposite* (*getCl ?state-euip*)) (*getC ?state-euip*) (*getM ?state-euip*)›
using ‹*currentLevel* (*getM ?state-euip*) > *0*›
by (*auto simp only*: *Let-def*)

have *isUIP* (*opposite* (*getCl ?state-l*)) (*getC ?state-l*) (*getM ?state-l*)
using ‹*isUIP* (*opposite* (*getCl ?state-euip*)) (*getC ?state-euip*) (*getM ?state-euip*)›
using $
by *simp*

have *InvariantClCurrentLevel* (*getCl ?state-l*) (*getM ?state-l*)
using ‹*InvariantClCurrentLevel* (*getCl ?state-euip*) (*getM ?state-euip*)›
using $
by *simp*

have *InvariantCEntailed* (*getConflictFlag ?state-l*) *F0′* (*getC ?state-l*)
using ‹*InvariantCEntailed* (*getConflictFlag ?state-euip*) *F0′* (*getC ?state-euip*)›
using $
unfolding *InvariantCEntailed-def*
by *simp*

have *InvariantCFalse* (*getConflictFlag ?state-l*) (*getM ?state-l*) (*getC ?state-l*)
using ‹*InvariantCFalse* (*getConflictFlag ?state-euip*) (*getM ?state-euip*) (*getC ?state-euip*)›
using $
by *simp*

have *InvariantUniqC* (*getC ?state-l*)
using ‹*InvariantUniqC* (*getC ?state-euip*)›
using $
by *simp*

have *InvariantClCharacterization* (*getCl ?state-l*) (*getC ?state-l*) (*getM ?state-l*)
using ‹*InvariantClCharacterization* (*getCl ?state-euip*) (*getC*

?state-euip) (*getM ?state-euip*)›
unfolding *applyLearn-def*
unfolding *setWatch1-def*
unfolding *setWatch2-def*
by (*auto simp add*:*Let-def*)

have *InvariantCllCharacterization* (*getCl ?state-l*) (*getCll ?state-l*) (*getC ?state-l*) (*getM ?state-l*)
using ‹*InvariantClCharacterization* (*getCl ?state-euip*) (*getC ?state-euip*) (*getM ?state-euip*)›
‹*InvariantUniqC* (*getC ?state-euip*)›
‹*InvariantCFalse* (*getConflictFlag ?state-euip*) (*getM ?state-euip*) (*getC ?state-euip*)›
‹*getConflictFlag ?state-euip*›
‹*?inv' ?state-euip*›
using *InvariantCllCharacterizationAfterApplyLearn*[*of ?state-euip*]
by (*simp add*: *Let-def*)

have *InvariantEquivalentZL* (*getF ?state-l*) (*getM ?state-l*) *F0'*
using ‹*InvariantEquivalentZL* (*getF ?state-euip*) (*getM ?state-euip*) *F0'*›
using ‹*getConflictFlag ?state-euip*›
using *InvariantEquivalentZLAfterApplyLearn*[*of ?state-euip F0'*]
using ‹*InvariantCEntailed* (*getConflictFlag ?state-euip*) *F0'* (*getC ?state-euip*)›
by (*simp add*: *Let-def*)

have *InvariantGetReasonIsReason* (*getReason ?state-l*) (*getF ?state-l*) (*getM ?state-l*) (*set* (*getQ ?state-l*))
using ‹*InvariantGetReasonIsReason* (*getReason ?state-euip*) (*getF ?state-euip*) (*getM ?state-euip*) (*set* (*getQ ?state-euip*))›
using *InvariantGetReasonIsReasonAfterApplyLearn*[*of ?state-euip*]
by (*simp only*: *Let-def*)

have *InvariantVarsM* (*getM ?state-l*) *F0 Vbl*
InvariantVarsQ (*getQ ?state-l*) *F0 Vbl*
InvariantVarsF (*getF ?state-l*) *F0 Vbl*
using ‹*InvariantVarsM* (*getM ?state-euip*) *F0 Vbl*›
using ‹*InvariantVarsQ* (*getQ ?state-euip*) *F0 Vbl*›
using ‹*InvariantVarsF* (*getF ?state-euip*) *F0 Vbl*›
using $
using ‹*InvariantCFalse* (*getConflictFlag ?state-euip*) (*getM ?state-euip*) (*getC ?state-euip*)›
using ‹*getConflictFlag ?state-euip*›
using *InvariantVarsFAfterApplyLearn*[*of ?state-euip F0 Vbl*]
by *auto*

have *getConflictFlag ?state-l*

using ‹*getConflictFlag ?state-euip*›
using $
by *simp*

have *getSATFlag ?state-l* = *getSATFlag state*
using ‹*getSATFlag ?state-euip* = *getSATFlag state*›
unfolding *applyLearn-def*
unfolding *setWatch2-def*
unfolding *setWatch1-def*
by (*simp add*: *Let-def*)

have *currentLevel* (*getM ?state-l*) > *0*
using ‹*currentLevel* (*getM ?state-euip*) > *0*›
using $
by *simp*

have *getM ?state-l* = *getM state* ∨ (*?state-l*, *state*) ∈ *terminationLessState1* (*vars F0* ∪ *Vbl*)
proof (*cases getM ?state-euip* = *getM state*)
case *True*
thus *?thesis*
using $
by *simp*
next
case *False*
with ‹*getM ?state-euip* = *getM state* ∨ (*?state-euip*, *state*) ∈ *terminationLessState1* (*vars F0* ∪ *Vbl*)›
have (*?state-euip*, *state*) ∈ *terminationLessState1* (*vars F0* ∪ *Vbl*)
by *simp*
hence (*?state-l*, *state*) ∈ *terminationLessState1* (*vars F0* ∪ *Vbl*)
using $
using ‹*getSATFlag ?state-l* = *getSATFlag state*›
using ‹*getSATFlag ?state-euip* = *getSATFlag state*›
unfolding *terminationLessState1-def*
unfolding *satFlagLessState-def*
unfolding *lexLessState1-def*
unfolding *lexLessRestricted-def*
by (*simp add*: *Let-def*)
thus *?thesis*
by *simp*
qed

let *?state-bj* = *applyBackjump ?state-l*

have *?inv′ ?state-bj* ∧

InvariantVarsM (*getM ?state-bj*) *F0 Vbl* ∧
InvariantVarsQ (*getQ ?state-bj*) *F0 Vbl* ∧
InvariantVarsF (*getF ?state-bj*) *F0 Vbl*
proof (*cases getC ?state-l* = [*opposite ?l''*])
case *True*
thus *?thesis*
using *WatchInvariantsAfterApplyBackjump*[*of ?state-l F0'*]
using *InvariantUniqAfterApplyBackjump*[*of ?state-l F0'*]
using *InvariantConsistentAfterApplyBackjump*[*of ?state-l F0'*]
using *invariantQCharacterizationAfterApplyBackjump-1*[*of ?state-l F0'*]
using *InvariantConflictFlagCharacterizationAfterApplyBackjump-1*[*of ?state-l F0'*]
using *InvariantUniqQAfterApplyBackjump*[*of ?state-l*]
using *InvariantConflictClauseCharacterizationAfterApplyBackjump*[*of ?state-l*]
using *InvariantsVarsAfterApplyBackjump*[*of ?state-l F0' F0 Vbl*]
using ‹*?inv' ?state-l*›
using ‹*getConflictFlag ?state-l*›
using ‹*InvariantClCurrentLevel* (*getCl ?state-l*) (*getM ?state-l*)›
using ‹*InvariantUniqC* (*getC ?state-l*)›
using ‹*InvariantCFalse* (*getConflictFlag ?state-l*) (*getM ?state-l*) (*getC ?state-l*)›
using ‹*InvariantCEntailed* (*getConflictFlag ?state-l*) *F0'* (*getC ?state-l*)›
using ‹*InvariantClCharacterization* (*getCl ?state-l*) (*getC ?state-l*) (*getM ?state-l*)›
using ‹*InvariantCllCharacterization* (*getCl ?state-l*) (*getCll ?state-l*) (*getC ?state-l*) (*getM ?state-l*)›
using ‹*isUIP* (*opposite* (*getCl ?state-l*)) (*getC ?state-l*) (*getM ?state-l*)›
using ‹*currentLevel* (*getM ?state-l*) > *0*›
using ‹*InvariantNoDecisionsWhenConflict* (*getF ?state-euip*) (*getM ?state-l*) (*currentLevel* (*getM ?state-l*))›
using ‹*InvariantNoDecisionsWhenUnit* (*getF ?state-euip*) (*getM ?state-l*) (*currentLevel* (*getM ?state-l*))›
using ‹*InvariantEquivalentZL* (*getF ?state-l*) (*getM ?state-l*) *F0'*›
using ‹*InvariantVarsM* (*getM ?state-l*) *F0 Vbl*›
using ‹*InvariantVarsQ* (*getQ ?state-l*) *F0 Vbl*›
using ‹*InvariantVarsF* (*getF ?state-l*) *F0 Vbl*›
using ‹*vars F0'* ⊆ *vars F0*›
using $
by (*simp add*: *Let-def*)
next
case *False*

thus *?thesis*
using *WatchInvariantsAfterApplyBackjump*[*of ?state-l F0'*]
using *InvariantUniqAfterApplyBackjump*[*of ?state-l F0'*]
using *InvariantConsistentAfterApplyBackjump*[*of ?state-l F0'*]
using *invariantQCharacterizationAfterApplyBackjump-2*[*of ?state-l F0'*]
using *InvariantConflictFlagCharacterizationAfterApplyBackjump-2*[*of ?state-l F0'*]
using *InvariantUniqQAfterApplyBackjump*[*of ?state-l*]
using *InvariantConflictClauseCharacterizationAfterApplyBackjump*[*of ?state-l*]
using *InvariantsVarsAfterApplyBackjump*[*of ?state-l F0' F0 Vbl*]
using ‹*?inv' ?state-l*›
using ‹*getConflictFlag ?state-l*›
using ‹*InvariantClCurrentLevel* (*getCl ?state-l*) (*getM ?state-l*)›
using ‹*InvariantUniqC* (*getC ?state-l*)›
using ‹*InvariantCFalse* (*getConflictFlag ?state-l*) (*getM ?state-l*) (*getC ?state-l*)›
using ‹*InvariantCEntailed* (*getConflictFlag ?state-l*) *F0'* (*getC ?state-l*)›
using ‹*InvariantClCharacterization* (*getCl ?state-l*) (*getC ?state-l*) (*getM ?state-l*)›
using ‹*InvariantCllCharacterization* (*getCl ?state-l*) (*getCll ?state-l*) (*getC ?state-l*) (*getM ?state-l*)›
using ‹*isUIP* (*opposite* (*getCl ?state-l*)) (*getC ?state-l*) (*getM ?state-l*)›
using ‹*currentLevel* (*getM ?state-l*) > *0*›
using ‹*InvariantNoDecisionsWhenConflict* (*getF ?state-euip*) (*getM ?state-l*) (*currentLevel* (*getM ?state-l*))›
using ‹*InvariantNoDecisionsWhenUnit* (*getF ?state-euip*) (*getM ?state-l*) (*currentLevel* (*getM ?state-l*))›
using ‹*InvariantNoDecisionsWhenConflict* [*getC ?state-euip*] (*getM ?state-l*) (*getBackjumpLevel ?state-l*)›
using ‹*InvariantNoDecisionsWhenUnit* [*getC ?state-euip*] (*getM ?state-l*) (*getBackjumpLevel ?state-l*)›
using $
using ‹*InvariantEquivalentZL* (*getF ?state-l*) (*getM ?state-l*) *F0'*›
using ‹*InvariantVarsM* (*getM ?state-l*) *F0 Vbl*›
using ‹*InvariantVarsQ* (*getQ ?state-l*) *F0 Vbl*›
using ‹*InvariantVarsF* (*getF ?state-l*) *F0 Vbl*›
using ‹*vars F0'* ⊆ *vars F0*›
by (*simp add*: *Let-def*)
qed

have *?inv'' ?state-bj*

proof (*cases getC ?state-l* = [*opposite ?l′*])
case *True*
thus *?thesis*
using *InvariantsNoDecisionsWhenConflictNorUnitAfterApplyBackjump-1*[*of ?state-l F0′*]
using ‹*?inv′ ?state-l*›
using ‹*getConflictFlag ?state-l*›
using ‹*InvariantClCurrentLevel* (*getCl ?state-l*) (*getM ?state-l*)›
using ‹*InvariantUniqC* (*getC ?state-l*)›
using ‹*InvariantCFalse* (*getConflictFlag ?state-l*) (*getM ?state-l*) (*getC ?state-l*)›
using ‹*InvariantCEntailed* (*getConflictFlag ?state-l*) *F0′* (*getC ?state-l*)›
using ‹*InvariantClCharacterization* (*getCl ?state-l*) (*getC ?state-l*) (*getM ?state-l*)›
using ‹*InvariantCllCharacterization* (*getCl ?state-l*) (*getCll ?state-l*) (*getC ?state-l*) (*getM ?state-l*)›
using ‹*isUIP* (*opposite* (*getCl ?state-l*)) (*getC ?state-l*) (*getM ?state-l*)›
using ‹*currentLevel* (*getM ?state-l*) > *0*›
using ‹*InvariantNoDecisionsWhenConflict* (*getF ?state-euip*) (*getM ?state-l*) (*currentLevel* (*getM ?state-l*))›
using ‹*InvariantNoDecisionsWhenUnit* (*getF ?state-euip*) (*getM ?state-l*) (*currentLevel* (*getM ?state-l*))›
using $
by (*simp add*: *Let-def*)
next
case *False*
thus *?thesis*
using *InvariantsNoDecisionsWhenConflictNorUnitAfterApplyBackjump-2*[*of ?state-l*]
using ‹*?inv′ ?state-l*›
using ‹*getConflictFlag ?state-l*›
using ‹*InvariantClCurrentLevel* (*getCl ?state-l*) (*getM ?state-l*)›
using ‹*InvariantCFalse* (*getConflictFlag ?state-l*) (*getM ?state-l*) (*getC ?state-l*)›
using ‹*InvariantUniqC* (*getC ?state-l*)›
using ‹*InvariantCEntailed* (*getConflictFlag ?state-l*) *F0′* (*getC ?state-l*)›
using ‹*InvariantClCharacterization* (*getCl ?state-l*) (*getC ?state-l*) (*getM ?state-l*)›
using ‹*InvariantCllCharacterization* (*getCl ?state-l*) (*getCll ?state-l*) (*getC ?state-l*) (*getM ?state-l*)›
using ‹*isUIP* (*opposite* (*getCl ?state-l*)) (*getC ?state-l*) (*getM ?state-l*)›
using ‹*currentLevel* (*getM ?state-l*) > *0*›
using ‹*InvariantNoDecisionsWhenConflict* (*getF ?state-euip*)

(*getM ?state-l*) (*currentLevel* (*getM ?state-l*))›
using ‹*InvariantNoDecisionsWhenUnit* (*getF ?state-euip*) (*getM ?state-l*) (*currentLevel* (*getM ?state-l*))›
using ‹*InvariantNoDecisionsWhenConflict* [*getC ?state-euip*] (*getM ?state-l*) (*getBackjumpLevel ?state-l*)›
using ‹*InvariantNoDecisionsWhenUnit* [*getC ?state-euip*] (*getM ?state-l*) (*getBackjumpLevel ?state-l*)›
using $
by (*simp add*: *Let-def*)
qed

have *getBackjumpLevel ?state-l* > *0* ⟶ (*getF ?state-l*) ≠ [] ∧ (*last* (*getF ?state-l*) = (*getC ?state-l*))
proof (*cases getC ?state-l* = [*opposite ?l''*])
case *True*
thus *?thesis*
unfolding *getBackjumpLevel-def*
by *simp*
next
case *False*
thus *?thesis*
using $
by *simp*
qed
hence *InvariantGetReasonIsReason* (*getReason ?state-bj*) (*getF ?state-bj*) (*getM ?state-bj*) (*set* (*getQ ?state-bj*))
using ‹*InvariantGetReasonIsReason* (*getReason ?state-l*) (*getF ?state-l*) (*getM ?state-l*) (*set* (*getQ ?state-l*))›
using ‹*?inv' ?state-l*›
using ‹*getConflictFlag ?state-l*›
using ‹*isUIP* (*opposite* (*getCl ?state-l*)) (*getC ?state-l*) (*getM ?state-l*)›
using ‹*InvariantClCurrentLevel* (*getCl ?state-l*) (*getM ?state-l*)›
using ‹*InvariantCEntailed* (*getConflictFlag ?state-l*) *F0'* (*getC ?state-l*)›
using ‹*InvariantCFalse* (*getConflictFlag ?state-l*) (*getM ?state-l*) (*getC ?state-l*)›
using ‹*InvariantUniqC* (*getC ?state-l*)›
using ‹*InvariantClCharacterization* (*getCl ?state-l*) (*getC ?state-l*) (*getM ?state-l*)›
using ‹*InvariantCllCharacterization* (*getCl ?state-l*) (*getCll ?state-l*) (*getC ?state-l*) (*getM ?state-l*)›
using ‹*currentLevel* (*getM ?state-l*) > *0*›
using *InvariantGetReasonIsReasonAfterApplyBackjump*[*of ?state-l F0'*]
by (*simp only*: *Let-def*)

have *InvariantEquivalentZL* (*getF ?state-bj*) (*getM ?state-bj*) *F0'*
using ‹*InvariantEquivalentZL* (*getF ?state-l*) (*getM ?state-l*) *F0'*›
using ‹*?inv' ?state-l*›
using ‹*getConflictFlag ?state-l*›
using ‹*isUIP* (*opposite* (*getCl ?state-l*)) (*getC ?state-l*) (*getM ?state-l*)›
using ‹*InvariantClCurrentLevel* (*getCl ?state-l*) (*getM ?state-l*)›
using ‹*InvariantUniqC* (*getC ?state-l*)›
using ‹*InvariantCEntailed* (*getConflictFlag ?state-l*) *F0'* (*getC ?state-l*)›
using ‹*InvariantCFalse* (*getConflictFlag ?state-l*) (*getM ?state-l*) (*getC ?state-l*)›
using ‹*InvariantClCharacterization* (*getCl ?state-l*) (*getC ?state-l*) (*getM ?state-l*)›
using ‹*InvariantCllCharacterization* (*getCl ?state-l*) (*getCll ?state-l*) (*getC ?state-l*) (*getM ?state-l*)›
using *InvariantEquivalentZLAfterApplyBackjump*[*of ?state-l F0'*]
using ‹*currentLevel* (*getM ?state-l*) > *0*›
by (*simp only*: *Let-def*)

have *getSATFlag ?state-bj* = *getSATFlag state*
using ‹*getSATFlag ?state-l* = *getSATFlag state*›
using ‹*?inv' ?state-l*›
using *applyBackjumpPreservedVariables*[*of ?state-l*]
by (*simp only*: *Let-def*)

let *?level* = *getBackjumpLevel ?state-l*
let *?prefix* = *prefixToLevel ?level* (*getM ?state-l*)
let *?l* = *opposite* (*getCl ?state-l*)

have *isMinimalBackjumpLevel* (*getBackjumpLevel ?state-l*) (*opposite* (*getCl ?state-l*)) (*getC ?state-l*) (*getM ?state-l*)
using *isMinimalBackjumpLevelGetBackjumpLevel*[*of ?state-l*]
using ‹*?inv' ?state-l*›
using ‹*InvariantClCurrentLevel* (*getCl ?state-l*) (*getM ?state-l*)›
using ‹*InvariantCEntailed* (*getConflictFlag ?state-l*) *F0'* (*getC ?state-l*)›
using ‹*InvariantCFalse* (*getConflictFlag ?state-l*) (*getM ?state-l*) (*getC ?state-l*)›
using ‹*InvariantUniqC* (*getC ?state-l*)›
using ‹*InvariantClCharacterization* (*getCl ?state-l*) (*getC ?state-l*) (*getM ?state-l*)›
using ‹*InvariantCllCharacterization* (*getCl ?state-l*) (*getCll ?state-l*) (*getC ?state-l*) (*getM ?state-l*)›
using ‹*isUIP* (*opposite* (*getCl ?state-l*)) (*getC ?state-l*) (*getM*

?state-l)›
using ‹*getConflictFlag ?state-l*›
using ‹*currentLevel* (*getM ?state-l*) > *0*›
by (*simp add*: *Let-def*)
hence *getBackjumpLevel ?state-l* < *elementLevel* (*getCl ?state-l*) (*getM ?state-l*)
unfolding *isMinimalBackjumpLevel-def*
unfolding *isBackjumpLevel-def*
by *simp*
hence *getBackjumpLevel ?state-l* < *currentLevel* (*getM ?state-l*)
using *elementLevelLeqCurrentLevel*[*of getCl ?state-l getM ?state-l*]
by *simp*
hence (*?state-bj*, *?state-l*) ∈ *terminationLessState1* (*vars F0* ∪ *Vbl*)
using *applyBackjumpEffect*[*of ?state-l F0'*]
using ‹*?inv' ?state-l*›
using ‹*getConflictFlag ?state-l*›
using ‹*isUIP* (*opposite* (*getCl ?state-l*)) (*getC ?state-l*) (*getM ?state-l*)›
using ‹*InvariantClCurrentLevel* (*getCl ?state-l*) (*getM ?state-l*)›
using ‹*InvariantCEntailed* (*getConflictFlag ?state-l*) *F0'* (*getC ?state-l*)›
using ‹*InvariantCFalse* (*getConflictFlag ?state-l*) (*getM ?state-l*) (*getC ?state-l*)›
using ‹*InvariantUniqC* (*getC ?state-l*)›
using ‹*InvariantClCharacterization* (*getCl ?state-l*) (*getC ?state-l*) (*getM ?state-l*)›
using ‹*InvariantCllCharacterization* (*getCl ?state-l*) (*getCll ?state-l*) (*getC ?state-l*) (*getM ?state-l*)›
using ‹*currentLevel* (*getM ?state-l*) > *0*›
using *lexLessBackjump*[*of ?prefix ?level getM ?state-l ?l*]
using ‹*getSATFlag ?state-bj* = *getSATFlag state*›
using ‹*getSATFlag ?state-l* = *getSATFlag state*›
using ‹*getSATFlag state* = *UNDEF*›
using ‹*?inv' ?state-l*›
using ‹*InvariantVarsM* (*getM ?state-l*) *F0 Vbl*›
using ‹*?inv' ?state-bj* ∧ *InvariantVarsM* (*getM ?state-bj*) *F0 Vbl* ∧
InvariantVarsQ (*getQ ?state-bj*) *F0 Vbl* ∧
InvariantVarsF (*getF ?state-bj*) *F0 Vbl*›
unfolding *InvariantConsistent-def*
unfolding *InvariantUniq-def*
unfolding *InvariantVarsM-def*
unfolding *terminationLessState1-def*
unfolding *satFlagLessState-def*
unfolding *lexLessState1-def*
unfolding *lexLessRestricted-def*
by (*simp add*: *Let-def*)

hence (*?state-bj*, *state*) ∈ *terminationLessState1* (*vars F0* ∪ *Vbl*)
using ‹*getM ?state-l* = *getM state* ∨ (*?state-l*, *state*) ∈ *terminationLessState1* (*vars F0* ∪ *Vbl*)›
using ‹*getSATFlag state* = *UNDEF*›
using ‹*getSATFlag ?state-bj* = *getSATFlag state*›
using ‹*getSATFlag ?state-l* = *getSATFlag state*›
using *transTerminationLessState1I*[*of ?state-bj ?state-l vars F0* ∪ *Vbl state*]
unfolding *terminationLessState1-def*
unfolding *satFlagLessState-def*
unfolding *lexLessState1-def*
unfolding *lexLessRestricted-def*
by *auto*

show *?thesis*
using ‹*?inv′ ?state-bj* ∧ *InvariantVarsM* (*getM ?state-bj*) *F0 Vbl* ∧
InvariantVarsQ (*getQ ?state-bj*) *F0 Vbl* ∧
InvariantVarsF (*getF ?state-bj*) *F0 Vbl*›
using ‹*?inv″ ?state-bj*›
using ‹*InvariantEquivalentZL* (*getF ?state-bj*) (*getM ?state-bj*) *F0′*›
using ‹*InvariantGetReasonIsReason* (*getReason ?state-bj*) (*getF ?state-bj*) (*getM ?state-bj*) (*set* (*getQ ?state-bj*))›
using ‹*getSATFlag state* = *UNDEF*›
using ‹*getSATFlag ?state-bj* = *getSATFlag state*›
using ‹*getConflictFlag ?state-up*›
using ‹*currentLevel* (*getM ?state-up*) ≠ *0*›
using ‹(*?state-bj*, *state*) ∈ *terminationLessState1* (*vars F0* ∪ *Vbl*)›
unfolding *solve-loop-body-def*
by (*auto simp add*: *Let-def*)
qed
qed
next
case *False*
show *?thesis*
proof (*cases vars* (*elements* (*getM ?state-up*)) ⊇ *Vbl*)
case *True*
hence *satisfiable F0′*
using *soundnessForSat*[*of F0′ Vbl getF ?state-up getM ?state-up*]
using ‹*InvariantEquivalentZL* (*getF ?state-up*) (*getM ?state-up*) *F0′*›
using ‹*?inv′ ?state-up*›
using ‹*InvariantVarsF* (*getF ?state-up*) *F0 Vbl*›
using ‹¬ *getConflictFlag ?state-up*›
using ‹*vars F0* ⊆ *Vbl*›
using ‹*vars F0′* ⊆ *vars F0*›

using *True*
unfolding *InvariantConflictFlagCharacterization-def*
unfolding *satisfiable-def*
unfolding *InvariantVarsF-def*
by *blast*
moreover
let *?state′* = *?state-up* (| *getSATFlag* := *TRUE* |)
have (*?state′*, *state*) ∈ *terminationLessState1* (*vars F0* ∪ *Vbl*)
using ‹*getSATFlag state* = *UNDEF*›
unfolding *terminationLessState1-def*
unfolding *satFlagLessState-def*
by *simp*
ultimately
show *?thesis*
using ‹*vars* (*elements* (*getM ?state-up*)) ⊇ *Vbl*›
using ‹*?inv′ ?state-up*›
using ‹*?inv″ ?state-up*›
using ‹*InvariantEquivalentZL* (*getF ?state-up*) (*getM ?state-up*) *F0′*›
using ‹*InvariantGetReasonIsReason* (*getReason ?state-up*) (*getF ?state-up*) (*getM ?state-up*) (*set* (*getQ ?state-up*))›
using ‹*InvariantVarsM* (*getM ?state-up*) *F0 Vbl*›
using ‹*InvariantVarsQ* (*getQ ?state-up*) *F0 Vbl*›
using ‹*InvariantVarsF* (*getF ?state-up*) *F0 Vbl*›
using ‹¬ *getConflictFlag ?state-up*›
unfolding *solve-loop-body-def*
by (*simp add*: *Let-def*)
next
case *False*
let *?literal* = *selectLiteral ?state-up Vbl*
let *?state-d* = *applyDecide ?state-up Vbl*

have *InvariantConsistent* (*getM ?state-d*)
using *InvariantConsistentAfterApplyDecide* [*of Vbl ?state-up*]
using *False*
using ‹*?inv′ ?state-up*›
by (*simp add*: *Let-def*)
moreover
have *InvariantUniq* (*getM ?state-d*)
using *InvariantUniqAfterApplyDecide* [*of Vbl ?state-up*]
using *False*
using ‹*?inv′ ?state-up*›
by (*simp add*: *Let-def*)
moreover
have *InvariantQCharacterization* (*getConflictFlag ?state-d*) (*getQ ?state-d*) (*getF ?state-d*) (*getM ?state-d*)
using *InvariantQCharacterizationAfterApplyDecide* [*of Vbl ?state-up*]
using *False*

using ‹*?inv′ ?state-up*›
using ‹¬ *getConflictFlag ?state-up*›
using ‹*exhaustiveUnitPropagate-dom state*›
using *conflictFlagOrQEmptyAfterExhaustiveUnitPropagate*[*of state*]
by (*simp add*: *Let-def*)
moreover
have *InvariantConflictFlagCharacterization* (*getConflictFlag ?state-d*) (*getF ?state-d*) (*getM ?state-d*)
using ‹*InvariantConsistent* (*getM ?state-d*)›
using ‹*InvariantUniq* (*getM ?state-d*)›
using *InvariantConflictFlagCharacterizationAfterAssertLiteral*[*of ?state-up ?literal True*]
using ‹*?inv′ ?state-up*›
using *assertLiteralEffect*
unfolding *applyDecide-def*
by (*simp only*: *Let-def*)
moreover
have *InvariantConflictClauseCharacterization* (*getConflictFlag ?state-d*) (*getConflictClause ?state-d*) (*getF ?state-d*) (*getM ?state-d*)
using *InvariantConflictClauseCharacterizationAfterAssertLiteral*[*of ?state-up ?literal True*]
using ‹*?inv′ ?state-up*›
using *assertLiteralEffect*
unfolding *applyDecide-def*
by (*simp only*: *Let-def*)
moreover
have *InvariantNoDecisionsWhenConflict* (*getF ?state-d*) (*getM ?state-d*) (*currentLevel* (*getM ?state-d*))
InvariantNoDecisionsWhenUnit (*getF ?state-d*) (*getM ?state-d*) (*currentLevel* (*getM ?state-d*))
using ‹*exhaustiveUnitPropagate-dom state*›
using *conflictFlagOrQEmptyAfterExhaustiveUnitPropagate*[*of state*]
using ‹¬ *getConflictFlag ?state-up*›
using ‹*?inv′ ?state-up*›
using ‹*?inv′′ ?state-up*›
using *InvariantsNoDecisionsWhenConflictNorUnitAfterAssertLiteral*[*of ?state-up True ?literal*]
unfolding *applyDecide-def*
by (*auto simp add*: *Let-def*)
moreover
have *InvariantEquivalentZL* (*getF ?state-d*) (*getM ?state-d*) *F0′*
using *InvariantEquivalentZLAfterApplyDecide*[*of ?state-up F0′ Vbl*]
using ‹*?inv′ ?state-up*›
using ‹*InvariantEquivalentZL* (*getF ?state-up*) (*getM ?state-up*) *F0′*›
by (*simp add*: *Let-def*)

moreover
have *InvariantGetReasonIsReason* (*getReason ?state-d*) (*getF ?state-d*) (*getM ?state-d*) (*set* (*getQ ?state-d*))
using *InvariantGetReasonIsReasonAfterApplyDecide*[*of Vbl ?state-up*]
using ‹*?inv′ ?state-up*›
using ‹*InvariantGetReasonIsReason* (*getReason ?state-up*) (*getF ?state-up*) (*getM ?state-up*) (*set* (*getQ ?state-up*))›
using *False*
using ‹¬ *getConflictFlag ?state-up*›
using ‹*getConflictFlag ?state-up* ∨ *getQ ?state-up* = []›
by (*simp add*: *Let-def*)
moreover
have *getSATFlag ?state-d* = *getSATFlag state*
unfolding *applyDecide-def*
using ‹*getSATFlag ?state-up* = *getSATFlag state*›
using *assertLiteralEffect*[*of ?state-up selectLiteral ?state-up Vbl True*]
using ‹*?inv′ ?state-up*›
by (*simp only*: *Let-def*)
moreover
have *InvariantVarsM* (*getM ?state-d*) *F0 Vbl*
InvariantVarsF (*getF ?state-d*) *F0 Vbl*
InvariantVarsQ (*getQ ?state-d*) *F0 Vbl*
using *InvariantsVarsAfterApplyDecide*[*of Vbl ?state-up*]
using *False*
using ‹*?inv′ ?state-up*›
using ‹¬ *getConflictFlag ?state-up*›
using ‹*getConflictFlag ?state-up* ∨ *getQ ?state-up* = []›
using ‹*InvariantVarsM* (*getM ?state-up*) *F0 Vbl*›
using ‹*InvariantVarsQ* (*getQ ?state-up*) *F0 Vbl*›
using ‹*InvariantVarsF* (*getF ?state-up*) *F0 Vbl*›
by (*auto simp only*: *Let-def*)
moreover
have (*?state-d*, *?state-up*) ∈ *terminationLessState1* (*vars F0* ∪ *Vbl*)
using ‹*getSATFlag ?state-up* = *getSATFlag state*›
using *assertLiteralEffect*[*of ?state-up selectLiteral ?state-up Vbl True*]
using ‹*?inv′ ?state-up*›
using ‹*InvariantVarsM* (*getM state*) *F0 Vbl*›
using ‹*InvariantVarsM* (*getM ?state-up*) *F0 Vbl*›
using ‹*InvariantVarsM* (*getM ?state-d*) *F0 Vbl*›
using ‹*getSATFlag state* = *UNDEF*›
using ‹*?inv′ ?state-up*›
using ‹*InvariantConsistent* (*getM ?state-d*)›
using ‹*InvariantUniq* (*getM ?state-d*)›
using *lexLessAppend*[*of* [(*selectLiteral ?state-up Vbl*, *True*)]*getM ?state-up*]

unfolding *applyDecide-def*
unfolding *terminationLessState1-def*
unfolding *lexLessState1-def*
unfolding *lexLessRestricted-def*
unfolding *InvariantVarsM-def*
unfolding *InvariantUniq-def*
unfolding *InvariantConsistent-def*
by (*simp add*: *Let-def*)
hence (*?state-d*, *state*) ∈ *terminationLessState1* (*vars F0* ∪ *Vbl*)
using ‹*?state-up* = *state* ∨ (*?state-up*, *state*) ∈ *termination-LessState1* (*vars F0* ∪ *Vbl*)›
using *transTerminationLessState1I*[*of ?state-d ?state-up vars F0* ∪ *Vbl state*]
by *auto*
ultimately
show *?thesis*
using ‹*?inv′ ?state-up*›
using ‹*getSATFlag state* = *UNDEF*›
using ‹¬ *getConflictFlag ?state-up*›
using *False*
using *WatchInvariantsAfterAssertLiteral*[*of ?state-up ?literal True*]
using *InvariantWatchCharacterizationAfterAssertLiteral*[*of ?state-up ?literal True*]
using *InvariantUniqQAfterAssertLiteral*[*of ?state-up ?literal True*]
using *assertLiteralEffect*[*of ?state-up ?literal True*]
unfolding *solve-loop-body-def*
unfolding *applyDecide-def*
unfolding *selectLiteral-def*
by (*simp add*: *Let-def*)
qed
qed
qed

lemma *SolveLoopTermination*:
assumes
InvariantConsistent (*getM state*)
InvariantUniq (*getM state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
and
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2*

state) (*getM state*) **and**
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantUniqQ (*getQ state*) **and**
InvariantQCharacterization (*getConflictFlag state*) (*getQ state*) (*getF state*) (*getM state*) **and**
InvariantConflictFlagCharacterization (*getConflictFlag state*) (*getF state*) (*getM state*) **and**
InvariantNoDecisionsWhenConflict (*getF state*) (*getM state*) (*currentLevel* (*getM state*)) **and**
InvariantNoDecisionsWhenUnit (*getF state*) (*getM state*) (*currentLevel* (*getM state*)) **and**
InvariantGetReasonIsReason (*getReason state*) (*getF state*) (*getM state*) (*set* (*getQ state*)) **and**
getSATFlag state = *UNDEF* ⟶ *InvariantEquivalentZL* (*getF state*) (*getM state*) *F0′* **and**
InvariantConflictClauseCharacterization (*getConflictFlag state*) (*getConflictClause state*) (*getF state*) (*getM state*) **and**
finite Vbl
vars F0′ ⊆ *vars F0*
vars F0 ⊆ *Vbl*
InvariantVarsM (*getM state*) *F0 Vbl*
InvariantVarsQ (*getQ state*) *F0 Vbl*
InvariantVarsF (*getF state*) *F0 Vbl*
shows
solve-loop-dom state Vbl
using *assms*
proof (*induct rule*: *wf-induct*[*of terminationLessState1* (*vars F0* ∪ *Vbl*)])
case *1*
thus *?case*
using ‹*finite Vbl*›
using *finiteVarsFormula*[*of F0*]
using *wellFoundedTerminationLessState1*[*of vars F0* ∪ *Vbl*]
by *simp*
next
case (*2 state′*)
note *ih* = *this*
show *?case*
proof (*cases getSATFlag state′* = *UNDEF*)
case *False*
show *?thesis*
apply (*rule solve-loop-dom.intros*)
using *False*
by *simp*
next

case *True*
let *?state''* = *solve-loop-body state' Vbl*
have
InvariantConsistent (*getM ?state''*)
InvariantUniq (*getM ?state''*)
InvariantWatchesEl (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*) **and**
InvariantWatchesDiffer (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*) **and**
InvariantWatchCharacterization (*getF ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*) (*getM ?state''*) **and**
InvariantWatchListsContainOnlyClausesFromF (*getWatchList ?state''*) (*getF ?state''*) **and**
InvariantWatchListsUniq (*getWatchList ?state''*) **and**
InvariantWatchListsCharacterization (*getWatchList ?state''*) (*getWatch1 ?state''*) (*getWatch2 ?state''*) **and**
InvariantUniqQ (*getQ ?state''*) **and**
InvariantQCharacterization (*getConflictFlag ?state''*) (*getQ ?state''*) (*getF ?state''*) (*getM ?state''*) **and**
InvariantConflictFlagCharacterization (*getConflictFlag ?state''*) (*getF ?state''*) (*getM ?state''*) **and**
InvariantNoDecisionsWhenConflict (*getF ?state''*) (*getM ?state''*) (*currentLevel* (*getM ?state''*)) **and**
InvariantNoDecisionsWhenUnit (*getF ?state''*) (*getM ?state''*) (*currentLevel* (*getM ?state''*)) **and**
InvariantConflictClauseCharacterization (*getConflictFlag ?state''*) (*getConflictClause ?state''*) (*getF ?state''*) (*getM ?state''*)
InvariantGetReasonIsReason (*getReason ?state''*) (*getF ?state''*) (*getM ?state''*) (*set* (*getQ ?state''*))
InvariantEquivalentZL (*getF ?state''*) (*getM ?state''*) *F0'*
InvariantVarsM (*getM ?state''*) *F0 Vbl*
InvariantVarsQ (*getQ ?state''*) *F0 Vbl*
InvariantVarsF (*getF ?state''*) *F0 Vbl*
getSATFlag ?state'' = *FALSE* ⟶ ¬ *satisfiable F0'*
getSATFlag ?state'' = *TRUE* ⟶ *satisfiable F0'*
(*?state''*, *state'*) ∈ *terminationLessState1* (*vars F0* ∪ *Vbl*)
using *InvariantsAfterSolveLoopBody*[*of state' F0' Vbl F0*]
using *ih*(*2*) *ih*(*3*) *ih*(*4*) *ih*(*5*) *ih*(*6*) *ih*(*7*) *ih*(*8*) *ih*(*9*) *ih*(*10*) *ih*(*11*) *ih*(*12*) *ih*(*13*) *ih*(*14*) *ih*(*15*)
ih(*16*) *ih*(*17*) *ih*(*18*) *ih*(*19*) *ih*(*20*) *ih*(*21*) *ih*(*22*) *ih*(*23*)
using *True*
by (*auto simp only*: *Let-def*)
hence *solve-loop-dom ?state'' Vbl*
using *ih*
by *auto*
thus *?thesis*
using *solve-loop-dom.intros*[*of state' Vbl*]
using *True*
by *simp*

qed
qed

lemma *SATFlagAfterSolveLoop*:
assumes
solve-loop-dom state Vbl
InvariantConsistent (*getM state*)
InvariantUniq (*getM state*)
InvariantWatchesEl (*getF state*) (*getWatch1 state*) (*getWatch2 state*)
and
InvariantWatchesDiffer (*getF state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantWatchCharacterization (*getF state*) (*getWatch1 state*) (*getWatch2 state*) (*getM state*) **and**
InvariantWatchListsContainOnlyClausesFromF (*getWatchList state*) (*getF state*) **and**
InvariantWatchListsUniq (*getWatchList state*) **and**
InvariantWatchListsCharacterization (*getWatchList state*) (*getWatch1 state*) (*getWatch2 state*) **and**
InvariantUniqQ (*getQ state*) **and**
InvariantQCharacterization (*getConflictFlag state*) (*getQ state*) (*getF state*) (*getM state*) **and**
InvariantConflictFlagCharacterization (*getConflictFlag state*) (*getF state*) (*getM state*) **and**
InvariantNoDecisionsWhenConflict (*getF state*) (*getM state*) (*currentLevel* (*getM state*)) **and**
InvariantNoDecisionsWhenUnit (*getF state*) (*getM state*) (*currentLevel* (*getM state*)) **and**
InvariantGetReasonIsReason (*getReason state*) (*getF state*) (*getM state*) (*set* (*getQ state*)) **and**
getSATFlag state = *UNDEF* ⟶ *InvariantEquivalentZL* (*getF state*) (*getM state*) *F0′* **and**
InvariantConflictClauseCharacterization (*getConflictFlag state*) (*getConflictClause state*) (*getF state*) (*getM state*)
getSATFlag state = *FALSE* ⟶ ¬ *satisfiable F0′*
getSATFlag state = *TRUE* ⟶ *satisfiable F0′*
finite Vbl
vars F0′ ⊆ *vars F0*
vars F0 ⊆ *Vbl*
InvariantVarsM (*getM state*) *F0 Vbl*
InvariantVarsF (*getF state*) *F0 Vbl*
InvariantVarsQ (*getQ state*) *F0 Vbl*
shows
let state′ = *solve-loop state Vbl in*
(*getSATFlag state′* = *FALSE* ∧ ¬ *satisfiable F0′*) ∨ (*getSATFlag state′* = *TRUE* ∧ *satisfiable F0′*)
using *assms*
proof (*induct state Vbl rule*: *solve-loop-dom.induct*)

case (*step state′ Vbl*)
note *ih* = *this*
show *?case*
proof (*cases getSATFlag state′* = *UNDEF*)
case *False*
with *solve-loop.simps*[*of state′*]
have *solve-loop state′ Vbl* = *state′*
by *simp*
thus *?thesis*
using *False*
using *ih*(*19*) *ih*(*20*)
using *ExtendedBool.nchotomy*
by (*auto simp add*: *Let-def*)
next
case *True*
let *?state′′* = *solve-loop-body state′ Vbl*
have *solve-loop state′ Vbl* = *solve-loop ?state′′ Vbl*
using *solve-loop.simps*[*of state′*]
using *True*
by (*simp add*: *Let-def*)
moreover
have *InvariantEquivalentZL* (*getF state′*) (*getM state′*) *F0′*
using *True*
using *ih*(*17*)
by *simp*
hence
InvariantConsistent (*getM ?state′′*)
InvariantUniq (*getM ?state′′*)
InvariantWatchesEl (*getF ?state′′*) (*getWatch1 ?state′′*) (*getWatch2 ?state′′*) **and**
InvariantWatchesDiffer (*getF ?state′′*) (*getWatch1 ?state′′*) (*getWatch2 ?state′′*) **and**
InvariantWatchCharacterization (*getF ?state′′*) (*getWatch1 ?state′′*) (*getWatch2 ?state′′*) (*getM ?state′′*) **and**
InvariantWatchListsContainOnlyClausesFromF (*getWatchList ?state′′*) (*getF ?state′′*) **and**
InvariantWatchListsUniq (*getWatchList ?state′′*) **and**
InvariantWatchListsCharacterization (*getWatchList ?state′′*) (*getWatch1 ?state′′*) (*getWatch2 ?state′′*) **and**
InvariantUniqQ (*getQ ?state′′*) **and**
InvariantQCharacterization (*getConflictFlag ?state′′*) (*getQ ?state′′*) (*getF ?state′′*) (*getM ?state′′*) **and**
InvariantConflictFlagCharacterization (*getConflictFlag ?state′′*) (*getF ?state′′*) (*getM ?state′′*) **and**
InvariantNoDecisionsWhenConflict (*getF ?state′′*) (*getM ?state′′*) (*currentLevel* (*getM ?state′′*)) **and**
InvariantNoDecisionsWhenUnit (*getF ?state′′*) (*getM ?state′′*) (*currentLevel* (*getM ?state′′*)) **and**
InvariantConflictClauseCharacterization (*getConflictFlag ?state′′*)

(*getConflictClause ?state''*) (*getF ?state''*) (*getM ?state''*)
InvariantGetReasonIsReason (*getReason ?state''*) (*getF ?state''*) (*getM ?state''*) (*set* (*getQ ?state''*))
InvariantEquivalentZL (*getF ?state''*) (*getM ?state''*) *F0'*
InvariantVarsM (*getM ?state''*) *F0 Vbl*
InvariantVarsQ (*getQ ?state''*) *F0 Vbl*
InvariantVarsF (*getF ?state''*) *F0 Vbl*
getSATFlag ?state'' = *FALSE* ⟶ ¬ *satisfiable F0'*
getSATFlag ?state'' = *TRUE* ⟶ *satisfiable F0'*
using *ih*(*1*) *ih*(*3*) *ih*(*4*) *ih*(*5*) *ih*(*6*) *ih*(*7*) *ih*(*8*) *ih*(*9*) *ih*(*10*) *ih*(*11*) *ih*(*12*) *ih*(*13*) *ih*(*14*)
ih(*15*) *ih*(*16*) *ih*(*18*) *ih*(*21*) *ih*(*22*) *ih*(*23*) *ih*(*24*) *ih*(*25*) *ih*(*26*)
using *InvariantsAfterSolveLoopBody*[*of state' F0' Vbl F0*]
using *True*
by (*auto simp only*: *Let-def*)
ultimately
show *?thesis*
using *True*
using *ih*(*2*)
using *ih*(*21*)
using *ih*(*22*)
using *ih*(*23*)
by (*simp add*: *Let-def*)
qed
qed

end
theory *FunctionalImplementation*
imports *Initialization SolveLoop*
begin

8.2 Total correctness theorem

theorem *correctness*:
shows
(*solve F0* = *TRUE* ∧ *satisfiable F0*) ∨ (*solve F0* = *FALSE* ∧ ¬ *satisfiable F0*)
proof−
let *?istate* = *initialize F0 initialState*
let *?F0'* = *filter* (λ *c*. ¬ *clauseTautology c*) *F0*
have
InvariantConsistent (*getM ?istate*)
InvariantUniq (*getM ?istate*)
InvariantWatchesEl (*getF ?istate*) (*getWatch1 ?istate*) (*getWatch2 ?istate*) **and**
InvariantWatchesDiffer (*getF ?istate*) (*getWatch1 ?istate*) (*getWatch2

?istate) **and**
InvariantWatchCharacterization (*getF ?istate*) (*getWatch1 ?istate*) (*getWatch2 ?istate*) (*getM ?istate*) **and**
InvariantWatchListsContainOnlyClausesFromF (*getWatchList ?istate*) (*getF ?istate*) **and**
InvariantWatchListsUniq (*getWatchList ?istate*) **and**
InvariantWatchListsCharacterization (*getWatchList ?istate*) (*getWatch1 ?istate*) (*getWatch2 ?istate*) **and**
InvariantUniqQ (*getQ ?istate*) **and**
InvariantQCharacterization (*getConflictFlag ?istate*) (*getQ ?istate*) (*getF ?istate*) (*getM ?istate*) **and**
InvariantConflictFlagCharacterization (*getConflictFlag ?istate*) (*getF ?istate*) (*getM ?istate*) **and**
InvariantNoDecisionsWhenConflict (*getF ?istate*) (*getM ?istate*) (*currentLevel* (*getM ?istate*)) **and**
InvariantNoDecisionsWhenUnit (*getF ?istate*) (*getM ?istate*) (*currentLevel* (*getM ?istate*)) **and**
InvariantGetReasonIsReason (*getReason ?istate*) (*getF ?istate*) (*getM ?istate*) (*set* (*getQ ?istate*)) **and**
InvariantConflictClauseCharacterization (*getConflictFlag ?istate*) (*getConflictClause ?istate*) (*getF ?istate*) (*getM ?istate*)
InvariantVarsM (*getM ?istate*) *F0* (*vars F0*)
InvariantVarsQ (*getQ ?istate*) *F0* (*vars F0*)
InvariantVarsF (*getF ?istate*) *F0* (*vars F0*)
getSATFlag ?istate = *UNDEF* ⟶ *InvariantEquivalentZL* (*getF ?istate*) (*getM ?istate*) *?F0′* **and**
getSATFlag ?istate = *FALSE* ⟶ ¬ *satisfiable ?F0′*
getSATFlag ?istate = *TRUE* ⟶ *satisfiable F0*
  using *InvariantsAfterInitialization*[*of F0*]
  using *InvariantEquivalentZLAfterInitialization*[*of F0*]
  unfolding *InvariantVarsM-def*
  unfolding *InvariantVarsF-def*
  unfolding *InvariantVarsQ-def*
  by (*auto simp add*: *Let-def*)
moreover
hence *solve-loop-dom ?istate* (*vars F0*)
  using *SolveLoopTermination*[*of ?istate ?F0′ vars F0 F0*]
  using *finiteVarsFormula*[*of F0*]
  using *varsSubsetFormula*[*of ?F0′ F0*]
  by *auto*
ultimately
show *?thesis*
  using *finiteVarsFormula*[*of F0*]
  using *SATFlagAfterSolveLoop*[*of ?istate vars F0 ?F0′ F0*]
  using *satisfiableFilterTautologies*[*of F0*]
  unfolding *solve-def*
  using *varsSubsetFormula*[*of ?F0′ F0*]
  by (*auto simp add*: *Let-def*)
qed

end

References

[1] S. Krstic and A. Goel. Architecting solvers for sat modulo theories: Nelson-oppen with dpll. In *FroCos*, pages 1–27, 2007.

[2] F. Maric. Formalization and implementation of modern sat solvers. *submitted to Journal of Automated Reasoning*, 2008.

[3] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, Nov. 2006.