Converting Linear-Time Temporal Logic to Generalized Büchi Automata

Alexander Schimpf and Peter Lammich

March 17, 2025

Abstract

We formalize linear-time temporal logic (LTL) and the algorithm by Gerth et al. to convert LTL formulas to generalized Büchi automata. We also formalize some syntactic rewrite rules that can be applied to optimize the LTL formula before conversion. Moreover, we integrate the Stuttering Equivalence AFP-Entry by Stefan Merz, adapting the lemma that next-free LTL formula cannot distinguish between stuttering equivalent runs to our setting.

We use the Isabelle Refinement and Collection framework, as well as the Autoref tool, to obtain a refined version of our algorithm, from which efficiently executable code can be extracted.

Contents

1 Introduction

In LTL model checking obtaining an equivalent automaton from a linear temporal logic (LTL) formula makes up an important nontrivial part of the whole process. Gerth et al. [2] present a simple tableau-based construction, which takes an LTL formula and decomposes it according to its structure gaining the desired automaton step-by-step.

In this entry, we formalize Linear Temporal Logic (LTL), some optimizing syntactic rewrite rules on LTL formulas, and Gerth's algorithm. Using the Isabelle Refinement Framework, we extract efficient code from our formalization.

Moreover, we connect our LTL formalization to the one of Stefan Merz [3], adapting the lemma that next-free LTL formula cannot distinguish between stuttering equivalent runs to our setting.

This work is part of the CAVA project [1] to implement an executable fully verified LTL model checker.

2 LTL to GBA translation

theory *LTL-to-GBA*
imports
CAVA-Base.CAVA-Base
LTL.LTL
CAVA-Automata.Automata
begin

2.1 Statistics

```
code-printing
  code-module Gerth-Statistics ⇀ (SML) ‹
    structure Gerth-Statistics = struct
      val active = Unsynchronized.ref false
      val data = Unsynchronized.ref (0,0,0)

      fun is-active () = !active
      fun set-data num-states num-init num-acc = (
        active := true;
        data := (num-states, num-init, num-acc)
      )

      fun to-string () = let
        val (num-states, num-init, num-acc) = !data
      in
          Num states: ^ IntInf.toString (num-states) ^ \n
        ^ Num initial: ^ IntInf.toString num-init ^ \n
        ^ Num acc−classes: ^ IntInf.toString num-acc ^ \n
```

```
    end

    val - = Statistics.register-stat (Gerth LTL-to-GBA,is-active,to-string)
  end
›
code-reserved (SML) Gerth-Statistics

consts
  stat-set-data-int :: integer ⇒ integer ⇒ integer ⇒ unit

code-printing
  constant stat-set-data-int ⇀ (SML) Gerth'-Statistics.set'-data

definition stat-set-data ns ni na
  ≡ stat-set-data-int (integer-of-nat ns) (integer-of-nat ni) (integer-of-nat na)

lemma [autoref-rules]:
  (stat-set-data,stat-set-data) ∈ nat-rel → nat-rel → nat-rel → unit-rel
  by auto

abbreviation stat-set-data-nres ns ni na ≡ RETURN (stat-set-data ns ni na)

lemma discard-stat-refine[refine]:
  m1≤m2 ⟹ stat-set-data-nres ns ni na ≫ m1 ≤ m2 by simp-all
```

2.2 Preliminaries

Some very special lemmas for reasoning about the nres-monad

```
lemma SPEC-rule-nested2:
  ⟦m ≤ SPEC P; ⋀r1 r2. P (r1, r2) ⟹ g (r1, r2) ≤ SPEC P⟧
  ⟹ m ≤ SPEC (λr'. g r' ≤ SPEC P)
  by (simp add: pw-le-iff) blast

lemma SPEC-rule-param2:
  assumes f x ≤ SPEC (P x)
      and ⋀r1 r2. (P x) (r1, r2) ⟹ (P x') (r1, r2)
    shows f x ≤ SPEC (P x')
  using assms
  by (simp add: pw-le-iff)

lemma SPEC-rule-weak:
  assumes f x ≤ SPEC (Q x) and f x ≤ SPEC (P x)
      and ⋀r1 r2. ⟦(Q x) (r1, r2); (P x) (r1, r2)⟧ ⟹ (P x') (r1, r2)
    shows f x ≤ SPEC (P x')
  using assms
  by (simp add: pw-le-iff)

lemma SPEC-rule-weak-nested2: ⟦f ≤ SPEC Q; f ≤ SPEC P;
  ⋀r1 r2. ⟦Q (r1, r2); P (r1, r2)⟧ ⟹ g (r1, r2) ≤ SPEC P⟧
```

$\Longrightarrow f \leq SPEC\ (\lambda r'.\ g\ r' \leq SPEC\ P)$
by (*simp add*: *pw-le-iff*) *blast*

2.3 Creation of States

In this section, the first part of the algorithm, which creates the states of the automaton, is formalized.

type-synonym *node-name* = *nat*

type-synonym *'a frml* = *'a ltlr*

type-synonym *'a interprt* = *'a set word*

record *'a node* =
  name :: *node-name*
  incoming :: *node-name set*
  new :: *'a frml set*
  old :: *'a frml set*
  next :: *'a frml set*

context
begin

fun *new1* **where**
  new1 $(\mu\ and_r\ \psi) = \{\mu,\psi\}$
| *new1* $(\mu\ U_r\ \psi) = \{\mu\}$
| *new1* $(\mu\ R_r\ \psi) = \{\psi\}$
| *new1* $(\mu\ or_r\ \psi) = \{\mu\}$
| *new1* - = {}

fun *next1* **where**
  next1 $(X_r\ \psi) = \{\psi\}$
| *next1* $(\mu\ U_r\ \psi) = \{\mu\ U_r\ \psi\}$
| *next1* $(\mu\ R_r\ \psi) = \{\mu\ R_r\ \psi\}$
| *next1* - = {}

fun *new2* **where**
  new2 $(\mu\ U_r\ \psi) = \{\psi\}$
| *new2* $(\mu\ R_r\ \psi) = \{\mu, \psi\}$
| *new2* $(\mu\ or_r\ \psi) = \{\psi\}$
| *new2* - = {}

definition *expand-init* ≡ *0*
definition *expand-new-name* ≡ *Suc*

lemma *expand-new-name-expand-init*: *expand-init* < *expand-new-name nm*

by (*auto simp add*:*expand-init-def expand-new-name-def*)

lemma *expand-new-name-step*[*intro*]:
fixes $n :: 'a\ node$
shows $name\ n < expand\text{-}new\text{-}name\ (name\ n)$
by (*auto simp add*:*expand-new-name-def*)

lemma *expand-new-name--less-zero*[*intro*]: $0 < expand\text{-}new\text{-}name\ nm$
proof −
from *expand-new-name-expand-init* **have** $expand\text{-}init < expand\text{-}new\text{-}name\ nm$
by *auto*
then show *?thesis*
by (*metis gr0I less-zeroE*)
qed

abbreviation
$upd\text{-}incoming\text{-}f\ n \equiv (\lambda n'.$
if $(old\ n' = old\ n \wedge next\ n' = next\ n)$ *then*
$n'(\!|\ incoming := incoming\ n \cup incoming\ n'\ |\!)$
else n'
)

definition
$upd\text{-}incoming\ n\ ns \equiv ((upd\text{-}incoming\text{-}f\ n)\ `\ ns)$

lemma *upd-incoming--elem*:
assumes $nd \in upd\text{-}incoming\ n\ ns$
shows $nd \in ns$
$\vee\ (\exists nd' \in ns.\ nd = nd'(\!|\ incoming := incoming\ n \cup incoming\ nd'\ |\!) \wedge$
$old\ nd' = old\ n \wedge$
$next\ nd' = next\ n)$
proof −
obtain nd' **where** $nd' \in ns$ **and** *nd-eq*: $nd = upd\text{-}incoming\text{-}f\ n\ nd'$
using *assms* **unfolding** *upd-incoming-def* **by** *blast*
then show *?thesis* **by** *auto*
qed

lemma *upd-incoming--ident-node*:
assumes $nd \in upd\text{-}incoming\ n\ ns$ **and** $nd \in ns$
shows $incoming\ n \subseteq incoming\ nd \vee \neg\ (old\ nd = old\ n \wedge next\ nd = next\ n)$
proof (*rule ccontr*)
assume ¬ *?thesis*
then have *nsubset*: $\neg\ (incoming\ n \subseteq incoming\ nd)$ **and**
$old\text{-}next\text{-}eq$: $old\ nd = old\ n \wedge next\ nd = next\ n$
by *auto*
moreover
obtain nd' **where** $nd' \in ns$ **and** *nd-eq*: $nd = upd\text{-}incoming\text{-}f\ n\ nd'$
using *assms* **unfolding** *upd-incoming-def* **by** *blast*
{ assume $old\ nd' = old\ n \wedge next\ nd' = next\ n$

```
    with nd-eq have nd = nd'⦇incoming := incoming n ∪ incoming nd'⦈ by auto
    with nsubset have False by auto }
  moreover
  { assume ¬ (old nd' = old n ∧ next nd' = next n)
    with nd-eq old-next-eq have False by auto }
  ultimately show False by fast
qed

lemma upd-incoming--ident:
  assumes ∀ n∈ns. P n
      and ⋀n. ⟦n∈ns; P n⟧ ⟹ (⋀v. P (n⦇ incoming := v ⦈))
    shows ∀ n∈upd-incoming n ns. P n
proof
  fix nd v
  let ?f = λn'.
    if (old n' = old n ∧ next n' = next n) then
      n'⦇ incoming := incoming n ∪ incoming n' ⦈
    else n'
  assume nd∈upd-incoming n ns
  then obtain nd' where nd'∈ns and nd-eq: nd = ?f nd'
    unfolding upd-incoming-def by blast
  { assume old nd' = old n ∧ next nd' = next n
    then obtain v where nd = nd'⦇ incoming := v ⦈ using nd-eq by auto
    with assms ‹nd'∈ns› have P nd by auto }
  then show P nd using nd-eq ‹nd'∈ns› assms by auto
qed


lemma name-upd-incoming-f[simp]: name (upd-incoming-f n x) = name x
  by auto


lemma name-upd-incoming[simp]:
  name ‘ (upd-incoming n ns) = name ‘ ns (is ?lhs = ?rhs)
  unfolding upd-incoming-def
proof safe
  fix x
  assume x∈ns
  then have upd-incoming-f n x ∈ (λn'. local.upd-incoming-f n n') ‘ ns by auto
  then have name (upd-incoming-f n x) ∈ name ‘ (λn'. local.upd-incoming-f n n')
‘ ns
    by blast
  then show name x ∈ name ‘ (λn'. local.upd-incoming-f n n') ‘ ns by simp
qed auto


abbreviation expand-body
where
  expand-body ≡ (λexpand (n,ns).
```

```
    if new n = {} then (
      if (∃ n'∈ns. old n' = old n ∧ next n' = next n) then
        RETURN (name n, upd-incoming n ns)
      else
        expand (
          (|
            name=expand-new-name (name n),
            incoming={name n},
            new=next n,
            old={},
            next={}
          |),
          {n} ∪ ns)
    ) else do {
      φ ← SPEC (λx. x∈(new n));
      let n = n(| new := new n - {φ} |);
      if (∃ q. φ = prop_r(q) ∨ φ = nprop_r(q)) then
        (if (not_r φ) ∈ old n then RETURN (name n, ns)
         else expand (n(| old := {φ} ∪ old n |), ns))
      else if φ = true_r then expand (n(| old := {φ} ∪ old n |), ns)
      else if φ = false_r then RETURN (name n, ns)
      else if (∃ ν μ. (φ = ν and_r μ) ∨ (φ = X_r ν)) then
        expand (
          n(|
            new := new1 φ ∪ new n,
            old := {φ} ∪ old n,
            next := next1 φ ∪ next n
          |),
          ns)
      else do {
        (nm, nds) ← expand (
          n(|
            new := new1 φ ∪ new n,
            old := {φ} ∪ old n,
            next := next1 φ ∪ next n
          |),
          ns);
        expand (n(| name := nm, new := new2 φ ∪ new n, old := {φ} ∪ old n |),
nds)
      }
    }
  )
```

lemma *expand-body-mono*: *trimono expand-body* **by** *refine-mono*

definition *expand* :: (*'a node* × (*'a node set*)) ⇒ (*node-name* × *'a node set*) *nres*
  where *expand* ≡ *REC expand-body*

lemma *REC-rule-old*:

fixes $x::'x$
assumes M: *trimono body*
assumes $I0$: $\Phi\ x$
assumes IS: $\bigwedge f\ x.\ [\![\ \bigwedge x.\ \Phi\ x \Longrightarrow f\ x \leq M\ x;\ \Phi\ x;\ f \leq REC\ body\ ]\!]$
$\Longrightarrow$ *body* $f\ x \leq M\ x$
shows $REC\ body\ x \leq M\ x$
by (*rule REC-rule-arb*[**where** *pre*=λ-. Φ **and** M=λ-. M, *OF assms*])

lemma *expand-rec-rule*:
assumes $I0$: $\Phi\ x$
assumes IS: $\bigwedge f\ x.\ [\![\ \bigwedge x.\ f\ x \leq expand\ x;\ \bigwedge x.\ \Phi\ x \Longrightarrow f\ x \leq M\ x;\ \Phi\ x\ ]\!]$
$\Longrightarrow$ *expand-body* $f\ x \leq M\ x$
shows *expand* $x \leq M\ x$
unfolding *expand-def*
apply (*rule REC-rule-old*[**where** Φ=Φ])
apply (*rule expand-body-mono*)
apply (*rule I0*)
apply (*rule IS*[*unfolded expand-def*])
apply (*blast dest*: *le-funD*)
apply *blast*
apply *blast*
done

abbreviation
expand-assm-incoming n-ns
$\equiv (\forall\, nm{\in}incoming\ (fst\ n\text{-}ns).\ name\ (fst\ n\text{-}ns) > nm)$
$\wedge\ 0 < name\ (fst\ n\text{-}ns)$
$\wedge\ (\forall\, q{\in}snd\ n\text{-}ns.$
$name\ (fst\ n\text{-}ns) > name\ q$
$\wedge\ (\forall\, nm{\in}incoming\ q.\ name\ (fst\ n\text{-}ns) > nm))$

abbreviation
expand-rslt-incoming nm-nds
$\equiv (\forall\, q{\in}snd\ nm\text{-}nds.\ (fst\ nm\text{-}nds > name\ q \wedge (\forall\, nm'{\in}incoming\ q.\ fst\ nm\text{-}nds > nm')))$

abbreviation
expand-rslt-name n-ns nm-nds
$\equiv (name\ (fst\ n\text{-}ns) \leq fst\ nm\text{-}nds \wedge name\ `\ (snd\ n\text{-}ns) \subseteq name\ `\ (snd\ nm\text{-}nds))$
$\wedge\ name\ `\ (snd\ nm\text{-}nds)$
$= name\ `\ (snd\ n\text{-}ns) \cup name\ `\ \{nd{\in}snd\ nm\text{-}nds.\ name\ nd \geq name\ (fst\ n\text{-}ns)\}$

abbreviation
expand-name-ident nds
$\equiv (\forall\, q{\in}nds.\ \exists!\, q'{\in}nds.\ name\ q = name\ q')$

abbreviation
expand-assm-exist ξ *n-ns*
$\equiv \{\eta.\ \exists\, \mu.\ \mu\ U_r\ \eta \in old\ (fst\ n\text{-}ns) \wedge \xi \models_r \eta\} \subseteq new\ (fst\ n\text{-}ns) \cup old\ (fst\ n\text{-}ns)$

$\wedge$ ($\forall\,\psi{\in}new$ (*fst n-ns*). $\xi \models_r \psi$)
$\wedge$ ($\forall\,\psi{\in}old$ (*fst n-ns*). $\xi \models_r \psi$)
$\wedge$ ($\forall\,\psi{\in}next$ (*fst n-ns*). $\xi \models_r X_r\ \psi$)

abbreviation
expand-rslt-exist--node-prop ξ *n nd*
$\equiv$ *incoming n* $\subseteq$ *incoming nd*
$\wedge$ ($\forall\,\psi{\in}old\ nd.\ \xi \models_r \psi$) $\wedge$ ($\forall\,\psi{\in}next\ nd.\ \xi \models_r X_r\ \psi$)
$\wedge$ $\{\eta.\ \exists\,\mu.\ \mu\ U_r\ \eta \in old\ nd \wedge \xi \models_r \eta\} \subseteq old\ nd$

abbreviation
expand-rslt-exist ξ *n-ns nm-nds*
$\equiv$ ($\exists\,nd{\in}snd\ nm\text{-}nds.$ *expand-rslt-exist--node-prop* ξ (*fst n-ns*) *nd*)

abbreviation
expand-rslt-exist-eq--node n nd
$\equiv$ *name n* = *name nd*
$\wedge$ *old n* = *old nd*
$\wedge$ *next n* = *next nd*
$\wedge$ *incoming n* $\subseteq$ *incoming nd*

abbreviation
expand-rslt-exist-eq n-ns nm-nds $\equiv$
($\forall\,n{\in}snd\ n\text{-}ns.\ \exists\,nd{\in}snd\ nm\text{-}nds.$ *expand-rslt-exist-eq--node n nd*)

lemma *expand-name-propag*:
assumes *expand-assm-incoming n-ns* $\wedge$ *expand-name-ident* (*snd n-ns*) (**is** *?Q n-ns*)
shows *expand n-ns* $\leq$ *SPEC* ($\lambda r.$ *expand-rslt-incoming r*
$\wedge$ *expand-rslt-name n-ns r*
$\wedge$ *expand-name-ident* (*snd r*))
(**is** *expand* - $\leq$ *SPEC* (*?P n-ns*))
using *assms*
proof (*rule-tac expand-rec-rule*[**where** Φ=*?Q*], *simp*, *intro refine-vcg*, *goal-cases*)
case *prems*: (*1* - - *n ns*)
then have *Q*: *?Q* (*n*, *ns*) **by** *fast*
let *?nds* = *upd-incoming n ns*

from *prems* **have** $\forall\,q{\in}?nds.$ *name q* < *name n*
by (*rule-tac upd-incoming--ident*) *auto*
moreover
have $\forall\,q{\in}?nds.\ \forall\,nm'{\in}incoming\ q.$ *nm′* < *name n* (**is** $\forall\,q{\in}\text{-}.$ *?sg q*)
proof
fix *q*
assume *q-in*:*q*$\in$*?nds*
show *?sg q*
proof (*cases q*$\in$*ns*)
case *True*
with *prems* **show** *?thesis* **by** *auto*

```
  next
    case False
    with upd-incoming--elem[OF q-in]
    obtain nd' where
      nd'-def:nd'∈ns ∧ q = nd'(|incoming := incoming n ∪ incoming nd'|)
    by blast

    { fix nm'
      assume nm'∈incoming q
      moreover
      { assume nm'∈incoming n
        with Q have nm' < name n by auto }
      moreover
      { assume nm'∈incoming nd'
        with Q nd'-def have nm' < name n by auto }
      ultimately have nm' < name n using nd'-def by auto }

    then show ?thesis by fast
  qed
qed
moreover
have expand-name-ident ?nds
proof (rule upd-incoming--ident, goal-cases)
  case 1
  show ?case
  proof
    fix q
    assume q∈ns

    with Q have ∃!q'∈ns. name q = name q' by auto
    then obtain q' where q'∈ns and name q = name q'
                     and q'-all: ∀ q''∈ns. name q' = name q'' ⟶ q' = q''
      by auto
    let ?q' = upd-incoming-f n q'
    have P-a: ?q'∈?nds ∧ name q = name ?q'
      using ‹q'∈ns› ‹name q = name q'› q'-all
      unfolding upd-incoming-def by auto

    have P-all: ∀ q''∈?nds. name ?q' = name q'' ⟶ ?q' = q''
    proof(clarify)
      fix q''
      assume q''∈?nds and q''-name-eq: name ?q' = name q''
      { assume q''∉ns
        with upd-incoming--elem[OF ‹q''∈?nds›]
        obtain nd'' where
          nd''∈ns
          and q''-is: q'' = nd''(|incoming := incoming n ∪ incoming nd''|)
                      ∧ old nd'' = old n ∧ next nd'' = next n
          by auto
```

```
        then have name nd'' = name q'
          using q''-name-eq
          by (cases old q' = old n ∧ next q' = next n) auto
        with ‹nd''∈ns› q'-all have nd'' = q' by auto
        then have ?q' = q'' using q''-is by simp }
       moreover
       { assume q''∈ns
        moreover
        have name q' = name q''
          using q''-name-eq
          by (cases old q' = old n ∧ next q' = next n) auto
        moreover
        then have incoming n ⊆ incoming q''
          ⟹ incoming q'' = incoming n ∪ incoming q''
          by auto
        ultimately have ?q' = q''
          using upd-incoming--ident-node[OF ‹q''∈?nds›] q'-all
          by auto
       }
       ultimately show ?q' = q'' by fast
      qed

      show ∃!q'∈upd-incoming n ns. name q = name q'
      proof(rule ex1I[of - ?q'], goal-cases)
        case 1
        then show ?case using P-a by simp
      next
        case 2
        then show ?case
          using P-all unfolding P-a[THEN conjunct2, THEN sym]
          by blast
      qed
    qed
  qed simp
  ultimately show ?case using prems by auto
next
  case prems: (2 f x n ns)
  then have step: ⋀x. ?Q x ⟹ f x ≤ SPEC (?P x) by simp
  from prems have Q: ?Q (n, ns) by auto

  show ?case unfolding ‹x = (n, ns)›
  proof (rule-tac SPEC-rule-param2[where P = ?P], rule-tac step, goal-cases)
    case 1
    with expand-new-name-step[of n] show ?case
      using Q
      by (auto elim: order-less-trans[rotated])
  next
    case prems': (2 - nds)
    then have name ‘ ns ⊆ name ‘ nds by auto
```

```
    with expand-new-name-step[of n] show ?case
      using prems' by auto
  qed
next
  case 3
  then show ?case by auto
next
  case prems: (4 f)
  then have step: ⋀x. ?Q x ⟹ f x ≤ SPEC (?P x)
    by simp-all
  with prems show ?case
    by (rule-tac SPEC-rule-param2, rule-tac step) auto
next
  case prems: (5 f)
  then have step: ⋀x. ?Q x ⟹ f x ≤ SPEC (?P x)
    by simp-all
  from prems show ?case
    by (rule-tac SPEC-rule-param2, rule-tac step) auto
next
  case 6
  then show ?case by auto
next
  case prems: (7 f)
  then have step: ⋀x. ?Q x ⟹ f x ≤ SPEC (?P x)
    by simp-all
  from prems show ?case
    by (rule-tac SPEC-rule-param2, rule-tac step) auto
next
  case prems: (8 f x n ns ψ)
  then have goal-assms: ψ ∈ new n ∧ (∃ν μ. ψ = ν or_r μ ∨ ψ = ν U_r μ ∨ ψ =
ν R_r μ)
    by (cases ψ) auto
  from prems have Q: ?Q (n, ns) and step: ⋀x. ?Q x ⟹ f x ≤ SPEC (?P x)
    by simp-all
  show ?case
    using goal-assms Q
    unfolding case-prod-unfold ‹x = (n, ns)›
  proof (rule-tac SPEC-rule-nested2, goal-cases)
    case 1
    then show ?case
      by (rule-tac SPEC-rule-param2, rule-tac step) auto
  next
    case prems: (2 nm nds)
    then have P-x: (name n ≤ nm ∧ name ‘ ns ⊆ name ‘ nds)
      ∧ name ‘ nds = name ‘ ns ∪ name ‘ {nd∈nds. name nd ≥ name n}
      (is - ∧ ?nodes-eq nds ns (name n)) by auto
    show ?case
    proof (rule-tac SPEC-rule-param2[where P = ?P], goal-cases)
      case 1
```

```
      with prems show ?case by (rule-tac step) auto
    next
      case prems': (2 nm' nds')
      then have ∀ q∈nds'. name q < nm' ∧ (∀ nm''∈incoming q. nm'' < nm') by
auto
      moreover
      have ?nodes-eq nds ns (name n) and ?nodes-eq nds' nds nm and name n ≤
nm
        using prems' P-x by auto
      then have ?nodes-eq nds' ns (name n) by auto
      then have expand-rslt-name (n, ns) (nm', nds')
        using prems' P-x subset-trans[of name ‘ ns name ‘ nds] by auto
      ultimately show ?case using prems' by auto
    qed
  qed
qed

lemmas expand-name-propag--incoming = SPEC-rule-conjunct1[OF expand-name-propag]
lemmas expand-name-propag--name =
  SPEC-rule-conjunct1[OF SPEC-rule-conjunct2[OF expand-name-propag]]
lemmas expand-name-propag--name-ident =
  SPEC-rule-conjunct2[OF SPEC-rule-conjunct2[OF expand-name-propag]]


lemma expand-rslt-exist-eq:
  shows expand n-ns ≤ SPEC (expand-rslt-exist-eq n-ns)
    (is - ≤ SPEC (?P n-ns))
proof (rule-tac expand-rec-rule[where Φ=λ-. True], simp, intro refine-vcg, goal-cases)
  case prems: (1 f x n ns)
  let ?r = (name n, upd-incoming n ns)
  have expand-rslt-exist-eq (n, ns) ?r
    unfolding snd-conv
  proof
    fix n'
    assume n'∈ns
    { assume old n' = old n ∧ next n' = next n
      with ‹n'∈ns›
      have n'(| incoming := incoming n ∪ incoming n' |) ∈ upd-incoming n ns
        unfolding upd-incoming-def by auto
    }
    moreover
    { assume ¬ (old n' = old n ∧ next n' = next n)
      with ‹n'∈ns› have n' ∈ upd-incoming n ns
        unfolding upd-incoming-def by auto
    }
    ultimately show ∃ nd∈upd-incoming n ns. expand-rslt-exist-eq--node n' nd
      by force
  qed
  with prems show ?case by auto
```

next
case *prems*: (*2 f*)
then have *step*: $\bigwedge x.\ f\ x \leq SPEC\ (?P\ x)$ **by** *simp*
with *prems* **show** *?case* **by** (*rule-tac SPEC-rule-param2*, *rule-tac step*) *auto*
next
case *3* **then show** *?case* **by** *auto*
next
case *prems*: (*4 f*)
then have *step*: $\bigwedge x.\ f\ x \leq SPEC\ (?P\ x)$ **by** *simp*
with *prems* **show** *?case* **by** (*rule-tac SPEC-rule-param2*, *rule-tac step*) *auto*
next
case *prems*: (*5 f*)
then have *step*: $\bigwedge x.\ f\ x \leq SPEC\ (?P\ x)$ **by** *simp*
with *prems* **show** *?case* **by** (*rule-tac SPEC-rule-param2*, *rule-tac step*) *auto*
next
case *6* **then show** *?case* **by** *auto*
next
case *prems*: (*7 f*)
then have *step*: $\bigwedge x.\ f\ x \leq SPEC\ (?P\ x)$ **by** *simp*
with *prems* **show** *?case* **by** (*rule-tac SPEC-rule-param2*, *rule-tac step*) *auto*
next
case *prems*: (*8 f x n ns*)
then have *step*: $\bigwedge x.\ f\ x \leq SPEC\ (?P\ x)$ **by** *simp*

show *?case*
proof (*rule-tac SPEC-rule-nested2*, *goal-cases*)
case *1*
with *prems* **show** *?case*
by (*rule-tac SPEC-rule-param2*, *rule-tac step*) *auto*
next
case (*2 nm nds*)
with *prems* **have** *P-x*: *?P* (*n*, *ns*) (*nm*, *nds*) **by** *fast*
show *?case*
unfolding *case-prod-unfold* ‹$x = (n,\ ns)$›
proof (*rule-tac SPEC-rule-param2*[**where** $P = ?P$], *goal-cases*)
case *1*
then show *?case* **by** (*rule-tac step*)
next
case *prems'*: (*2 nm' nds'*)
{
fix *n'*
assume *n'∈ns*
with *P-x* **obtain** *nd* **where** *nd∈nds* **and** *n'-split*: *expand-rslt-exist-eq--node n' nd*
by *auto*
with *prems'* **obtain** *nd'* **where** *nd'∈nds'* **and** *expand-rslt-exist-eq--node nd nd'*
by *auto*
then have $\exists\, nd' \in nds'.$ *expand-rslt-exist-eq--node n' nd'*

```
        using n'-split subset-trans[of incoming n'] by auto
      }
      then have expand-rslt-exist-eq (n, ns) (nm', nds') by auto
      with prems show ?case by auto
    qed
  qed
qed

lemma expand-prop-exist:
  expand n-ns ≤ SPEC (λr. expand-assm-exist ξ n-ns ⟶ expand-rslt-exist ξ n-ns
r)
  (is - ≤ SPEC (?P n-ns))
proof (rule-tac expand-rec-rule[where Φ=λ-. True], simp, intro refine-vcg, goal-cases)
  case prems: (1 f x n ns)
  let ?nds = upd-incoming n ns
  let ?r = (name n, ?nds)
  { assume Q: expand-assm-exist ξ (n, ns)
    note ‹∃n'∈ns. old n' = old n ∧ next n' = next n›
    then obtain n' where n'∈ns and assm-eq: old n' = old n ∧ next n' = next n
      by auto
    let ?nd = n'(| incoming := incoming n ∪ incoming n' |)
    have ?nd ∈ ?nds using ‹n'∈ns› assm-eq unfolding upd-incoming-def by auto
    moreover
    have incoming n ⊆ incoming ?nd by auto
    moreover
    have expand-rslt-exist--node-prop ξ n ?nd using Q assm-eq ‹new n = {}›
      by simp
    ultimately have expand-rslt-exist ξ (n, ns) ?r
      unfolding fst-conv snd-conv by blast
  }
  with prems show ?case
    by auto
next
  case prems: (2 f x n ns)
  then have step: ⋀x. f x ≤ SPEC (?P x)
    and f-sup: ⋀x. f x ≤ expand x by auto
  show ?case
    unfolding ‹x = (n, ns)›
  proof (rule-tac SPEC-rule-weak[where Q = expand-rslt-exist-eq], goal-cases)
    case 1
    then show ?case
      by (rule-tac order-trans, rule-tac f-sup, rule-tac expand-rslt-exist-eq)
  next
    case 2
    then show ?case by (rule-tac step)
  next
    case prems': (3 nm nds)
    then have name ` ns ⊆ name ` nds by auto
    moreover
```

```
    { assume assm-ex: expand-assm-exist ξ (n, ns)
      with prems' obtain nd where nd∈nds and expand-rslt-exist-eq--node n nd
        by force+
      then have expand-rslt-exist--node-prop ξ n nd
       using assm-ex ‹new n = {}› by auto
      then have expand-rslt-exist ξ (n, ns) (nm, nds) using ‹nd∈nds› by auto }
    ultimately show ?case
      using expand-new-name-step[of n] prems' by auto
  qed
next
  case prems: (3 f x n ns ψ)
  { assume expand-assm-exist ξ (n, ns)
    with prems have ξ ⊨r ψ and ξ ⊨r not_r ψ
    by (metis (no-types, lifting) fstI node.select-convs(4) node.surjective node.update-convs(3))+
    then have False by simp }
  with prems show ?case by auto
next
  case prems: (4 f x n ns ψ)
  then have goal-assms: ψ ∈ new n ∧ (∃ q. ψ = prop_r(q) ∨ ψ = nprop_r(q))
    and step: ⋀x. f x ≤ SPEC (?P x) by simp-all
  show ?case
    using goal-assms unfolding ‹x = (n, ns)›
  proof (rule-tac SPEC-rule-param2, rule-tac step, goal-cases)
    case prems': (1 nm nds)
    { assume expand-assm-exist ξ (n, ns)
      with prems' have expand-rslt-exist ξ (n, ns) (nm, nds) by auto }
    then show ?case by auto
  qed
next
  case prems: (5 f x n ns ψ)
  then have goal-assms: ψ ∈ new n ∧ ψ = true_r
    and step: ⋀x. f x ≤ SPEC (?P x) by simp-all
  show ?case
    using goal-assms unfolding ‹x = (n, ns)›
  proof (rule-tac SPEC-rule-param2, rule-tac step, goal-cases)
    case prems': (1 nm nds)
    { assume expand-assm-exist ξ (n, ns)
      with prems' have expand-rslt-exist ξ (n, ns) (nm, nds) by auto }
    then show ?case by auto
  qed
next
  case prems: (6 f x n ns ψ)
  { assume expand-assm-exist ξ (n, ns)
    with prems have ξ ⊨r false_r by auto }
  with prems show ?case by auto
next
  case prems: (7 f x n ns ψ)
  then have goal-assms: ψ ∈ new n ∧ (∃ ν μ. ψ = ν and_r μ ∨ ψ = X_r ν)
    and step: ⋀x. f x ≤ SPEC (?P x) by simp-all
```

```
  show ?case
    using goal-assms unfolding ‹x = (n, ns)›
  proof (rule-tac SPEC-rule-param2, rule-tac step, goal-cases)
    case prems': (1 nm nds)
    { assume expand-assm-exist ξ (n, ns)
      with prems' have expand-rslt-exist ξ (n, ns) (nm, nds) by auto }
    then show ?case by auto
  qed
next
  case prems: (8 f x n ns ψ)
  then have goal-assms: ψ ∈ new n ∧ (∃ν μ. ψ = ν or_r μ ∨ ψ = ν U_r μ ∨ ψ =
ν R_r μ)
    by (cases ψ) auto
  from prems have step: ⋀x. f x ≤ SPEC (?P x)
    and f-sup: ⋀x. f x ≤ expand x by auto
  let ?x1 = (n(|new := new n − {ψ}, new := new1 ψ ∪ new (n(|new := new n −
{ψ}|)),
              old := {ψ} ∪ old n, next := next1 ψ ∪ next n|), ns)

  let ?new1-assm-sel = λψ. (case ψ of μ U_r η => η | μ R_r η ⇒ μ | μ or_r η ⇒ η)

  { assume new1-assm: ¬ (ξ ⊨_r (?new1-assm-sel ψ))
    then have ?case
      using goal-assms unfolding ‹x = (n, ns)›
    proof (rule-tac SPEC-rule-nested2, goal-cases)
      case prems': 1
      then show ?case
      proof (rule-tac SPEC-rule-param2, rule-tac step, goal-cases)
        case prems'': (1 nm nds)
        { assume expand-assm-exist ξ (n, ns)
          with prems'' have expand-assm-exist ξ ?x1
            unfolding fst-conv
          proof (cases ψ, goal-cases)
            case ψ: (8 μ η)
            then have ξ ⊨_r μ U_r η by fast
            then have ξ ⊨_r μ and ξ ⊨_r X_r (μ U_r η)
              using ψ ltlr-expand-Until[of ξ μ η] by auto
            with ψ show ?case by auto
          next
            case ψ: (9 μ η)
            then have ∗: ξ ⊨_r μ R_r η by fast
            with ψ have ξ ⊨_r η and ξ ⊨_r X_r (μ R_r η)
              using ltlr-expand-Release[of ξ μ η] by auto
            with ψ ∗ show ?case by auto
          qed auto
          with prems'' have expand-rslt-exist ξ (n, ns) (nm, nds) by force }
        with prems'' show ?case by auto
      qed
    next
```

```
    case prems': (2 nm nds)
    then have P-x: ?P (n, ns) (nm, nds) by fast

    show ?case
      unfolding case-prod-unfold
   proof (rule-tac SPEC-rule-weak[where P = ?P and Q = expand-rslt-exist-eq],
goal-cases)
      case 1
      then show ?case
        by (rule-tac order-trans,
          rule-tac f-sup,
          rule-tac expand-rslt-exist-eq)
    next
      case 2
      then show ?case by (rule-tac step)
    next
      case prems'': (3 nm' nds')
      { assume expand-assm-exist ξ (n, ns)
        with P-x have expand-rslt-exist ξ (n, ns) (nm, nds) by simp
        then obtain nd where nd: nd∈nds expand-rslt-exist--node-prop ξ n nd
          using goal-assms by auto
        with prems'' obtain nd' where
          nd'∈nds' and expand-rslt-exist-eq--node nd nd'
          by force
        with nd have expand-rslt-exist--node-prop ξ n nd'
          using subset-trans[of incoming n incoming nd] by auto
        then have expand-rslt-exist ξ (n,ns) (nm', nds')
          using ‹nd'∈nds'› goal-assms by auto }
      then show ?case by fast
    qed
  qed
 }
 moreover
 { assume new1-assm: ξ ⊨_r (?new1-assm-sel ψ)
  let ?x2f = λ(nm::node-name, nds::'a node set). (
      n(|new := new n − {ψ},
        name := nm,
        new := new2 ψ ∪ new (n(|new := new n − {ψ}|)),
        old := {ψ} ∪ old n|),
      nds)
  have P-x: f ?x1 ≤ SPEC (?P ?x1) by (rule step)
  moreover
  { fix r :: node-name × 'a node set
    let ?x2 = ?x2f r

    assume assm: (?P ?x1) r
    have f ?x2 ≤ SPEC (?P (n, ns))
      unfolding case-prod-unfold fst-conv
    proof (rule-tac SPEC-rule-param2, rule-tac step, goal-cases)
```

```
      case prems': (1 nm' nds')
      { assume expand-assm-exist ξ (n, ns)
        with new1-assm goal-assms have expand-assm-exist ξ ?x2
        proof (cases r, cases ψ, goal-cases)
          case prems'': (9 - - μ η)
          then have *: ξ ⊨r μ Rr η unfolding fst-conv by fast
          with ltlr-expand-Release[of ξ μ η] have ξ ⊨r η by auto
          with prems'' * show ?case by auto
        qed auto
        with prems' have expand-rslt-exist ξ ?x2 (nm', nds')
          unfolding case-prod-unfold fst-conv snd-conv by fast
        then have expand-rslt-exist ξ (n, ns) (nm', nds') by (cases r, auto) }
      then show ?case by simp
    qed
  }
  then have SPEC (?P ?x1)
    ≤ SPEC (λr. (case r of (nm, nds) =>
        f (?x2f (nm, nds))) ≤ SPEC (?P (n, ns)))
    using goal-assms by (rule-tac SPEC-rule) force
  finally have ?case unfolding case-prod-unfold ‹x = (n, ns)› by simp
 }
 ultimately show ?case by fast
qed
```

Termination proof

```
definition expand_T :: ('a node × ('a node set)) ⇒ (node-name × 'a node set) nres
  where expand_T n-ns ≡ REC_T expand-body n-ns

abbreviation old-next-pair n ≡ (old n, next n)

abbreviation old-next-limit φ ≡ Pow (subfrmlsr φ) × Pow (subfrmlsr φ)

lemma old-next-limit-finite: finite (old-next-limit φ)
  using subfrmlsr-finite by auto

definition
  expand-ord φ ≡
    inv-image (finite-psupset (old-next-limit φ) <*lex*> less-than)
          (λ(n, ns). (old-next-pair ‘ ns, size-set (new n)))

lemma expand-ord-wf[simp]: wf (expand-ord φ)
  using finite-psupset-wf[OF old-next-limit-finite]
  unfolding expand-ord-def by auto

abbreviation
  expand-inv-node φ n
  ≡ finite (new n) ∧ finite (old n) ∧ finite (next n)
  ∧ (new n) ∪ (old n) ∪ (next n) ⊆ subfrmlsr φ
```

abbreviation
expand-inv-result φ *ns* $\equiv$ *finite ns* $\wedge$ $(\forall\, n' \in ns.\ (new\ n') \cup (old\ n') \cup (next\ n') \subseteq subfrmlsr\ \varphi)$

definition
expand-inv φ *n-ns* $\equiv$ (*case n-ns of* $(n,\ ns) \Rightarrow$ *expand-inv-node* $\varphi\ n \wedge$ *expand-inv-result* $\varphi\ ns$)

lemma *new1-less-sum*:
size-set (*new1* φ) $<$ *size-set* $\{\varphi\}$
proof (*cases* φ)
case (*And-ltlr* $\nu\ \mu$)
thus *?thesis*
by (*cases* $\nu = \mu$; *simp*)
qed (*simp-all*)

lemma *new2-less-sum*:
size-set (*new2* φ) $<$ *size-set* $\{\varphi\}$
proof (*cases* φ)
case (*Release-ltlr* $\nu\ \mu$)
thus *?thesis*
by (*cases* $\nu = \mu$; *simp*)
qed (*simp-all*)

lemma *new1-finite*[*intro*]: *finite* (*new1* ψ)
by (*cases* ψ) *auto*
lemma *new1-subset-frmls*: $\varphi \in new1\ \psi \Longrightarrow \varphi \in subfrmlsr\ \psi$
by (*cases* ψ) *auto*

lemma *new2-finite*[*intro*]: *finite* (*new2* ψ)
by (*cases* ψ) *auto*
lemma *new2-subset-frmls*: $\varphi \in new2\ \psi \Longrightarrow \varphi \in subfrmlsr\ \psi$
by (*cases* ψ) *auto*

lemma *next1-finite*[*intro*]: *finite* (*next1* ψ)
by (*cases* ψ) *auto*
lemma *next1-subset-frmls*: $\varphi \in next1\ \psi \Longrightarrow \varphi \in subfrmlsr\ \psi$
by (*cases* ψ) *auto*

lemma *expand-inv-impl*[*intro*!]:
assumes *expand-inv* φ $(n,\ ns)$
and *newn*: $\psi \in new\ n$
and *old-next-pair* ‘ *ns* $\subseteq$ *old-next-pair* ‘ *ns*′
and *expand-inv-result* φ *ns*′
and $(n' = n(\!|\ new := new\ n - \{\psi\},$
$old := \{\psi\} \cup old\ n\ |\!)) \vee$
$(n' = n(\!|\ new := new1\ \psi \cup (new\ n - \{\psi\}),$
$old := \{\psi\} \cup old\ n,$
$next := next1\ \psi \cup next\ n\ |\!)) \vee$

```
        (n' = n(| name := nm,
                  new := new2 ψ ∪ (new n − {ψ}),
                  old := {ψ} ∪ old n |))
        (is ?case1 ∨ ?case2 ∨ ?case3)
  shows ((n', ns'), (n, ns))∈expand-ord φ ∧ expand-inv φ (n', ns')
        (is ?concl1 ∧ ?concl2)
proof
  from assms consider ?case1 | ?case2 | ?case3 by blast
  then show ?concl1
  proof cases
    case n'def: 1
    with assms show ?thesis
      unfolding expand-ord-def expand-inv-def finite-psupset-def
      apply (cases old-next-pair ‘ ns ⊂ old-next-pair ‘ ns')
      apply simp-all
      apply auto [1]
       apply (metis (no-types, lifting) add-Suc diff-Suc-less psubsetI sum.remove
sum-diff1-nat zero-less-Suc)
      done
  next
    case n'def: 2
    have ψinnew: ψ ∈ new n and fin-new: finite (new n)
      using assms unfolding expand-inv-def by auto
    then have size-set (new n − {ψ}) = size-set (new n) − size-set {ψ}
      using size-set-diff by fastforce
    moreover
    from fin-new sum-Un-nat[OF new1-finite -, of new n − {ψ} size ψ]
    have size-set (new n') ≤ size-set (new1 ψ) + size-set (new n − {ψ})
      unfolding n'def by (simp add: new1-finite sum-Un-nat)
    moreover
    have sum-leq: size-set (new n) ≥ size-set {ψ}
      using ψinnew sum-mono2[OF fin-new, of {ψ}]
      by blast
    ultimately
    have size-set (new n') < size-set (new n)
      using new1-less-sum[of ψ] by auto
    with assms show ?thesis
      unfolding expand-ord-def finite-psupset-def by auto
  next
    case n'def: 3
    have ψinnew: ψ ∈ new n and fin-new: finite (new n)
      using assms unfolding expand-inv-def by auto
    from ψinnew sum-diff1-nat[of size new n ψ]
    have size-set (new n − {ψ}) = size-set (new n) − size-set {ψ}
      using size-set-diff[of new n {ψ}] by fastforce
    moreover
    from fin-new sum-Un-nat[OF new2-finite -, of new n − {ψ} size ψ]
    have size-set (new n') ≤ size-set (new2 ψ) + size-set (new n − {ψ})
      unfolding n'def by (simp add: new2-finite sum-Un-nat)
```

```
    moreover
    have sum-leq:size-set (new n) ≥  size-set {ψ}
      using ψinnew sum-mono2[OF fin-new, of {ψ}] by blast
    ultimately
    have size-set (new n') < size-set (new n)
      using new2-less-sum[of ψ] sum-leq by auto
    with assms show ?thesis
      unfolding expand-ord-def finite-psupset-def by auto
  qed
next
  have new1 ψ ⊆ subfrmlsr φ
    and new2 ψ ⊆ subfrmlsr φ
    and next1 ψ ⊆ subfrmlsr φ
    using assms subfrmlsr-subset[OF new1-subset-frmls[of - ψ]]
      subfrmlsr-subset[of ψ φ, OF rev-subsetD[of - new n]]
      subfrmlsr-subset[OF new2-subset-frmls[of - ψ]]
      subfrmlsr-subset[OF next1-subset-frmls[of - ψ]]
    unfolding expand-inv-def

    apply −
    apply (clarsimp split: prod.splits)
    apply (metis in-mono new1-subset-frmls)
    apply (clarsimp split: prod.splits)
    apply (metis new2-subset-frmls rev-subsetD)
    apply (clarsimp split: prod.splits)
    apply (metis in-mono next1-subset-frmls)
    done
  with assms show ?concl2
    unfolding expand-inv-def
    by auto
qed

lemma expand-term-prop-help:
  assumes ((n', ns'), (n, ns))∈expand-ord φ ∧ expand-inv φ (n', ns')
    and assm-rule: ⟦expand-inv φ (n', ns'); ((n', ns'), n, ns) ∈ expand-ord φ⟧
      ⟹ f (n', ns') ≤ SPEC P
  shows f (n', ns') ≤ SPEC P
  using assms by (rule-tac assm-rule) auto

lemma expand-inv-upd-incoming:
  assumes expand-inv φ (n, ns)
  shows expand-inv-result φ (upd-incoming n ns)
  using assms unfolding expand-inv-def upd-incoming-def by force


lemma upd-incoming-eq-old-next-pair: old-next-pair ‘ ns = old-next-pair ‘ (upd-incoming
n ns)
  (is ?A = ?B)
proof
```

```
  show ?A ⊆ ?B
  proof
    fix x
    let ?f = λn'. n'(|incoming := incoming n ∪ incoming n'|)
    assume x ∈ ?A
    then obtain n' where n' ∈ ns and xeq: x = (old n', next n')
      by auto
    have x ∈ (old-next-pair ‘ (λn'. n'(|incoming := incoming n ∪ incoming n'|))
            ‘ (ns ∩ {n' ∈ ns. old n' = old n ∧ next n' = next n}))
        ∪ (old-next-pair ‘ (ns ∩ {n' ∈ ns. old n' ≠ old n ∨ next n' ≠ next n}))
    proof (cases old n' = old n ∧ next n' = next n)
      case True
      with ‹n' ∈ ns› 
      have ?f n' ∈ ?f ‘ (ns ∩ {n'∈ns. old n' = old n ∧ next n' = next n})  (is - ∈
?C)
        by auto
      then have old-next-pair (?f n') ∈ old-next-pair ‘ ?C
        by (rule-tac imageI) auto
      with xeq have x∈old-next-pair ‘ ?C by auto
      then show ?thesis by blast
    next
      case False
      with ‹n' ∈ ns› xeq
      have x ∈ old-next-pair ‘ (ns ∩ {n'∈ns. old n' ≠ old n ∨ next n' ≠ next n})
        by auto
      then show ?thesis by blast
    qed
    then show x ∈ ?B
      using ‹x ∈ ?A› unfolding upd-incoming-def by auto
  qed
  show ?B ⊆ ?A
    unfolding upd-incoming-def by (force intro:imageI)
qed

lemma expand-term-prop:
  expand-inv φ n-ns ⟹
    expand_T n-ns ≤ SPEC (λ(-, nds). old-next-pair ‘ snd n-ns⊆old-next-pair ‘nds
    ∧ expand-inv-result φ nds)
  (is - ⟹ - ≤ SPEC (?P n-ns))
  unfolding expand_T-def
   apply (rule-tac RECT-rule[where pre=λ(n, ns). expand-inv φ (n, ns) and
V=expand-ord φ])
  apply (refine-mono)
  apply simp
  apply simp
proof (intro refine-vcg, goal-cases)
  case prems: (1 - - n ns)
  have old-next-pair ‘ ns ⊆ old-next-pair ‘ (upd-incoming n ns)
    by (rule equalityD1[OF upd-incoming-eq-old-next-pair])
```

```
  with prems show ?case
    using expand-inv-upd-incoming[of φ n ns] by auto
next
  case prems: (2 expand x n ns)
  let ?n' = ⦇
    name = expand-new-name (name n),
    incoming = {name n},
    new = next n,
    old = {},
    next = {}⦈
  let ?ns' = {n} ∪ ns
  from prems have SPEC-sub:SPEC (?P (?n', ?ns')) ≤ SPEC (?P x)
    by (rule-tac SPEC-rule) auto
  from prems have old-next-pair n ∉ old-next-pair ‘ ns
    by auto
  then have old-next-pair ‘ ns ⊂ old-next-pair ‘ (insert n ns)
    by auto
  moreover from prems have expand-inv φ (n, ns)
    by simp
  ultimately have ((?n', ?ns'), (n, ns)) ∈ expand-ord φ
    by (auto simp add: expand-ord-def finite-psupset-def expand-inv-def)
  moreover from prems have expand-inv φ (?n', ?ns')
    unfolding expand-inv-def by auto
  ultimately have expand (?n', ?ns') ≤ SPEC (?P (?n', ?ns'))
    using prems by fast
  with SPEC-sub show ?case
    by (rule-tac order-trans) fast+
next
  case 3
  then show ?case by (auto simp add:expand-inv-def)
next
  case 4
  then show ?case
    apply (rule-tac expand-term-prop-help[OF expand-inv-impl])
    apply (simp add: expand-inv-def)+
    apply force
    done
next
  case 5
  then show ?case
    apply (rule-tac expand-term-prop-help[OF expand-inv-impl])
    apply (simp add: expand-inv-def)+
    apply force
    done
next
  case 6
  then show ?case by (simp add: expand-inv-def)
next
  case 7
```

```
  then show ?case
    apply (rule-tac expand-term-prop-help[OF expand-inv-impl])
    apply (simp add: expand-inv-def)+
    apply force
    done
next
  case prems: (8 f x a b xa)
  let ?n' = a(|
    new := new1 xa ∪ (new a − {xa}),
    old := insert xa (old a),
    next := next1 xa ∪ next a|)
  let ?n'' = λnm. a(|
    name := nm,
    new := new2 xa ∪ (new a − {xa}),
    old := insert xa (old a)|)
  have step:((?n', b), (a, b)) ∈ expand-ord φ ∧ expand-inv φ (?n', b)
    using prems by (rule-tac expand-inv-impl) (auto simp add: expand-inv-def)
  with prems have assm1: f (?n', b) ≤ SPEC (?P (a, b))
    by auto
  moreover
  {
    fix nm::node-name and nds::'a node set
    assume assm1: old-next-pair ‘ b ⊆ old-next-pair ‘ nds
      and assm2: expand-inv-result φ nds
    with prems step have ((?n'' nm, nds), (a, b)) ∈ expand-ord φ ∧ expand-inv φ
(?n'' nm, nds)
      by (rule-tac expand-inv-impl) auto
    with prems have f (?n'' nm, nds) ≤ SPEC (?P (?n'' nm, nds))
      by auto
    moreover
    have SPEC (?P (?n'' nm, nds)) ≤ SPEC (?P (a, b))
      using assm2 subset-trans[OF assm1] by auto
    ultimately have f (?n'' nm, nds) ≤ SPEC (?P (a, b))
      by (rule order-trans)
  }
  then have assm2: SPEC (?P (a, b))
    ≤ SPEC (λr. (case r of (nm, nds) ⇒ f (?n'' nm, nds)) ≤ SPEC (?P (a, b)))
    by (rule-tac SPEC-rule) auto
  from prems order-trans[OF assm1 assm2] show ?case
    by auto
qed

lemma expand-eq-expand_T:
  assumes inv: expand-inv φ n-ns
  shows expand_T n-ns = expand n-ns
  unfolding expand_T-def expand-def
  apply (rule RECT-eq-REC)
  unfolding expand_T-def[symmetric]
  using expand-term-prop[OF inv] apply auto
```

done

lemma *expand-nofail*:
assumes *inv*: *expand-inv* φ *n-ns*
shows *nofail* (*expand*$_T$ *n-ns*)
using *expand-term-prop*[*OF inv*] **by** (*simp add*: *pw-le-iff*)

lemma [*intro*!]: *expand-inv* φ (
(|
name = *expand-new-name expand-init*,
incoming = {*expand-init*},
new = {φ},
old = {},
next = {} |),
{})
by (*auto simp add*: *expand-inv-def*)

definition *create-graph* :: *'a frml* ⇒ *'a node set nres*
where
create-graph φ ≡
do {
(-, *nds*) ← *expand* (
(|
name = *expand-new-name expand-init*,
incoming = {*expand-init*},
new = {φ},
old = {},
next = {}
|)::*'a node*,
{}::*'a node set*);
RETURN nds
}

definition *create-graph*$_T$ :: *'a frml* ⇒ *'a node set nres*
where
create-graph$_T$ φ ≡ *do* {
(-, *nds*) ← *expand*$_T$ (
(|
name = *expand-new-name expand-init*,
incoming = {*expand-init*},
new = {φ},
old = {},
next = {}
|)::*'a node*,
{}::*'a node set*);
RETURN nds
}

lemma *create-graph-eq-create-graph$_T$*: *create-graph* φ = *create-graph$_T$* φ
unfolding *create-graph$_T$-def create-graph-def*
unfolding *eq-iff*
proof (*standard*, *goal-cases*)
case *1*
then show *?case*
by *refine-mono* (*unfold expand-def expand$_T$-def*, *rule REC-le-RECT*)
next
case *2*
then show *?case*
by (*refine-mono*, *rule expand-eq-expand$_T$*[*unfolded eq-iff*, *THEN conjunct1*])
auto
qed

lemma *create-graph-finite*: *create-graph* $\varphi \leq$ *SPEC finite*
unfolding *create-graph-def expand-def*
apply (*intro refine-vcg*)
apply (*rule-tac order-trans*)
apply (*rule-tac REC-le-RECT*)
apply (*fold expand$_T$-def*)
apply (*rule-tac order-trans*[*OF expand-term-prop*])
apply *auto*[*1*]
apply (*rule-tac SPEC-rule*)
apply *auto*
done

lemma *create-graph-nofail*: *nofail* (*create-graph* φ)
by (*rule-tac pwD1*[*OF create-graph-finite*]) *auto*

abbreviation
create-graph-rslt-exist ξ *nds*
$\equiv \exists\, nd{\in}nds.$
expand-init$\in$*incoming nd*
$\wedge\ (\forall\, \psi{\in}old\ nd.\ \xi \models_r \psi) \wedge (\forall\, \psi{\in}next\ nd.\ \xi \models_r X_r\ \psi)$
$\wedge\ \{\eta.\ \exists\, \mu.\ \mu\ U_r\ \eta \in old\ nd \wedge \xi \models_r \eta\} \subseteq old\ nd$

lemma *L4-7*:
assumes $\xi \models_r \varphi$
shows *create-graph* $\varphi \leq$ *SPEC* (*create-graph-rslt-exist* ξ)
using *assms* **unfolding** *create-graph-def*
by (*intro refine-vcg*, *rule-tac order-trans*, *rule-tac expand-prop-exist*) (
auto simp add:*expand-new-name-expand-init*)

lemma *expand-incoming-name-exist*:
assumes *name* (*fst n-ns*) > *expand-init*
$\wedge\ (\forall\, nm{\in}incoming\ (fst\ n\text{-}ns).\ nm \neq expand\text{-}init \longrightarrow nm{\in}name\ `\ (snd\ n\text{-}ns))$

```
    ∧ expand-assm-incoming n-ns ∧ expand-name-ident (snd n-ns) (is ?Q n-ns)
  and ∀ nd∈snd n-ns.
    name nd > expand-init
    ∧ (∀ nm∈incoming nd. nm ≠ expand-init ⟶ nm∈name ‘ (snd n-ns))
    (is ?P (snd n-ns))
  shows expand n-ns ≤ SPEC (λnm-nds. ?P (snd nm-nds))
  using assms
  apply (rule-tac expand-rec-rule[where Φ=λn-ns. ?Q n-ns ∧ ?P (snd n-ns)])
  apply simp
  apply (intro refine-vcg)
proof goal-cases
  case (1 f x n ns)
  then show ?case
  proof (simp, clarify, goal-cases)
    case prems: (1 - - nd)
    { assume nd∈ns
      with prems have ?case by auto }
    moreover
    { assume nd∉ns
      with upd-incoming--elem[OF ‹nd∈upd-incoming n ns›]
      obtain nd' where nd'∈ns and nd = nd'(|incoming :=
        incoming n ∪ incoming nd'|) ∧
        old nd' = old n ∧
        next nd' = next n by auto
      with prems have ?case by auto }
    ultimately show ?case by fast
  qed
next
  case (2 f x n ns)
  then have step: ⋀x. ?Q x ∧ ?P (snd x) ⟹ f x ≤ SPEC (λx. ?P (snd x))
    and QP: ?Q (n, ns) ∧ ?P ns
    and f-sup: ⋀x. f x ≤ expand x by auto
  show ?case
    unfolding ‹x = (n, ns)›
    using QP expand-new-name-expand-init
  proof (rule-tac step, goal-cases)
    case prems: 1
    then have name-less: name n < expand-new-name (name n)
      by auto
    moreover
    from prems name-less have ∀ nm∈incoming n. nm < expand-new-name (name
n)
      by auto
    moreover
    from prems name-less have ∗∗: ∀ q∈ns. name q < expand-new-name (name n)
∧
      (∀ nm∈incoming q. nm < expand-new-name (name n))
      by force
    moreover
```

```
    from QP have ∃!q'. (q' = n ∨ q' ∈ ns) ∧ name n = name q'
      by force
    moreover
    have ∀ q∈ns. ∃!q'.
      (q' = n ∨ q' ∈ ns) ∧
      name q = name q'  using QP by auto
    ultimately show ?case using prems by simp
  qed
next
  case 3
  then show ?case by simp
next
  case 4
  then show ?case by simp
next
  case 5
  then show ?case by simp
next
  case 6
  then show ?case by simp
next
  case 7
  then show ?case by simp
next
  case prems: (8 f x n ns ψ)
  then have goal-assms: ψ ∈ new n ∧ (∃ν μ. ψ = ν or_r μ ∨ ψ = ν U_r μ ∨ ψ =
ν R_r μ)
    by (cases ψ) auto
  with prems have QP: ?Q (n, ns) ∧ ?P ns
    and step: ⋀x. ?Q x ∧ ?P (snd x) ⟹ f x ≤ SPEC (λx'. ?P (snd x'))
    and f-sup: ⋀x. f x ≤ expand x by auto
  let ?x = (n(|new := new n − {ψ}, new := new1 ψ ∪ new (n(|new := new n −
{ψ}|)),
        old := {ψ} ∪ old (n(|new := new n − {ψ}|)),
        next := next1 ψ ∪ next (n(|new := new n − {ψ}|)|), ns)
  let ?props = λx r. expand-rslt-incoming r
    ∧ expand-rslt-name x r
    ∧ expand-name-ident (snd r)

  show ?case
    using goal-assms QP unfolding case-prod-unfold ‹x = (n, ns)›
  proof (rule-tac SPEC-rule-weak-nested2[where Q = ?props ?x], goal-cases)
    case 1
    then show ?case
      by (rule-tac order-trans, rule-tac f-sup, rule-tac expand-name-propag) simp
  next
    case 2
    then show ?case
      by (rule-tac SPEC-rule-param2[where P = λx r. ?P (snd r)], rule-tac step)
```

```
      auto
  next
    case (3 nm nds)
    then show ?case
    proof (rule-tac SPEC-rule-weak[where P = λx r. ?P (snd r)
        and Q = λx r. expand-rslt-exist-eq x r ∧ ?props x r], goal-cases)
      case 1
      then show ?case
        by (rule-tac SPEC-rule-conjI,
              rule-tac order-trans,
              rule-tac f-sup,
              rule-tac expand-rslt-exist-eq,
              rule-tac order-trans,
              rule-tac f-sup,
              rule-tac expand-name-propag) force
    next
      case 2
      then show ?case
        by (rule-tac SPEC-rule-param2[where P = λx r. ?P (snd r)],
          rule-tac step) force
    next
      case (3 nm' nds')
      then show ?case
        by simp
    qed
  qed
qed

lemma create-graph--incoming-name-exist:
  create-graph φ ≤ SPEC (λnds. ∀ nd∈nds. expand-init < name nd ∧ (∀ nm∈incoming
nd. nm ≠ expand-init ⟶ nm∈name ‘ nds))
  unfolding create-graph-def
  by (intro refine-vcg,
    rule-tac order-trans,
    rule-tac expand-incoming-name-exist) (
    auto simp add:expand-new-name-expand-init)


abbreviation
  expand-rslt-all--ex-equiv ξ nd nds ≡
   (∃ nd'∈nds.
    name nd ∈ incoming nd'
    ∧ (∀ ψ∈old nd'. suffix 1 ξ ⊨r ψ) ∧ (∀ ψ∈next nd'. suffix 1 ξ ⊨r X_r ψ)
    ∧ {η. ∃ μ. μ U_r η ∈ old nd' ∧ suffix 1 ξ ⊨r η} ⊆ old nd')

abbreviation
  expand-rslt-all ξ n-ns nm-nds ≡
   (∀ nd∈snd nm-nds. name nd ∉ name ‘ (snd n-ns) ∧
        (∀ ψ∈old nd. ξ ⊨r ψ) ∧ (∀ ψ∈next nd. ξ ⊨r X_r ψ)
```

$\longrightarrow$ *expand-rslt-all--ex-equiv* ξ *nd* (*snd nm-nds*))

lemma *expand-prop-all*:
assumes *expand-assm-incoming n-ns* $\wedge$ *expand-name-ident* (*snd n-ns*) (**is** *?Q n-ns*)
shows *expand n-ns* $\leq$ *SPEC* (*expand-rslt-all* ξ *n-ns*)
(**is** - $\leq$ *SPEC* (*?P n-ns*))
using *assms*
apply (*rule-tac expand-rec-rule*[**where** Φ=*?Q*])
apply *simp*
apply (*intro refine-vcg*)
proof *goal-cases*
case *1*
then show *?case* **by** (*simp*, *rule-tac upd-incoming--ident*) *simp-all*
next
case (*2 f x n ns*)
then have *step*: $\bigwedge x$. *?Q x* $\Longrightarrow$ *f x* $\leq$ *SPEC* (*?P x*)
and *Q*: *?Q* (*n*, *ns*)
and *f-sup*: $\bigwedge x$. *f x* $\leq$ *expand x* **by** *auto*
let *?x* = (⦇*name* = *expand-new-name* (*name n*),
incoming = {*name n*}, *new* = *next n*, *old* = {}, *next* = {}⦈, {*n*} $\cup$ *ns*)
from *Q* **have** *name-le*: *name n* < *expand-new-name* (*name n*) **by** *auto*
show *?case*
unfolding ‹*x* = (*n*, *ns*)›
proof (*rule-tac SPEC-rule-weak*[**where**
Q = λ*p r*.
(*expand-assm-exist* (*suffix 1* ξ) *?x* $\longrightarrow$ *expand-rslt-exist* (*suffix 1* ξ) *?x r*)
$\wedge$ *expand-rslt-exist-eq p r* $\wedge$ (*expand-name-ident* (*snd r*))], *goal-cases*)
case *1*
then show *?case*
proof (*rule-tac SPEC-rule-conjI*,
rule-tac order-trans,
rule-tac f-sup,
rule-tac expand-prop-exist,
rule-tac SPEC-rule-conjI,
rule-tac order-trans,
rule-tac f-sup,
rule-tac expand-rslt-exist-eq,
rule-tac order-trans,
rule-tac f-sup,
rule-tac expand-name-propag--name-ident,
goal-cases)
case *1*
then show *?case* **using** *Q name-le* **by** *force*
qed
next
case *2*
then show *?case* **using** *Q name-le* **by** (*rule-tac step*) *force*
next

```
    case prems: (3 nm nds)
    then obtain n' where n'∈nds
      and eq-node: expand-rslt-exist-eq--node n n' by auto
    with prems have ex1-name: ∃!q∈nds. name n = name q by auto
    then have nds-eq: nds = {n'} ∪ {x ∈ nds. name n ≠ name x}
      using eq-node ‹n'∈nds› by blast
    have name-notin: name n ∉ name ‘ ns using Q by auto
    from prems have P-x: expand-rslt-all ξ ?x (nm, nds) by fast
    show ?case
      unfolding snd-conv
    proof clarify
      fix nd
      assume nd ∈ nds and name-img: name nd ∉ name ‘ ns
        and nd-old-equiv: ∀ψ∈old nd. ξ ⊨r ψ
        and nd-next-equiv: ∀ψ∈next nd. ξ ⊨r X_r ψ
      show expand-rslt-all--ex-equiv ξ nd nds
      proof (cases name nd = name n)
        case True
        with nds-eq eq-node ‹nd∈nds› have nd = n' by auto
        with prems(1)[THEN conjunct1, simplified]
          nd-old-equiv nd-next-equiv eq-node
        show ?thesis by simp
      next
        case False
        with name-img ‹nd ∈ nds› nd-old-equiv nd-next-equiv P-x
        show ?thesis by simp
      qed
    qed
  qed
next
  case 3
  then show ?case by auto
next
  case prems: (4 f)
  then have step: ⋀x. ?Q x ⟹ f x ≤ SPEC (?P x) by simp
  from prems show ?case by (rule-tac SPEC-rule-param2, rule-tac step) auto
next
  case prems: (5 f)
  then have step: ⋀x. ?Q x ⟹ f x ≤ SPEC (?P x) by simp
  from prems show ?case by (rule-tac SPEC-rule-param2, rule-tac step) auto
next
  case 6
  then show ?case by auto
next
  case prems: (7 f)
  then have step: ⋀x. ?Q x ⟹ f x ≤ SPEC (?P x) by simp
  from prems show ?case by (rule-tac SPEC-rule-param2, rule-tac step) auto
next
  case prems: (8 f x n ns ψ)
```

```
  then have goal-assms: ψ ∈ new n ∧ (∃ν μ. ψ = ν or_r μ ∨ ψ = ν U_r μ ∨ ψ =
ν R_r μ)
    by (cases ψ) auto
  from prems have Q: ?Q (n, ns)
    and step: ⋀x. ?Q x ⟹ f x ≤ SPEC (?P x)
    and f-sup: ⋀x. f x ≤ expand x by auto
  let ?x = (n(|new := new n − {ψ}, new := new1 ψ ∪ new (n(|new := new n −
{ψ}|)),
          old := {ψ} ∪ old (n(|new := new n − {ψ}|)),
                       next := next1 ψ ∪ next (n(|new := new n − {ψ}|)
        |), ns)
  let ?props = λx r. expand-rslt-incoming r
    ∧ expand-rslt-name x r
    ∧ expand-name-ident (snd r)
  show ?case
    using goal-assms Q
    unfolding case-prod-unfold ‹x = (n, ns)›
  proof (rule-tac SPEC-rule-weak-nested2[where Q = ?props ?x], goal-cases)
    case 1
    then show ?case
      by (rule-tac order-trans, rule-tac f-sup, rule-tac expand-name-propag) simp
  next
    case 2
    then show ?case
      by (rule-tac SPEC-rule-param2[where P = ?P], rule-tac step) auto
  next
    case prems: (3 nm nds)
    then show ?case
    proof (rule-tac SPEC-rule-weak[where
        P = ?P and
        Q = λx r. expand-rslt-exist-eq x r ∧ ?props x r], goal-cases)
      case 1
      then show ?case
        by (rule-tac SPEC-rule-conjI,
              rule-tac order-trans,
              rule-tac f-sup,
              rule-tac expand-rslt-exist-eq,
              rule-tac order-trans,
              rule-tac f-sup,
              rule-tac expand-name-propag) auto
    next
      case 2
      then show ?case
        by (rule-tac SPEC-rule-param2[where P = ?P], rule-tac step) auto
    next
      case prems': (3 nm' nds')
      then have P-x: ?P (n, ns) (nm, nds)
        and P-x': ?P (n, nds) (nm', nds') by simp-all
      show ?case
```

```
      unfolding snd-conv
    proof clarify
      fix nd
      assume nd∈nds'
        and name-nd-notin: name nd ∉ name ‘ ns
        and old-equiv: ∀ψ∈old nd. ξ ⊨r ψ
        and next-equiv: ∀ψ∈next nd. ξ ⊨r Xr ψ
      show expand-rslt-all--ex-equiv ξ nd nds'
      proof (cases name nd ∈ name ‘ nds)
        case True
        then obtain n' where n' ∈ nds and name nd = name n' by auto
        with prems' obtain nd' where nd'∈nds'
          and nd'-eq: expand-rslt-exist-eq--node n' nd'
          by auto
        moreover from prems' have ∀q∈nds'. ∃!q'∈nds'. name q = name q'
          by simp
        ultimately have nd' = nd
          using ‹name nd = name n'› ‹nd ∈ nds'› by auto
        with nd'-eq have n'-eq: expand-rslt-exist-eq--node n' nd
          by simp
       then have name n'∉name ‘ ns and ∀ψ∈old n'. ξ ⊨r ψ and ∀ψ∈next n'.
ξ ⊨r Xr ψ
          using name-nd-notin old-equiv next-equiv ‹n' ∈ nds›
          by auto
        then have expand-rslt-all--ex-equiv ξ n' nds  (is ∃nd'∈nds. ?sthm n' nd')
          using P-x ‹n' ∈ nds› unfolding snd-conv by blast
        then obtain sucnd where sucnd: sucnd∈nds and sthm: ?sthm n' sucnd
          by blast
        moreover
        from prems' sucnd sthm obtain sucnd' where sucnd'∈nds'
            and sucnd'-eq: expand-rslt-exist-eq--node sucnd sucnd'
          by auto
        ultimately have ?sthm n' sucnd' by auto
        then show ?thesis
          using ‹sucnd' ∈ nds'›
          unfolding ‹name nd = name n'› by blast
      next
        case False
        with ‹nd ∈ nds'› P-x' old-equiv next-equiv
        show ?thesis unfolding snd-conv by blast
      qed
    qed
  qed
qed

abbreviation
  create-graph-rslt-all ξ nds
  ≡ ∀q∈nds. (∀ψ∈old q. ξ ⊨r ψ) ∧ (∀ψ∈next q. ξ ⊨r Xr ψ)
```

$\longrightarrow$ ($\exists\, q'{\in}nds.$ *name* $q \in$ *incoming* q'
$\wedge$ ($\forall\, \psi{\in}old\ q'.$ *suffix 1* $\xi \models_r \psi$)
$\wedge$ ($\forall\, \psi{\in}next\ q'.$ *suffix 1* $\xi \models_r X_r\ \psi$)
$\wedge$ $\{\eta.\ \exists \mu.\ \mu\ U_r\ \eta \in old\ q' \wedge$ *suffix 1* $\xi \models_r \eta\} \subseteq old\ q'$)

lemma *L4-5*: *create-graph* $\varphi \leq$ *SPEC* (*create-graph-rslt-all* ξ)
unfolding *create-graph-def*
apply (*refine-vcg expand-prop-all*)
apply (*auto simp add*:*expand-new-name-expand-init*)
done

2.4 Creation of GBA

This section formalizes the second part of the algorithm, that creates the actual generalized Büchi automata from the set of nodes.

definition *create-gba-from-nodes* :: *'a frml* $\Rightarrow$ *'a node set* $\Rightarrow$ (*'a node*, *'a set*) *gba-rec*
where *create-gba-from-nodes* φ *qs* $\equiv$ (|
g-V = *qs*,
g-E = $\{(q,\ q').\ q{\in}qs \wedge q'{\in}qs \wedge$ *name* $q{\in}$*incoming* $q'\}$,
g-V0 = $\{q{\in}qs.$ *expand-init*$\in$*incoming* $q\}$,
gbg-F = $\{\{q{\in}qs.\ \mu\ U_r\ \eta{\in}old\ q \longrightarrow \eta{\in}old\ q\} | \mu\ \eta.\ \mu\ U_r\ \eta \in subfrmlsr\ \varphi\}$,
gba-L = $\lambda q\ l.\ q{\in}qs \wedge \{p.\ prop_r(p){\in}old\ q\}{\subseteq}l \wedge \{p.\ nprop_r(p){\in}old\ q\} \cap l = \{\}$
|)

end

locale *create-gba-from-nodes-precond* =
fixes φ :: *'a ltlr*
fixes *qs* :: *'a node set*
assumes *res*: *inres* (*create-graph* φ) *qs*
begin

lemma *finite-qs*[*simp*, *intro*!]: *finite qs*
using *res create-graph-finite* **by** (*auto simp add*: *pw-le-iff*)

lemma *create-gba-from-nodes--invar*: *gba* (*create-gba-from-nodes* φ *qs*)
using [[*simproc finite-Collect*]]
apply *unfold-locales*
apply (*auto*
intro!: *finite-vimageI subfrmlsr-finite injI*
simp: *create-gba-from-nodes-def*)
done

sublocale *gba*: *gba create-gba-from-nodes* φ *qs*
by (*rule create-gba-from-nodes--invar*)

lemma *create-gba-from-nodes--fin*: *finite* (*g-V* (*create-gba-from-nodes* φ *qs*))
unfolding *create-gba-from-nodes-def* **by** *auto*

lemma *create-gba-from-nodes--ipath*:
ipath gba.E r $\longleftrightarrow$ ($\forall$ *i. r i* $\in$*qs* $\wedge$ *name* (*r i*)$\in$*incoming* (*r* (*Suc i*)))
unfolding *create-gba-from-nodes-def ipath-def*
by *auto*

lemma *create-gba-from-nodes--is-run*:
gba.is-run r $\longleftrightarrow$ *expand-init* $\in$ *incoming* (*r 0*)
$\wedge$ ($\forall$ *i. r i*$\in$*qs* $\wedge$ *name* (*r i*)$\in$*incoming* (*r* (*Suc i*)))
unfolding *gba.is-run-def*
apply (*simp add*: *create-gba-from-nodes--ipath*)
apply (*auto simp*: *create-gba-from-nodes-def*)
done

context
begin

abbreviation
auto-run-j j ξ *q* $\equiv$
($\forall$ $\psi$$\in$*old q. suffix j* ξ $\models_r$ ψ) $\wedge$ ($\forall$ $\psi$$\in$*next q. suffix j* ξ $\models_r$ X_r ψ) $\wedge$
{η. $\exists$ μ. μ U_r η $\in$ *old q* $\wedge$ *suffix j* ξ $\models_r$ η} $\subseteq$ *old q*

fun *auto-run* :: ['*a interprt*, '*a node set*] $\Rightarrow$ '*a node word*
where
auto-run ξ *nds 0*
= (*SOME q*. *q*$\in$*nds* $\wedge$ *expand-init* $\in$ *incoming q* $\wedge$ *auto-run-j 0* ξ *q*)
| *auto-run* ξ *nds* (*Suc k*)
= (*SOME q*$'$. *q*$'$$\in$*nds* $\wedge$ *name* (*auto-run* ξ *nds k*) $\in$ *incoming q*$'$
$\wedge$ *auto-run-j* (*Suc k*) ξ *q*$'$)

lemma *run-propag-on-create-graph*:
assumes *ipath gba.E* σ
shows σ *k*$\in$*qs* $\wedge$ *name* (σ *k*)$\in$*incoming* (σ (*k+1*))
using *assms*
by (*auto simp*: *create-gba-from-nodes--ipath*)

lemma *expand-false-propag*:
assumes *false*$_r$ $\notin$ *old* (*fst n-ns*) $\wedge$ ($\forall$ *nd*$\in$*snd n-ns. false*$_r$ $\notin$ *old nd*)
(**is** *?Q n-ns*)
shows *expand n-ns* $\leq$ *SPEC* (λ*nm-nds*. $\forall$ *nd*$\in$*snd nm-nds. false*$_r$ $\notin$ *old nd*)
using *assms*
proof (*rule-tac expand-rec-rule*[**where** Φ=*?Q*], *simp*, *intro refine-vcg*, *goal-cases*)
case *1*
then show *?case* **by** (*simp*, *rule-tac upd-incoming--ident*) *auto*
next
case *8*

then show *?case* **by** (*rule-tac SPEC-rule-nested2*) *auto*
qed *auto*

lemma *false-propag-on-create-graph*: *create-graph* $\varphi \leq$ *SPEC* (λ*nds*. $\forall$ *nd*$\in$*nds*. *false*$_r \notin$ *old nd*)
unfolding *create-graph-def*
by (*intro refine-vcg*, *rule-tac order-trans*, *rule-tac expand-false-propag*) *auto*

lemma *expand-and-propag*:
assumes μ *and*$_r$ $\eta \in$ *old* (*fst n-ns*)
$\longrightarrow \{\mu, \eta\} \subseteq$ *old* (*fst n-ns*) $\cup$ *new* (*fst n-ns*) (**is** *?Q n-ns*)
and $\forall$ *nd*$\in$*snd n-ns*. μ *and*$_r$ $\eta \in$ *old nd* $\longrightarrow \{\mu, \eta\} \subseteq$ *old nd* (**is** *?P* (*snd n-ns*))
shows *expand n-ns* $\leq$ *SPEC* (λ*nm-nds*. *?P* (*snd nm-nds*))
using *assms*
proof (*rule-tac expand-rec-rule*[**where** Φ=λ*x*. *?Q x* $\wedge$ *?P* (*snd x*)],
simp, *intro refine-vcg*, *goal-cases*)
case *1*
then show *?case* **by** (*simp*, *rule-tac upd-incoming--ident*) *auto*
next
case *prems*: (*4 f x n ns*)
then have *step*: $\bigwedge$*x*. *?Q x* $\wedge$ *?P* (*snd x*) $\Longrightarrow$ *f x* $\leq$ *SPEC* (λ*x'*. *?P* (*snd x'*)) **by** *simp*
with *prems* **show** *?case* **by** (*rule-tac step*) *auto*
next
case *prems*: (*5 f x n ns*)
then have *step*: $\bigwedge$*x*. *?Q x* $\wedge$ *?P* (*snd x*) $\Longrightarrow$ *f x* $\leq$ *SPEC* (λ*x'*. *?P* (*snd x'*)) **by** *simp*
with *prems* **show** *?case* **by** (*rule-tac step*) *auto*
next
case (*6 f x n ns*)
then show *?case* **by** *auto*
next
case *prems*: (*7 f x n ns*)
then have *step*: $\bigwedge$*x*. *?Q x* $\wedge$ *?P* (*snd x*) $\Longrightarrow$ *f x* $\leq$ *SPEC* (λ*x'*. *?P* (*snd x'*)) **by** *simp*
with *prems* **show** *?case* **by** (*rule-tac step*) *auto*
next
case *prems*: (*8 f x n ns* ψ)
then have *goal-assms*: $\psi \in$ *new n*
$\wedge \neg$ ($\exists$ *q*. ψ = *prop*$_r$(*q*) $\vee$ ψ = *nprop*$_r$(*q*))
$\wedge$ $\psi \neq$ *true*$_r$ $\wedge$ $\psi \neq$ *false*$_r$
$\wedge \neg$ ($\exists \nu$ μ. $\psi = \nu$ *and*$_r$ $\mu \vee \psi = X_r$ ν)
$\wedge$ ($\exists \nu$ μ. $\psi = \nu$ *or*$_r$ $\mu \vee \psi = \nu$ U_r $\mu \vee \psi = \nu$ R_r μ)
by (*cases* ψ) *auto*
from *prems* **have** *QP*: *?Q* (*n*, *ns*) $\wedge$ *?P ns*
and *step*: $\bigwedge$*x*. *?Q x* $\wedge$ *?P* (*snd x*) $\Longrightarrow$ *f x* $\leq$ *SPEC* (λ*x'*. *?P* (*snd x'*))
by *simp-all*
show *?case*

using *goal-assms QP* **unfolding** *case-prod-unfold* ‹$x = (n,\ ns)$›
proof (*rule-tac SPEC-rule-nested2*, *goal-cases*)
case *1*
then show *?case* **by** (*rule-tac step*) *auto*
next
case *2*
then show *?case* **by** (*rule-tac step*) *auto*
qed
qed *auto*

lemma *and-propag-on-create-graph*:
create-graph $\varphi \leq$ *SPEC* (λ*nds*. $\forall$ *nd*$\in$*nds*. μ *and*$_r$ $\eta \in$ *old nd* $\longrightarrow$ $\{\mu, \eta\} \subseteq$ *old nd*)
unfolding *create-graph-def*
by (*intro refine-vcg*, *rule-tac order-trans*, *rule-tac expand-and-propag*) *auto*

lemma *expand-or-propag*:
assumes μ *or*$_r$ $\eta \in$ *old* (*fst n-ns*)
$\longrightarrow \{\mu, \eta\} \cap$ (*old* (*fst n-ns*) $\cup$ *new* (*fst n-ns*)) $\neq \{\}$ (**is** *?Q n-ns*)
and $\forall$ *nd*$\in$*snd n-ns*. μ *or*$_r$ $\eta \in$ *old nd* $\longrightarrow \{\mu, \eta\} \cap$ *old nd* $\neq \{\}$
(**is** *?P* (*snd n-ns*))
shows *expand n-ns* $\leq$ *SPEC* (λ*nm-nds*. *?P* (*snd nm-nds*))
using *assms*
proof (*rule-tac expand-rec-rule*[**where** Φ=λ*x*. *?Q x* $\wedge$ *?P* (*snd x*)],
simp, *intro refine-vcg*, *goal-cases*)
case *1*
then show *?case* **by** (*simp*, *rule-tac upd-incoming--ident*) *auto*
next
case *prems*: (*4 f x n ns*)
then have *step*: $\bigwedge$*x*. *?Q x* $\wedge$ *?P* (*snd x*) $\Longrightarrow$ *f x* $\leq$ *SPEC* (λ*x'*. *?P* (*snd x'*)) **by** *simp*
with *prems* **show** *?case* **by** (*rule-tac step*) *auto*
next
case *prems*: (*5 f x n ns*)
then have *step*: $\bigwedge$*x*. *?Q x* $\wedge$ *?P* (*snd x*) $\Longrightarrow$ *f x* $\leq$ *SPEC* (λ*x'*. *?P* (*snd x'*)) **by** *simp*
with *prems* **show** *?case* **by** (*rule-tac step*) *auto*
next
case (*6 f x n ns*)
then show *?case* **by** *auto*
next
case *prems*: (*7 f x n ns*)
then have *step*: $\bigwedge$*x*. *?Q x* $\wedge$ *?P* (*snd x*) $\Longrightarrow$ *f x* $\leq$ *SPEC* (λ*x'*. *?P* (*snd x'*)) **by** *simp*
with *prems* **show** *?case* **by** (*rule-tac step*) *auto*
next
case *prems*: (*8 f x n ns* ψ)
then have *goal-assms*: $\psi \in$ *new n*
$\wedge \neg$ ($\exists$ *q*. ψ = *prop*$_r$(*q*) $\vee$ ψ = *nprop*$_r$(*q*))

$\wedge\ \psi \neq true_r \wedge \psi \neq false_r$
$\wedge\ \neg\ (\exists \nu\ \mu.\ \psi = \nu\ and_r\ \mu \vee \psi = X_r\ \nu)$
$\wedge\ (\exists \nu\ \mu.\ \psi = \nu\ or_r\ \mu \vee \psi = \nu\ U_r\ \mu \vee \psi = \nu\ R_r\ \mu)$
by (*cases* ψ) *auto*
from *prems* **have** *QP*: *?Q* (*n*, *ns*) ∧ *?P ns*
and *step*: $\bigwedge x$. *?Q x* ∧ *?P* (*snd x*) ⟹ *f x* ≤ *SPEC* (λ*x′*. *?P* (*snd x′*))
by *simp-all*
show *?case*
using *goal-assms QP*
unfolding *case-prod-unfold* ‹*x* = (*n*, *ns*)›
proof (*rule-tac SPEC-rule-nested2*, *goal-cases*)
case *1*
then show *?case* **by** (*rule-tac step*) *auto*
next
case *2*
then show *?case* **by** (*rule-tac step*) *auto*
qed
qed *auto*

lemma *or-propag-on-create-graph*:
create-graph φ ≤ *SPEC* (λ*nds*. ∀ *nd*∈*nds*. μ *or*$_r$ η ∈ *old nd* ⟶ {μ, η} ∩ *old nd* ≠ {})
unfolding *create-graph-def*
by (*intro refine-vcg*, *rule-tac order-trans*, *rule-tac expand-or-propag*) *auto*

abbreviation
next-propag--assm μ *n-ns* ≡
(X_r μ ∈ *old* (*fst n-ns*) ⟶ μ ∈ *next* (*fst n-ns*))
∧ (∀ *nd*∈*snd n-ns*. X_r μ ∈ *old nd* ∧ *name nd* ∈ *incoming* (*fst n-ns*)
⟶ μ ∈ *old* (*fst n-ns*) ∪ *new* (*fst n-ns*))

abbreviation
next-propag--rslt μ *ns* ≡
∀ *nd*∈*ns*. ∀ *nd′*∈*ns*. X_r μ ∈ *old nd* ∧ *name nd* ∈ *incoming nd′* ⟶ μ ∈ *old nd′*

lemma *expand-next-propag*:
fixes *n-ns* :: - × *′a node set*
assumes *next-propag--assm* μ *n-ns*
∧ *next-propag--rslt* μ (*snd n-ns*)
∧ *expand-assm-incoming n-ns*
∧ *expand-name-ident* (*snd n-ns*) (**is** *?Q n-ns*)
shows *expand n-ns* ≤ *SPEC* (λ*r*. *next-propag--rslt* μ (*snd r*))
(**is** - ≤ *SPEC ?P*)
using *assms*
proof (*rule-tac expand-rec-rule*[**where** Φ=*?Q*], *simp*, *intro refine-vcg*, *goal-cases*)
case (*1 f x n ns*)
then show *?case*
proof (*simp*, *rule-tac upd-incoming--ident*, *goal-cases*)

```
    case prems: 1
    {
      fix nd :: 'a node and nd' :: 'a node
      assume nd∈ns and nd'-elem: nd'∈upd-incoming n ns
      have μ ∈ old nd' if *: X_r μ ∈ old nd and **: name nd ∈ incoming nd'
      proof (cases nd'∈ns)
        case True
        with prems * ** show ?thesis using ‹nd∈ns› by auto
      next
        case False
        with upd-incoming--elem[of nd' n ns] nd'-elem * **
        obtain nd'' where nd''∈ns
          and nd'-eq: nd' = nd''(|incoming := incoming n ∪ incoming nd''|)
          and old-eq: old nd'' = old n by auto
        have μ ∈ old nd'
        proof (cases name nd ∈ incoming n)
          case True
          with prems * ‹nd∈ns› have μ ∈ old n by auto
          then show ?thesis using nd'-eq old-eq by simp
        next
          case False
          then have name nd ∈ incoming nd''
            using ‹name nd ∈incoming nd'› nd'-eq by simp
          then show ?thesis
            unfolding nd'-eq using ‹nd∈ns› ‹nd''∈ns› * prems by auto
        qed
        then show ?thesis by auto
      qed
    }
    then show ?case by auto
  next
    case 2
    then show ?case by simp
  qed
next
  case prems: (2 f x n ns)
  then have step: ⋀x. ?Q x ⟹ f x ≤ SPEC ?P
    and f-sup: ⋀x. f x ≤ expand x by auto
  from prems have Q: ?Q (n, ns) by auto
  from Q have name-le: name n < expand-new-name (name n) by auto
  let ?x' = (|name = expand-new-name (name n),
              incoming = {name n}, new = next n,
              old = {}, next = {}|), {n} ∪ ns)
  have Q'1: expand-assm-incoming ?x' ∧ expand-name-ident (snd ?x')
    using ‹new n = {}› Q[THEN conjunct2, THEN conjunct2] name-le by force
  have Q'2: next-propag--assm μ ?x' ∧ next-propag--rslt μ (snd ?x')
    using Q ‹new n = {}› by auto
  show ?case
    using ‹new n = {}›
```

```
    unfolding ‹x = (n, ns)›
  proof (rule-tac SPEC-rule-weak[where
      Q = λ- r. expand-name-ident (snd r) and P = λ-. ?P], goal-cases)
    case 1
    then show ?case
    using Q'1
      by (rule-tac order-trans,
        rule-tac f-sup,
        rule-tac expand-name-propag--name-ident) auto
  next
    case 2
    then show ?case
      using Q'1 Q'2 by (rule-tac step) simp
  next
    case (3 nm nds)
    then show ?case by simp
  qed
next
  case 3
  then show ?case by auto
next
  case prems: (4 f)
  then have step: ⋀x. ?Q x ⟹ f x ≤ SPEC ?P by simp
  from prems show ?case by (rule-tac SPEC-rule-param2, rule-tac step) auto
next
  case prems: (5 f)
  then have step: ⋀x. ?Q x ⟹ f x ≤ SPEC ?P by simp
  from prems show ?case by (rule-tac SPEC-rule-param2, rule-tac step) auto
next
  case 6
  then show ?case by auto
next
  case prems: (7 f)
  then have step: ⋀x. ?Q x ⟹ f x ≤ SPEC ?P by simp
  from prems show ?case by (rule-tac SPEC-rule-param2, rule-tac step) auto
next
  case prems: (8 f x n ns ψ)
  then have goal-assms: ψ ∈ new n ∧ (∃ν μ. ψ = ν or_r μ ∨ ψ = ν U_r μ ∨ ψ =
ν R_r μ)
    by (cases ψ) auto
  from prems have Q: ?Q (n, ns)
      and step: ⋀x. ?Q x ⟹ f x ≤ SPEC ?P
      and f-sup: ⋀x. f x ≤ expand x
    by auto
  let ?x = (n(|new := new n − {ψ}, new := new1 ψ ∪ new (n(|new := new n −
{ψ}|)),
        old := {ψ} ∪ old (n(|new := new n − {ψ}|)),
                        next := next1 ψ ∪ next (n(|new := new n − {ψ}|)
        |), ns)
```

let *?props* = $\lambda x\ r.$ *expand-rslt-exist-eq x r*
$\wedge$ *expand-rslt-incoming r* $\wedge$ *expand-rslt-name x r* $\wedge$ *expand-name-ident* (*snd r*)
show *?case*
using *goal-assms Q* **unfolding** *case-prod-unfold* ‹$x = (n,\ ns)$›
proof (*rule-tac SPEC-rule-weak-nested2*[**where** Q = *?props ?x*], *goal-cases*)
case *1* **then**
show *?case*
by (*rule-tac SPEC-rule-conjI*,
rule-tac order-trans,
rule-tac f-sup,
rule-tac expand-rslt-exist-eq,
rule-tac order-trans,
rule-tac f-sup,
rule-tac expand-name-propag) *simp+*
next
case *2*
then show *?case*
by (*rule-tac SPEC-rule-param2*[**where** $P = \lambda$-. *?P*], *rule-tac step*) *auto*
next
case *prems'*: (*3 nm nds*)
let *?x'* = (*n*(|*new* := *new n* − {ψ},
name := *fst* (*nm*, *nds*),
new := *new2* ψ ∪ *new* (*n*(|*new* := *new n* − {ψ}|)),
old := {ψ} ∪ *old* (*n*(|*new* := *new n* − {ψ}|))|), *nds*)
from *prems'* **show** *?case*
proof (*rule-tac step*, *goal-cases*)
case *prems''*: *1*
then have *expand-assm-incoming ?x'* **by** *auto*
moreover
from *prems''* **have** *nds-ident*: *expand-name-ident* (*snd ?x'*) **by** *simp*
moreover
have $X_r\ \mu \in$ *old* (*fst ?x'*) $\longrightarrow \mu \in$ *next* (*fst ?x'*)
using *Q*[*THEN conjunct1*] *goal-assms* **by** *auto*
moreover
from *prems''* **have** *next-propag--rslt* μ (*snd ?x'*) **by** *simp*
moreover
from *prems''* **have** *name-nds-eq*: *name* ‘ *nds* = *name* ‘ *ns* ∪ *name* ‘ {*nd*∈*nds*. *name nd* ≥ *name n*}
by *auto*
have ∀ *nd*∈*nds*. ($X_r\ \mu \in$ *old nd* ∧ *name nd* ∈ *incoming* (*fst ?x'*))
$\longrightarrow \mu \in$ *old* (*fst ?x'*)∪*new* (*fst ?x'*)
(**is** ∀ *nd*∈*nds*. *?assm* (*fst ?x'*) *nd* $\longrightarrow$ *?concl* (*fst ?x'*) *nd*)
proof
fix *nd*
assume *nd*∈*nds*
{ **assume** *loc-assm*: *name nd*∈*name* ‘ *ns*
then obtain *n'* **where** *n'*: *n'*∈*ns name n'* = *name nd* **by** *auto*
moreover
from *prems''* *n'* **obtain** *nd'* **where** *nd'*∈*nds*

```
            and n'-nd'-eq: expand-rslt-exist-eq--node n' nd'
           by auto
         ultimately have nd = nd'
           using nds-ident ‹nd∈nds› loc-assm by auto
         moreover from prems'' have ?assm n n' ⟶ ?concl n n'
           using ‹n'∈ns› by auto
         ultimately have ?assm (fst ?x') nd ⟶ ?concl (fst ?x') nd
           using n'-nd'-eq by auto }
       moreover
       { assume name nd∉name ‘ ns
         with name-nds-eq ‹nd∈nds› have name nd ≥ name n by auto
         with prems'' have ¬ (?assm (fst ?x') nd) by auto }
       ultimately show ?assm (fst ?x') nd ⟶ ?concl (fst ?x') nd by auto
     qed
     ultimately show ?case by simp
   qed
 qed
qed

lemma next-propag-on-create-graph:
 create-graph φ ≤ SPEC (λnds. ∀ n∈nds. ∀ n'∈nds. X_r μ∈old n ∧ name n∈incoming
n' ⟶ μ∈old n')
 unfolding create-graph-def
 apply (refine-vcg expand-next-propag)
 apply (auto simp add:expand-new-name-expand-init)
 done


abbreviation
 release-propag--assm μ η n-ns ≡
   (μ R_r η ∈ old (fst n-ns)
    ⟶ {μ, η}⊆old (fst n-ns)∪new (fst n-ns) ∨
      (η∈old (fst n-ns)∪new (fst n-ns)) ∧ μ R_r η ∈ next (fst n-ns))
   ∧ (∀ nd∈snd n-ns.
      μ R_r η ∈ old nd ∧ name nd ∈ incoming (fst n-ns)
      ⟶ {μ, η}⊆old nd ∨
        (η∈old nd ∧ μ R_r η ∈ old (fst n-ns)∪new (fst n-ns)))

abbreviation
 release-propag--rslt μ η ns ≡
   ∀ nd∈ns.
    ∀ nd'∈ns.
      μ R_r η ∈ old nd ∧ name nd ∈ incoming nd'
      ⟶ {μ, η}⊆old nd ∨
        (η∈old nd ∧ μ R_r η ∈ old nd')

lemma expand-release-propag:
 fixes n-ns :: - × 'a node set
 assumes release-propag--assm μ η n-ns
```

```
        ∧ release-propag--rslt μ η (snd n-ns)
        ∧ expand-assm-incoming n-ns
        ∧ expand-name-ident (snd n-ns) (is ?Q n-ns)
  shows expand n-ns ≤ SPEC (λr. release-propag--rslt μ η (snd r))
    (is - ≤ SPEC ?P)
  using assms
proof (rule-tac expand-rec-rule[where Φ=?Q], simp, intro refine-vcg, goal-cases)
  case (1 f x n ns)
  then show ?case
  proof (simp, rule-tac upd-incoming--ident, goal-cases)
    case prems: 1
    { fix nd :: 'a node and nd' :: 'a node
      let ?V-prop = μ R_r η ∈ old nd ∧ name nd ∈ incoming nd'
        ⟶ {μ, η} ⊆ old nd ∨ η ∈ old nd ∧ μ R_r η ∈ old nd'
      assume nd∈ns and nd'-elem: nd'∈upd-incoming n ns
      { assume nd'∈ns
        with prems have ?V-prop using ‹nd∈ns› by auto }
      moreover
      { assume nd'∉ns
          and V-in-nd: μ R_r η ∈ old nd and name nd ∈incoming nd'
        with upd-incoming--elem[of nd' n ns] nd'-elem
        obtain nd'' where nd''∈ns
          and nd'-eq: nd' = nd''(|incoming := incoming n ∪ incoming nd''|)
          and old-eq: old nd'' = old n
          by auto
        { assume name nd ∈ incoming n
          with prems V-in-nd ‹nd∈ns›
          have {μ, η} ⊆ old nd ∨ η ∈ old nd ∧ μ R_r η ∈ old n
            by auto
          then have {μ, η} ⊆ old nd ∨ η ∈ old nd ∧ μ R_r η ∈ old nd'
            using nd'-eq old-eq by simp }
        moreover
        { assume name nd ∉ incoming n
          then have name nd ∈ incoming nd''
            using ‹name nd ∈incoming nd'› nd'-eq by simp
          then have {μ, η} ⊆ old nd ∨ η ∈ old nd ∧ μ R_r η ∈ old nd'
            unfolding nd'-eq
            using prems ‹nd∈ns› ‹nd''∈ns› V-in-nd by auto }
        ultimately have {μ, η} ⊆ old nd ∨ η ∈ old nd ∧ μ R_r η ∈ old nd'
          by fast
      }
      ultimately have ?V-prop by auto
    }
    then show ?case by auto
  next
    case 2
    then show ?case by simp
  qed
next
```

```
  case prems: (2 f x n ns)
  then have step: ⋀x. ?Q x ⟹ f x ≤ SPEC ?P
    and f-sup: ⋀x. f x ≤ expand x by auto
  from prems have Q: ?Q (n, ns) by auto
  from Q have name-le: name n < expand-new-name (name n) by auto
  let ?x' = (⦇name = expand-new-name (name n),
              incoming = {name n}, new = next n,
              old = {}, next = {}⦈, {n} ∪ ns)
  have Q'1: expand-assm-incoming ?x' ∧ expand-name-ident (snd ?x')
  using ‹new n = {}› Q[THEN conjunct2, THEN conjunct2] name-le by force
  have Q'2: release-propag--assm μ η ?x' ∧ release-propag--rslt μ η (snd ?x')
    using Q ‹new n = {}› by auto

  show ?case using ‹new n = {}› unfolding ‹x = (n, ns)›

  proof (rule-tac SPEC-rule-weak[where
      Q = λ- r. expand-name-ident (snd r) and P = λ-. ?P], goal-cases)
    case 1
    then show ?case using Q'1
      by (rule-tac order-trans,
        rule-tac f-sup,
        rule-tac expand-name-propag--name-ident) auto
  next
    case 2
    then show ?case using Q'1 Q'2 by (rule-tac step) simp
  next
    case (3 nm nds)
    then show ?case by simp
  qed
next
  case 3 then show ?case by auto
next
  case prems: (4 f x n ns ψ)
  then have goal-assms: ψ ∈ new n ∧ (∃ q. ψ = prop_r(q) ∨ ψ = nprop_r(q)) by
simp
  from prems have step: ⋀x. ?Q x ⟹ f x ≤ SPEC ?P and Q: ?Q (n, ns)
    by simp-all
  show ?case
    using Q goal-assms by (rule-tac SPEC-rule-param2, rule-tac step) auto
next
  case prems: (5 f x n ns ψ)
  then have goal-assms: ψ ∈ new n ∧ ψ = true_r by simp
  from prems have step: ⋀x. ?Q x ⟹ f x ≤ SPEC ?P and Q: ?Q (n, ns)
    by simp-all
  show ?case using Q goal-assms by (rule-tac SPEC-rule-param2, rule-tac step)
auto
next
  case 6
  then show ?case by auto
```

next
case *prems*: (*7 f x n ns* ψ)
then have *goal-assms*: $\psi \in$ *new n* $\wedge$ ($\exists \nu\ \mu.\ \psi = \nu$ *and*$_r$ $\mu \vee \psi = X_r\ \nu$) **by** *simp*
from *prems* **have** *step*: $\bigwedge x.$ *?Q x* $\Longrightarrow$ *f x* $\leq$ *SPEC ?P* **and** *Q*: *?Q* (*n*, *ns*)
by *simp-all*
show *?case* **using** *Q goal-assms* **by** (*rule-tac SPEC-rule-param2*, *rule-tac step*) *auto*
next
case *prems*: (*8 f x n ns* ψ)
then have *goal-assms*: $\psi \in$ *new n* $\wedge$ ($\exists \nu\ \mu.\ \psi = \nu$ *or*$_r$ $\mu \vee \psi = \nu\ U_r\ \mu \vee \psi = \nu\ R_r\ \mu$)
by (*cases* ψ) *auto*
from *prems* **have** *Q*: *?Q* (*n*, *ns*)
and *step*: $\bigwedge x.$ *?Q x* $\Longrightarrow$ *f x* $\leq$ *SPEC ?P*
and *f-sup*: $\bigwedge x.$ *f x* $\leq$ *expand x* **by** *auto*
let *?x* = (*n*(|*new* := *new n* $-$ {ψ}, *new* := *new1* ψ $\cup$ *new* (*n*(|*new* := *new n* $-$ {ψ}|)),
old := {ψ} $\cup$ *old* (*n*(|*new* := *new n* $-$ {ψ}|)),
next := *next1* ψ $\cup$ *next* (*n*(|*new* := *new n* $-$ {ψ}|)
|), *ns*)
let *?props* = λ*x r. expand-rslt-exist-eq x r*
$\wedge$ *expand-rslt-incoming r* $\wedge$ *expand-rslt-name x r* $\wedge$ *expand-name-ident* (*snd r*)

show *?case* **using** *goal-assms Q* **unfolding** *case-prod-unfold* ‹*x* = (*n*, *ns*)›

proof (*rule-tac SPEC-rule-weak-nested2*[**where** *Q* = *?props ?x*], *goal-cases*)
case *1*
then show *?case*
by (*rule-tac SPEC-rule-conjI*,
rule-tac order-trans,
rule-tac f-sup,
rule-tac expand-rslt-exist-eq,
rule-tac order-trans,
rule-tac f-sup,
rule-tac expand-name-propag) *simp+*
next
case *2*
then show *?case*
by (*rule-tac SPEC-rule-param2*[**where** *P* = λ-. *?P*], *rule-tac step*) *auto*
next
case *prems′*: (*3 nm nds*)
let *?x′* = (*n*(|*new* := *new n* $-$ {ψ},
name := *fst* (*nm*, *nds*),
new := *new2* ψ $\cup$ *new* (*n*(|*new* := *new n* $-$ {ψ}|)),
old := {ψ} $\cup$ *old* (*n*(|*new* := *new n* $-$ {ψ}|))|), *nds*)
from *prems′* **show** *?case*
proof (*rule-tac step*, *goal-cases*)
case *prems″*: *1*
then have *expand-assm-incoming ?x′* **by** *auto*

```
      moreover
      from prems'' have nds-ident: expand-name-ident (snd ?x') by simp
      moreover
      have (μ R_r η ∈ old (fst ?x')
            ⟶ ({μ, η}⊆old (fst ?x') ∪ new (fst ?x')
                ∨ (η∈old (fst ?x')∪new (fst ?x')
                    ∧ μ R_r η ∈ next (fst ?x'))))
        using Q[THEN conjunct1] goal-assms by auto
      moreover
      from prems'' have release-propag--rslt μ η (snd ?x') by simp
      moreover
     from prems'' have name-nds-eq: name ‘ nds = name ‘ ns ∪ name ‘ {nd∈nds.
name nd ≥ name n}
        by auto
      have ∀ nd∈nds. (μ R_r η ∈ old nd ∧ name nd ∈ incoming (fst ?x'))
              ⟶ ({μ, η}⊆old nd
                  ∨ (η∈old nd ∧ μ R_r η ∈old (fst ?x')∪new (fst ?x')))
       (is ∀ nd∈nds. ?assm (fst ?x') nd ⟶ ?concl (fst ?x') nd)
      proof
        fix nd
        assume nd∈nds
        { assume loc-assm: name nd∈name ‘ ns
          then obtain n' where n': n'∈ns name n' = name nd by auto
          with prems'' obtain nd' where nd'∈nds
            and n'-nd'-eq: expand-rslt-exist-eq--node n' nd'
            by auto
          with n' have nd = nd' using nds-ident ‹nd∈nds› loc-assm
            by auto
          moreover from prems'' have ?assm n n' ⟶ ?concl n n'
            using ‹n'∈ns› by auto
          ultimately have ?assm (fst ?x') nd ⟶ ?concl (fst ?x') nd
            using n'-nd'-eq by auto }
        moreover
        { assume name nd∉name ‘ ns
          with name-nds-eq ‹nd∈nds› have name nd ≥ name n by auto
          with prems'' have ¬ (?assm (fst ?x') nd) by auto }
        ultimately show ?assm (fst ?x') nd ⟶ ?concl (fst ?x') nd by auto
      qed
      ultimately show ?case by simp
    qed
  qed
qed

lemma release-propag-on-create-graph:
  create-graph φ
    ≤ SPEC (λnds. ∀ n∈nds. ∀ n'∈nds. μ R_r η∈old n ∧ name n∈incoming n'
                                ⟶ ({μ, η}⊆old n ∨ η∈old n ∧ μ R_r η∈old n'))
  unfolding create-graph-def
  apply (refine-vcg expand-release-propag)
```

by (*auto simp add*:*expand-new-name-expand-init*)

abbreviation
until-propag--assm f g n-ns ≡
(*f* U_r *g* ∈ *old* (*fst n-ns*)
⟶ (*g*∈*old* (*fst n-ns*)∪*new* (*fst n-ns*)
∨ (*f*∈*old* (*fst n-ns*)∪*new* (*fst n-ns*) ∧ *f* U_r *g* ∈ *next* (*fst n-ns*))))
∧ (∀ *nd*∈*snd n-ns*. *f* U_r *g* ∈ *old nd* ∧ *name nd* ∈ *incoming* (*fst n-ns*)
⟶ (*g*∈*old nd* ∨ (*f*∈*old nd* ∧ *f* U_r *g* ∈*old* (*fst n-ns*)∪*new* (*fst n-ns*))))

abbreviation
until-propag--rslt f g ns ≡
∀ *n*∈*ns*. ∀ *nd*∈*ns*. *f* U_r *g* ∈ *old n* ∧ *name n* ∈ *incoming nd*
⟶ (*g* ∈ *old n* ∨ (*f*∈*old n* ∧ *f* U_r *g* ∈ *old nd*))

lemma *expand-until-propag*:
fixes *n-ns* :: - × *'a node set*
assumes *until-propag--assm* μ η *n-ns*
∧ *until-propag--rslt* μ η (*snd n-ns*)
∧ *expand-assm-incoming n-ns*
∧ *expand-name-ident* (*snd n-ns*) (**is** *?Q n-ns*)
shows *expand n-ns* ≤ *SPEC* (λ*r*. *until-propag--rslt* μ η (*snd r*))
(**is** - ≤ *SPEC ?P*)
using *assms*
proof (*rule-tac expand-rec-rule*[**where** Φ=*?Q*], *simp*, *intro refine-vcg*, *goal-cases*)
case *prems*: (*1 f x n ns*)
then show *?case*
proof (*simp*, *rule-tac upd-incoming--ident*, *goal-cases*)
case *prems'*: *1*
{ fix *nd* :: *'a node* **and** *nd'* :: *'a node*
let *?U-prop* = μ U_r η ∈ *old nd* ∧ *name nd* ∈ *incoming nd'*
⟶ η ∈ *old nd* ∨ μ ∈ *old nd* ∧ μ U_r η ∈ *old nd'*
assume *nd*∈*ns* **and** *nd'-elem*: *nd'*∈*upd-incoming n ns*
{ assume *nd'*∈*ns*
with *prems'* **have** *?U-prop* **using** ‹*nd*∈*ns*› **by** *auto* **}**
moreover
{ assume *nd'*∉*ns* **and**
U-in-nd: μ U_r η ∈ *old nd* **and** *name nd* ∈*incoming nd'*
with *upd-incoming--elem*[*of nd' n ns*] *nd'-elem*
obtain *nd''* **where** *nd''*∈*ns*
and *nd'-eq*: *nd'* = *nd''*(|*incoming* := *incoming n* ∪ *incoming nd''*|)
and *old-eq*: *old nd''* = *old n* **by** *auto*
{ assume *name nd* ∈ *incoming n*
with *prems' U-in-nd* ‹*nd*∈*ns*›
have η ∈ *old nd* ∨ μ ∈ *old nd* ∧ μ U_r η ∈ *old n* **by** *auto*
then have η ∈ *old nd* ∨ μ ∈ *old nd* ∧ μ U_r η ∈ *old nd'*
using *nd'-eq old-eq* **by** *simp* **}**
moreover

```
      { assume name nd ∉ incoming n
        then have name nd ∈ incoming nd''
          using ‹name nd ∈incoming nd'› nd'-eq by simp
        then have η ∈ old nd ∨ μ ∈ old nd ∧ μ U_r η ∈ old nd'
          unfolding nd'-eq
          using prems' ‹nd∈ns› ‹nd''∈ns› U-in-nd by auto }
      ultimately have η ∈ old nd ∨ μ ∈ old nd ∧ μ U_r η ∈ old nd' by fast }
    ultimately have ?U-prop by auto }
  then show ?case by auto
 next
  case 2
  then show ?case by simp
 qed
next
 case prems: (2 f x n ns)
 then have step: ⋀x. ?Q x ⟹ f x ≤ SPEC ?P and f-sup: ⋀x. f x ≤ expand x
  by auto
 from prems have Q: ?Q (n, ns) by auto
 from Q have name-le: name n < expand-new-name (name n) by auto
 let ?x' = (⦇name = expand-new-name (name n),
            incoming = {name n}, new = next n,
            old = {}, next = {}⦈, {n} ∪ ns)
 have Q'1: expand-assm-incoming ?x' ∧ expand-name-ident (snd ?x')
  using ‹new n = {}› Q[THEN conjunct2, THEN conjunct2] name-le by force
 have Q'2: until-propag--assm μ η ?x' ∧ until-propag--rslt μ η (snd ?x')
  using Q ‹new n = {}› by auto

 show ?case
  using ‹new n = {}› unfolding ‹x = (n, ns)›
 proof (rule-tac SPEC-rule-weak[where
    Q = λ- r. expand-name-ident (snd r) and P = λ-. ?P], goal-cases)
  case 1
  then show ?case using Q'1
   by (rule-tac order-trans,
     rule-tac f-sup,
     rule-tac expand-name-propag--name-ident) auto
 next
  case 2
  then show ?case using Q'1 Q'2 by (rule-tac step) simp
 next
  case (3 nm nds)
  then show ?case by simp
 qed
next
 case 3
 then show ?case by auto
next
 case prems: (4 f)
 then have step: ⋀x. ?Q x ⟹ f x ≤ SPEC ?P by simp-all
```

```
  from prems show ?case by (rule-tac SPEC-rule-param2, rule-tac step) auto
next
  case prems: (5 f)
  then have step: ⋀x. ?Q x ⟹ f x ≤ SPEC ?P by simp-all
  from prems show ?case by (rule-tac SPEC-rule-param2, rule-tac step) auto
next
  case 6
  then show ?case by auto
next
  case prems: (7 f)
  then have step: ⋀x. ?Q x ⟹ f x ≤ SPEC ?P by simp-all
  from prems show ?case by (rule-tac SPEC-rule-param2, rule-tac step) auto
next
  case prems: (8 f x n ns ψ)
  then have goal-assms: ψ ∈ new n ∧ (∃ν μ. ψ = ν or_r μ ∨ ψ = ν U_r μ ∨ ψ =
ν R_r μ)
    by (cases ψ) auto
  from prems have Q: ?Q (n, ns)
    and step: ⋀x. ?Q x ⟹ f x ≤ SPEC ?P
    and f-sup: ⋀x. f x ≤ expand x by auto
  let ?x = (n(|new := new n − {ψ}, new := new1 ψ ∪ new (n(|new := new n −
{ψ}|)),
        old := {ψ} ∪ old (n(|new := new n − {ψ}|)),
                      next := next1 ψ ∪ next (n(|new := new n − {ψ}|)
      |), ns)
  let ?props = λx r. expand-rslt-exist-eq x r
    ∧ expand-rslt-incoming r ∧ expand-rslt-name x r ∧ expand-name-ident (snd r)

  show ?case
    using goal-assms Q unfolding case-prod-unfold ‹x = (n, ns)›
  proof (rule-tac SPEC-rule-weak-nested2[where Q = ?props ?x], goal-cases)
    case 1
    then show ?case
      by (rule-tac SPEC-rule-conjI,
          rule-tac order-trans,
          rule-tac f-sup,
          rule-tac expand-rslt-exist-eq,
          rule-tac order-trans,
          rule-tac f-sup,
          rule-tac expand-name-propag) simp+
  next
    case 2
    then show ?case
      by (rule-tac SPEC-rule-param2[where P = λ-. ?P], rule-tac step) auto
  next
    case prems: (3 nm nds)
    let ?x' = (n(|new := new n − {ψ},
      name := fst (nm, nds),
      new := new2 ψ ∪ new (n(|new := new n − {ψ}|)),
```

```
      old := {ψ} ∪ old (n(|new := new n − {ψ}|))|), nds)
    from prems show ?case
    proof (rule-tac step, goal-cases)
      case prems': 1
      then have expand-assm-incoming ?x' by auto
      moreover
      from prems' have nds-ident: expand-name-ident (snd ?x')
        by simp
      moreover
      have (μ U_r η ∈ old (fst ?x')
            ⟶ (η∈old (fst ?x')∪new (fst ?x')
                ∨ (μ∈old (fst ?x')∪new (fst ?x')
                    ∧ μ U_r η ∈ next (fst ?x'))))
        using Q[THEN conjunct1] goal-assms by auto
      moreover
      from prems' have until-propag--rslt μ η (snd ?x')
        by simp
      moreover
      from prems' have name-nds-eq:
        name ‘ nds = name ‘ ns ∪ name ‘ {nd∈nds. name nd ≥ name n}
        by auto
      have ∀ nd∈nds. (μ U_r η ∈ old nd ∧ name nd ∈ incoming (fst ?x'))
        ⟶ (η∈old nd ∨ (μ∈old nd ∧ μ U_r η ∈old (fst ?x')∪new (fst ?x')))
       (is ∀ nd∈nds. ?assm (fst ?x') nd ⟶ ?concl (fst ?x') nd)
      proof
        fix nd
        assume nd∈nds
        { assume loc-assm: name nd∈name ‘ ns
          then obtain n' where n': n'∈ns name n' = name nd by auto
          moreover
          from prems' n' obtain nd' where nd'∈nds
            and n'-nd'-eq: expand-rslt-exist-eq--node n' nd'
            by auto
          ultimately have nd = nd'
            using nds-ident ‹nd∈nds› loc-assm by auto
          moreover from prems' have ?assm n n' ⟶ ?concl n n'
            using ‹n'∈ns› by auto
          ultimately have ?assm (fst ?x') nd ⟶ ?concl (fst ?x') nd
            using n'-nd'-eq by auto }
        moreover
        { assume name nd∉name ‘ ns
          with name-nds-eq ‹nd∈nds› have name nd ≥ name n by auto
          with prems' have ¬ (?assm (fst ?x') nd) by auto }
        ultimately show ?assm (fst ?x') nd ⟶ ?concl (fst ?x') nd by auto
      qed
      ultimately show ?case by simp
    qed
  qed
qed
```

lemma *until-propag-on-create-graph*:
create-graph $\varphi \leq$ *SPEC* (λ*nds*. $\forall$ *n*$\in$*nds*. $\forall$ *n'*$\in$*nds*. μ U_r $\eta\in$*old n* $\wedge$ *name n*$\in$*incoming n'*
$\longrightarrow$ ($\eta\in$*old n* $\vee$ $\mu\in$*old n* $\wedge$ μ U_r $\eta\in$*old n'*))
unfolding *create-graph-def*
apply (*refine-vcg expand-until-propag*)
by (*auto simp add*:*expand-new-name-expand-init*)

definition *all-subfrmls* :: *'a node* $\Rightarrow$ *'a frml set*
where *all-subfrmls n* $\equiv \bigcup$(*subfrmlsr* ' (*new n* $\cup$ *old n* $\cup$ *next n*))

lemma *all-subfrmls--UnionD*:
assumes ($\bigcup x\in A$. *subfrmlsr x*) $\subseteq$ *B*
and *x*$\in$*A*
and *y*$\in$*subfrmlsr x*
shows *y*$\in$*B*
proof −
note *subfrmlsr-id*[*of x*]
also have *subfrmlsr x* $\subseteq$ ($\bigcup x\in A$. *subfrmlsr x*)
using *assms* **by** *auto*
finally show *?thesis* **using** *assms* **by** *auto*
qed

lemma *expand-all-subfrmls-propag*:
assumes *all-subfrmls* (*fst n-ns*) $\subseteq$ *B* $\wedge$ ($\forall$ *nd*$\in$*snd n-ns*. *all-subfrmls nd* $\subseteq$ *B*) (**is** *?Q n-ns*)
shows *expand n-ns* $\leq$ *SPEC* (λ*r*. $\forall$ *nd*$\in$*snd r*. *all-subfrmls nd* $\subseteq$ *B*)
(**is** - $\leq$ *SPEC ?P*)
using *assms*
proof (*rule-tac expand-rec-rule*[**where** Φ=*?Q*], *simp*, *intro refine-vcg*, *goal-cases*)
case *1*
then show *?case*
proof (*simp*, *rule-tac upd-incoming--ident*, *goal-cases*)
case *1*
then show *?case* **by** *auto*
next
case *2*
then show *?case* **by** (*simp add*: *all-subfrmls-def*)
qed
next
case *2*
then show *?case* **by** (*auto simp add*: *all-subfrmls-def*)
next
case *3*
then show *?case* **by** (*auto simp add*: *all-subfrmls-def*)
next
case *prems*: (*4 f*)

then have *step*: $\bigwedge x.\ ?Q\ x \Longrightarrow f\ x \leq SPEC\ ?P$ **by** *simp-all*
from *prems* **show** *?case* **by** (*rule-tac step*) (*auto simp add*: *all-subfrmls-def*)
next
case *prems*: (*5 f - n ns* ψ)
then have *goal-assms*: $\psi \in new\ n \wedge \psi = true_r$ **by** *simp*
from *prems* **have** *step*: $\bigwedge x.\ ?Q\ x \Longrightarrow f\ x \leq SPEC\ ?P$ **and** *Q*: *?Q* (*n*, *ns*)
by *simp-all*
show *?case* **using** *Q goal-assms*
by (*rule-tac step*) (*auto dest*: *all-subfrmls--UnionD simp add*: *all-subfrmls-def*)
next
case *6*
then show *?case* **by** *auto*
next
case *prems*: (*7 f x n ns* ψ)
then have *goal-assms*: $\psi \in new\ n \wedge (\exists \nu\ \mu.\ \psi = \nu\ and_r\ \mu \vee \psi = X_r\ \nu)$ **by** *simp*
from *prems* **have** *step*: $\bigwedge x.\ ?Q\ x \Longrightarrow f\ x \leq SPEC\ ?P$ **and** *Q*: *?Q* (*n*, *ns*)
by *simp-all*
show *?case*
using *Q goal-assms*
by (*rule-tac step*) (*auto dest*: *all-subfrmls--UnionD simp add*: *all-subfrmls-def*)
next
case *prems*: (*8 f x n ns* ψ)
then have *goal-assms*: $\psi \in new\ n$
$\wedge\ \neg\ (\exists\, q.\ \psi = prop_r(q) \vee \psi = nprop_r(q))$
$\wedge\ \psi \neq true_r \wedge \psi \neq false_r$
$\wedge\ \neg\ (\exists \nu\ \mu.\ \psi = \nu\ and_r\ \mu \vee \psi = X_r\ \nu)$
$\wedge\ (\exists \nu\ \mu.\ \psi = \nu\ or_r\ \mu \vee \psi = \nu\ U_r\ \mu \vee \psi = \nu\ R_r\ \mu)$
by (*cases* ψ) *auto*
from *prems* **have** *Q*: *?Q* (*n*, *ns*) **and** *step*: $\bigwedge x.\ ?Q\ x \Longrightarrow f\ x \leq SPEC\ ?P$
by *simp-all*
show *?case*
using *goal-assms Q* **unfolding** *case-prod-unfold* ‹$x = (n,\ ns)$›
proof (*rule-tac SPEC-rule-nested2*, *goal-cases*)
case *1*
then show *?case*
by (*rule-tac step*) (*auto dest*: *all-subfrmls--UnionD simp*: *all-subfrmls-def*)
next
case *2*
then show *?case*
by (*rule-tac step*) (*auto dest*: *all-subfrmls--UnionD simp*: *all-subfrmls-def*)
qed
qed

lemma *old-propag-on-create-graph*: *create-graph* $\varphi \leq SPEC$ ($\lambda nds.\ \forall\, n{\in}nds.\ old\ n \subseteq subfrmlsr\ \varphi$)
unfolding *create-graph-def*
by (*intro refine-vcg*,
rule-tac order-trans,
rule-tac expand-all-subfrmls-propag[**where** $B = subfrmlsr\ \varphi$])

(*force simp add:all-subfrmls-def expand-new-name-expand-init*)+

lemma *L4-2--aux*:
assumes *run*: *ipath gba.E* σ
and $\mu\ U_r\ \eta \in$ *old* $(\sigma\ 0)$
and $\forall j.\ (\forall i{<}j.\ \{\mu,\ \mu\ U_r\ \eta\} \subseteq$ *old* $(\sigma\ i)) \longrightarrow \eta \notin$ *old* $(\sigma\ j)$
shows $\forall i.\ \{\mu,\ \mu\ U_r\ \eta\} \subseteq$ *old* $(\sigma\ i) \wedge \eta \notin$ *old* $(\sigma\ i)$
proof −
have $\forall i{<}j.\ \{\mu,\ \mu\ U_r\ \eta\} \subseteq$ *old* $(\sigma\ i)$ (**is** *?sbthm j*) **for** *j*
proof (*induct j*)
show *?sbthm 0* **by** *auto*
next
fix *k*
assume *step*: *?sbthm k*
then have σ*-k-prop*: $\eta \notin$ *old* $(\sigma\ k)$
$\wedge\ \sigma\ k{\in}qs \wedge \sigma\ (Suc\ k){\in}qs$
$\wedge$ *name* $(\sigma\ k) \in$ *incoming* $(\sigma\ (Suc\ k))$
using *assms run-propag-on-create-graph*[*OF run*] **by** *auto*
with *inres-SPEC*[*OF res until-propag-on-create-graph*[**where** $\mu = \mu$ **and** $\eta = \eta$]]
have $\{\mu,\ \mu\ U_r\ \eta\} \subseteq$ *old* $(\sigma\ k)$ (**is** *?subsetthm*)
proof (*cases k*)
assume $k = 0$
with *assms* σ*-k-prop*
inres-SPEC[*OF res until-propag-on-create-graph*[**where** $\mu = \mu$ **and** $\eta = \eta$]]
show *?subsetthm* **by** *auto*
next
fix *l*
assume $k = Suc\ l$
then have $\{\mu,\ \mu\ U_r\ \eta\}{\subseteq}$*old* $(\sigma\ l) \wedge \eta{\notin}$*old* $(\sigma\ l)$
$\wedge\ \sigma\ l{\in}qs \wedge \sigma\ k{\in}qs$
$\wedge$ *name* $(\sigma\ l){\in}$*incoming* $(\sigma\ k)$
using *step assms run-propag-on-create-graph*[*OF run*] **by** *auto*
with *inres-SPEC*[*OF res until-propag-on-create-graph*[**where** $\mu = \mu$ **and** $\eta = \eta$]]
have $\mu\ U_r\ \eta{\in}$*old* $(\sigma\ k)$ **by** *auto*
with σ*-k-prop*
inres-SPEC[*OF res until-propag-on-create-graph*[**where** $\mu = \mu$ **and** $\eta = \eta$]]
show *?subsetthm* **by** *auto*
qed
with *step* **show** *?sbthm* (*Suc k*) **by** (*metis less-SucE*)
qed
with *assms* **show** *?thesis* **by** *auto*
qed

lemma *L4-2a*:
assumes *ipath gba.E* σ
and $\mu\ U_r\ \eta \in$ *old* $(\sigma\ 0)$
shows $(\forall i.\ \{\mu,\ \mu\ U_r\ \eta\} \subseteq$ *old* $(\sigma\ i) \wedge \eta \notin$ *old* $(\sigma\ i))$

```
      ∨ (∃j. (∀i<j. {μ, μ U_r η} ⊆ old (σ i)) ∧ η ∈ old (σ j))
   (is ?A ∨ ?B)
proof (rule disjCI)
  assume ¬ ?B
  then show ?A
    using assms by (rule-tac L4-2--aux) blast+
qed

lemma L4-2b:
  assumes run: ipath gba.E σ
    and μ U_r η ∈ old (σ 0)
    and ACC: gba.is-acc σ
  shows ∃j. (∀i<j. {μ, μ U_r η} ⊆ old (σ i)) ∧ η ∈ old (σ j)
proof (rule ccontr)
  assume ¬ ?thesis
  then have contr: ∀i. {μ, μ U_r η}⊆old(σ i) ∧ η∉old(σ i)
    using assms L4-2a[of σ μ η] by blast

  define S where S = {q ∈ qs. μ U_r η ∈ old q ⟶ η ∈ old q}

 from assms inres-SPEC[OF res old-propag-on-create-graph] create-gba-from-nodes--ipath
  have μ U_r η ∈ subfrmlsr φ
    by (metis assms(2) subsetD)
  then have S∈gbg-F(create-gba-from-nodes φ qs)
    unfolding S-def create-gba-from-nodes-def by auto
  with ACC have 1: ∃_∞i. σ i ∈ S
    unfolding gba.is-acc-def by blast

  from INFM-EX[OF 1] obtain k where σ k ∈ qs and μ U_r η ∈ old (σ k) ⟶
η ∈ old (σ k)
    unfolding S-def by auto
  moreover have {μ, μ U_r η}⊆old(σ k) ∧ η∉old(σ k)
    using contr by auto
  ultimately show False by auto
qed

lemma L4-2c:
  assumes run: ipath gba.E σ
    and μ R_r η ∈ old (σ 0)
  shows ∀i. η ∈ old (σ i) ∨ (∃j<i. μ ∈ old (σ j))
proof −
  have {η, μ R_r η} ⊆ old (σ i) ∨ (∃j<i. μ ∈ old (σ j)) (is ?thm i) for i
  proof (induct i)
    case 0
    have σ 0∈qs ∧ σ 1∈qs ∧ name (σ 0) ∈ incoming (σ 1)
    using create-gba-from-nodes--ipath assms by auto
    then show ?case
      using assms inres-SPEC[OF res release-propag-on-create-graph, of μ η]
      by auto
```

```
  next
    case (Suc k)
    note ‹?thm k›
    moreover
    { assume {η, μ R_r η} ⊆ old (σ k)
      moreover
      have σ k∈qs ∧ σ (Suc k)∈qs ∧ name (σ k) ∈ incoming (σ (Suc k))
      using create-gba-from-nodes--ipath assms by auto
      ultimately have μ ∈ old (σ k) ∨ μ R_r η ∈ old (σ (Suc k))
        using assms inres-SPEC[OF res release-propag-on-create-graph, of μ η]
        by auto
      moreover
      { assume μ ∈ old (σ k)
        then have ?case by blast }
      moreover
      { assume μ R_r η ∈ old (σ (Suc k))
        moreover
        have σ (Suc k)∈qs ∧ σ (Suc (Suc k))∈qs
          ∧ name (σ (Suc k)) ∈ incoming (σ (Suc (Suc k)))
        using create-gba-from-nodes--ipath assms by auto
        ultimately have ?case
          using assms
            inres-SPEC[OF res release-propag-on-create-graph, of μ η]
          by auto }
      ultimately have ?case by fast }
    moreover
    { assume ∃j<k. μ ∈ old (σ j)
      then have ?case by auto }
    ultimately show ?case by auto
  qed
  then show ?thesis by auto
qed

lemma L4-8':
  assumes ipath gba.E σ (is ?inf-run σ)
    and gba.is-acc σ (is ?gbarel-accept σ)
    and ∀ i. gba.L (σ i) (ξ i) (is ?lgbarel-accept ξ σ)
    and ψ ∈ old (σ 0)
  shows ξ ⊨_r ψ
  using assms
proof (induct ψ arbitrary: σ ξ)
  case True-ltlr
  show ?case by auto
next
  case False-ltlr
  then show ?case
    using inres-SPEC[OF res false-propag-on-create-graph]
      create-gba-from-nodes--ipath
    by (metis)
```

next
case (*Prop-ltlr p*)
then show *?case*
unfolding *create-gba-from-nodes-def* **by** *auto*
next
case (*Nprop-ltlr p*)
then show *?case*
unfolding *create-gba-from-nodes-def* **by** *auto*
next
case (*And-ltlr* μ η)
then show *?case*
using *inres-SPEC*[*OF res and-propag-on-create-graph*, *of* μ η]
create-gba-from-nodes--ipath
by (*metis insert-subset semantics-ltlr.simps*(5))
next
case (*Or-ltlr* μ η)
then have $\mu \in old\ (\sigma\ 0) \vee \eta \in old\ (\sigma\ 0)$
using *inres-SPEC*[*OF res or-propag-on-create-graph*, *of* μ η]
create-gba-from-nodes--ipath
by (*metis* (*full-types*) *Int-empty-left Int-insert-left-if0*)
moreover have $\xi \models_r \mu$ **if** $\mu \in old\ (\sigma\ 0)$
using *Or-ltlr that* **by** *auto*
moreover have $\xi \models_r \eta$ **if** $\eta \in old\ (\sigma\ 0)$
using *Or-ltlr that* **by** *auto*
ultimately show *?case* **by** *auto*
next
case (*Next-ltlr* μ)
with *create-gba-from-nodes--ipath*[*of* σ]
have $\sigma\ 0 \in qs \wedge \sigma\ 1 \in qs \wedge name\ (\sigma\ 0) \in incoming\ (\sigma\ 1)$
by *auto*
with *inres-SPEC*[*OF res next-propag-on-create-graph*, *of* μ] **have** $\mu \in old$ (*suffix 1* σ *0*)
using *Next-ltlr* **by** *auto*
moreover
have *?inf-run* (*suffix 1* σ)
and *?gbarel-accept* (*suffix 1* σ)
and *?lgbarel-accept* (*suffix 1* ξ) (*suffix 1* σ)
using *Next-ltlr create-gba-from-nodes--ipath*
apply −
apply (*metis ipath-suffix*)
apply (*auto simp del*: *suffix-nth*) []
apply *auto*
done
ultimately show *?case* **using** *Next-ltlr* **by** *simp*
next
case (*Until-ltlr* μ η)
then have $\exists j.\ (\forall i{<}j.\ \{\mu, \mu\ U_r\ \eta\} \subseteq old\ (\sigma\ i)) \wedge \eta \in old\ (\sigma\ j)$
using *L4-2b* **by** *auto*
then obtain *j* **where** σ-*pre*: $\forall i{<}j.\ \{\mu, \mu\ U_r\ \eta\} \subseteq old\ (\sigma\ i)$ **and** $\eta \in old$ (*suffix*

```
j σ 0)
    by auto
  moreover
  have ?inf-run (suffix j σ)
    and ?gbarel-accept (suffix j σ)
    and ?lgbarel-accept (suffix j ξ) (suffix j σ)
    unfolding limit-suffix
    using Until-ltlr create-gba-from-nodes--ipath
    apply −
    apply (metis ipath-suffix)
    apply (auto simp del: suffix-nth) []
    apply auto
    done
  ultimately have suffix j ξ ⊨r η
    using Until-ltlr by simp
  moreover {
    fix i
    assume i < j
    have ?inf-run (suffix i σ)
      and ?gbarel-accept (suffix i σ)
      and ?lgbarel-accept (suffix i ξ) (suffix i σ )
      unfolding limit-suffix
      using Until-ltlr create-gba-from-nodes--ipath
      apply −
      apply (metis ipath-suffix)
      apply (auto simp del: suffix-nth) []
      apply auto
      done
    moreover have μ∈old (suffix i σ 0)
      using σ-pre ‹i<j› by auto
    ultimately have suffix i ξ ⊨r μ using Until-ltlr by simp
  }
  ultimately show ?case by auto
next
  case (Release-ltlr μ η)
  { fix i
    assume η ∈ old (σ i) ∨ (∃j<i. μ ∈ old (σ j))
    moreover
    {
      assume ∗: η ∈ old (σ i)
      have ?inf-run (suffix i σ)
        and ?gbarel-accept (suffix i σ)
        and ?lgbarel-accept (suffix i ξ) (suffix i σ )
        unfolding limit-suffix
        using Release-ltlr create-gba-from-nodes--ipath
        apply −
        apply (metis ipath-suffix)
        apply (auto simp del: suffix-nth) []
        apply auto
```

```
      done
    with ∗ have suffix i ξ ⊨r η
      using Release-ltlr by auto
  }
  moreover
  {
    assume ∃j<i. μ ∈ old (σ j)
    then obtain j where j<i and μ ∈ old (σ j) by auto
    moreover
    have ?inf-run (suffix j σ)
      and ?gbarel-accept (suffix j σ)
      and ?lgbarel-accept (suffix j ξ) (suffix j σ ) unfolding limit-suffix
      using Release-ltlr create-gba-from-nodes--ipath
      apply −
      apply (metis ipath-suffix)
      apply (auto simp del: suffix-nth) []
      apply auto
      done
    ultimately have suffix j ξ ⊨r μ
      using Release-ltlr by auto
    then have ∃j<i. suffix j ξ ⊨r μ
      using ‹j<i› by auto
  }
  ultimately have suffix i ξ ⊨r η ∨ (∃j<i. suffix j ξ ⊨r μ) by auto
 }
 then show ?case using Release-ltlr L4-2c by auto
qed


lemma L4-8:
 assumes gba.is-acc-run σ
   and ∀ i. gba.L (σ i) (ξ i)
   and ψ ∈ old (σ 0)
 shows ξ ⊨r ψ
 using assms
 unfolding gba.is-acc-run-def gba.is-run-def
 using L4-8′ by blast

lemma expand-expand-init-propag:
 assumes (∀ nm∈incoming n′. nm < name n′)
         ∧ name n′ ≤ name (fst n-ns)
         ∧ (incoming n′ ∩ incoming (fst n-ns) ≠ {}
            ⟶ new n′ ⊆ old (fst n-ns) ∪ new (fst n-ns))
          (is ?Q n-ns)
 and ∀ nd∈snd n-ns. ∀ nm∈incoming n′. nm∈incoming nd ⟶ new n′ ⊆ old nd
   (is ?P (snd n-ns))
 shows expand n-ns ≤ SPEC (λr. name n′≤fst r ∧ ?P (snd r))
 using assms
proof (rule-tac expand-rec-rule[where Φ=λx. ?Q x ∧ ?P (snd x)],
```

```
    simp,
    intro refine-vcg, goal-cases)
  case prems: (1 f x n ns)
  then have goal-assms: new n = {} ∧ ?Q (n, ns) ∧ ?P ns by simp
  { fix nd nm
   assume nd∈upd-incoming n ns and nm∈incoming n' and nm∈incoming nd
   { assume nd∈ns
    with goal-assms ‹nm∈incoming n'› ‹nm∈incoming nd› have new n' ⊆ old nd
      by auto }
   moreover
   { assume nd∉ns
    with upd-incoming--elem[OF ‹nd∈upd-incoming n ns›]
    obtain nd' where nd'∈ns
                and nd-eq: nd = nd'(|incoming := incoming n ∪ incoming nd'|)
                and old-next-eq: old nd' = old n ∧ next nd' = next n by auto
    { assume nm∈incoming nd'
     with goal-assms ‹nm∈incoming n'› ‹nd'∈ns› have new n' ⊆ old nd
      unfolding nd-eq by auto }
    moreover
    { assume nm∈incoming n
     with nd-eq old-next-eq goal-assms ‹nm∈incoming n'›
     have new n' ⊆ old nd
      by auto }
    ultimately have new n' ⊆ old nd using ‹nm∈incoming nd› nd-eq by auto }
   ultimately have new n' ⊆ old nd by fast }
  with prems show ?case by auto
next
  case prems: (2 f x n ns)
  then have goal-assms: new n = {} ∧ ?Q (n, ns) ∧ ?P ns and step: ⋀x. ?Q x
∧ ?P (snd x)
     ⟹ f x ≤ SPEC (λr. name n' ≤ fst r ∧ ?P (snd r))
   by simp-all
  then show ?case
  proof (rule-tac step, goal-cases)
   case prems': 1
   have expand-new-name-less: name n < expand-new-name (name n)
    by auto
   moreover have name n ∉ incoming n'
    using goal-assms by auto
   ultimately show ?case using prems' by auto
  qed
next
  case 3 then show ?case by auto
next
  case prems: (4 f x n ns)
  then have step: ⋀x. ?Q x ∧ ?P (snd x) ⟹ f x ≤ SPEC (λr. name n' ≤ fst r
∧ ?P (snd r))
   by simp
  from prems show ?case by (rule-tac step) auto
```

next
case *prems*: (*5 f x n ns*)
then have *step*: $\bigwedge x.\ ?Q\ x \wedge\ ?P\ (snd\ x) \Longrightarrow f\ x \leq SPEC\ (\lambda r.\ name\ n' \leq fst\ r \wedge\ ?P\ (snd\ r))$
by *simp*
from *prems* **show** *?case* **by** (*rule-tac step*) *auto*
next
case *6* **then show** *?case* **by** *auto*
next
case *prems*: (*7 f x n ns*)
then have *step*: $\bigwedge x.\ ?Q\ x \wedge\ ?P\ (snd\ x) \Longrightarrow f\ x \leq SPEC\ (\lambda r.\ name\ n' \leq fst\ r \wedge\ ?P\ (snd\ r))$
by *simp*
from *prems* **show** *?case* **by** (*rule-tac step*) *auto*
next
case *prems*: (*8 f x n ns* ψ)
then have *goal-assms*: $\psi \in new\ n$
$\wedge \neg\ (\exists\, q.\ \psi = prop_r(q) \vee \psi = nprop_r(q))$
$\wedge\ \psi \neq true_r \wedge \psi \neq false_r$
$\wedge \neg\ (\exists\, \nu\ \mu.\ \psi = \nu\ and_r\ \mu \vee \psi = X_r\ \nu)$
$\wedge\ (\exists\, \nu\ \mu.\ \psi = \nu\ or_r\ \mu \vee \psi = \nu\ U_r\ \mu \vee \psi = \nu\ R_r\ \mu)$
by (*cases* ψ) *auto*
from *prems* **have** *QP*: *?Q* (*n*, *ns*) $\wedge$ *?P ns* **and**
step: $\bigwedge x.\ ?Q\ x \wedge\ ?P\ (snd\ x) \Longrightarrow f\ x \leq SPEC\ (\lambda r.\ name\ n' \leq fst\ r \wedge\ ?P\ (snd\ r))$
by *simp-all*
show *?case*
using *goal-assms QP* **unfolding** *case-prod-unfold* ‹$x = (n,\ ns)$›
proof (*rule-tac SPEC-rule-nested2*, *goal-cases*)
case *1*
then show *?case* **by** (*rule-tac step*) *auto*
next
case *2*
then show *?case* **by** (*rule-tac step*) *auto*
qed
qed

lemma *expand-init-propag-on-create-graph*:
create-graph $\varphi \leq SPEC(\lambda nds.\ \forall\, nd{\in}nds.\ expand\text{-}init{\in}incoming\ nd \longrightarrow \varphi \in old\ nd)$
unfolding *create-graph-def*
by (*intro refine-vcg*, *rule-tac order-trans*,
rule-tac expand-expand-init-propag[**where**
$n' = (\!|\ name = expand\text{-}new\text{-}name\ expand\text{-}init,$
$incoming = \{expand\text{-}init\},$
$new = \{\varphi\},$
$old = \{\},$
$next = \{\}\ |\!)$]) (*auto simp add*:*expand-new-name-expand-init*)

```
lemma L4-6:
  assumes q∈gba.V0
  shows φ∈old q
  using assms inres-SPEC[OF res expand-init-propag-on-create-graph]
  unfolding create-gba-from-nodes-def by auto

lemma L4-9:
  assumes acc: gba.accept ξ
  shows ξ ⊨r φ
proof −
  from acc obtain σ where accept: gba.is-acc-run σ ∧ (∀ i. gba.L (σ i) (ξ i))
    unfolding gba.accept-def by auto
  then have σ 0∈gba.V0
    unfolding gba.is-acc-run-def gba.is-run-def by simp
  with L4-6 have φ∈old (σ 0) by auto
  with L4-8 accept show ?thesis by auto
qed

lemma L4-10:
  assumes ξ ⊨r φ
  shows gba.accept ξ
proof −
  define σ where σ = auto-run ξ qs
  let ?G = create-graph φ

  have σ-prop-0: (σ 0)∈qs ∧ expand-init∈incoming(σ 0) ∧ auto-run-j 0 ξ (σ 0)
    (is ?sbthm)
  using inres-SPEC[OF res L4-7[OF ‹ξ ⊨r φ›]]
  unfolding σ-def auto-run.simps by (rule-tac someI-ex, simp) blast

  have σ-valid: ∀ j. σ j ∈ qs ∧ auto-run-j j ξ (σ j) (is ∀ j. ?σ-valid j)
  proof
    fix j
    show ?σ-valid j
    proof(induct j)
      from σ-prop-0 show ?σ-valid 0 by fast
    next
      fix k
      assume goal-assms: σ k ∈ qs ∧ auto-run-j k ξ (σ k)
      with inres-SPEC[OF res L4-5, of suffix k ξ]
      have sbthm: ∃ q'. q'∈qs ∧ name (σ k)∈incoming q' ∧ auto-run-j (Suc k) ξ q'
        (is ∃ q'. ?sbthm q')
        by auto
      have ?sbthm (σ (Suc k)) using someI-ex[OF sbthm]
      unfolding σ-def auto-run.simps by blast
      then show ?σ-valid (Suc k) by fast
    qed
  qed
```

have σ-*prop-Suc*: $\bigwedge k.\ \sigma\ k \in qs \wedge \sigma\ (Suc\ k) \in qs$
$\wedge$ *name* $(\sigma\ k) \in$ *incoming* $(\sigma\ (Suc\ k))$
$\wedge$ *auto-run-j* $(Suc\ k)\ \xi\ (\sigma\ (Suc\ k))$
proof $-$
fix k
from σ-*valid* **have** $\sigma\ k \in qs$ **and** *auto-run-j* $k\ \xi\ (\sigma\ k)$ **by** *blast*+
with *inres-SPEC*[*OF res L4-5*, *of suffix* $k\ \xi$]
have *sbthm*: $\exists\, q'.\ q' \in qs \wedge$ *name* $(\sigma\ k) \in$ *incoming* q'
$\wedge$ *auto-run-j* $(Suc\ k)\ \xi\ q'$ (**is** $\exists\, q'.$ *?sbthm* q')
by *auto*
show $\sigma\ k \in qs \wedge$ *?sbthm* $(\sigma\ (Suc\ k))$ **using** $\langle \sigma\ k \in qs \rangle$ *someI-ex*[*OF sbthm*]
unfolding σ-*def auto-run.simps* **by** *blast*
qed

have σ-*vnaccpt*:
$\forall\, k\ \mu\ \eta.\ \mu\ U_r\ \eta \in old\ (\sigma\ k) \longrightarrow \neg\ (\forall\, i.\ \{\mu,\ \mu\ U_r\ \eta\} \subseteq old\ (\sigma\ (k+i)) \wedge \eta \notin old\ (\sigma\ (k+i)))$
proof *clarify*
fix $k\ \mu\ \eta$
assume *U-in*: $\mu\ U_r\ \eta \in old\ (\sigma\ k)$
and *cntr-prm*: $\forall\, i.\ \{\mu,\ \mu\ U_r\ \eta\} \subseteq old\ (\sigma\ (k+i)) \wedge \eta \notin old\ (\sigma\ (k+i))$
have *suffix* $k\ \xi \models_r \mu\ U_r\ \eta$
using *U-in* σ-*valid* **by** *force*
then obtain i **where** *suffix* $(k+i)\ \xi \models_r \eta$ **and** $\forall\, j{<}i.$ *suffix* $(k+j)\ \xi \models_r \mu$
by *auto*
moreover have $\mu\ U_r\ \eta \in old\ (\sigma\ (k+i)) \wedge \eta \notin old\ (\sigma\ (k+i))$
using *cntr-prm* **by** *auto*
ultimately show *False*
using σ-*valid* **by** *force*
qed

have σ-*exec*: *gba.is-run* σ
using σ-*prop-0* σ-*prop-Suc* σ-*valid*
unfolding *gba.is-run-def ipath-def*
unfolding *create-gba-from-nodes-def*
by *auto*
moreover
have σ-*vaccpt*:
$\forall\, k\ \mu\ \eta.\ \mu\ U_r\ \eta \in old\ (\sigma\ k) \longrightarrow$
$(\exists\, j.\ (\forall\, i{<}j.\ \{\mu,\ \mu\ U_r\ \eta\} \subseteq old\ (\sigma\ (k+i))) \wedge \eta \in old\ (\sigma\ (k+j)))$
proof(*clarify*)
fix $k\ \mu\ \eta$
assume *U-in*: $\mu\ U_r\ \eta \in old\ (\sigma\ k)$
then have $\neg\ (\forall\, i.\ \{\mu,\ \mu\ U_r\ \eta\} \subseteq old\ (suffix\ k\ \sigma\ i) \wedge \eta \notin old\ (suffix\ k\ \sigma\ i))$
using σ-*vnaccpt*[*THEN allE*, *of k*] **by** *auto*
moreover have *suffix* $k\ \sigma\ 0 \in qs$ **using** σ-*valid* **by** *auto*
ultimately show $\exists\, j.\ (\forall\, i{<}j.\ \{\mu,\ \mu\ U_r\ \eta\} \subseteq old\ (\sigma\ (k+i))) \wedge \eta \in old\ (\sigma\ (k+j))$
apply $-$
apply (*rule make-pos-rule'*[*OF L4-2a*])

```
    apply (fold suffix-def)
    apply (rule ipath-suffix)
    using σ-exec[unfolded gba.is-run-def]
    apply simp
    using U-in apply simp
    apply simp
    done
qed

have gba.is-acc σ
  unfolding gba.is-acc-def
proof
  fix S
  assume S∈gba.F
  then obtain μ η where S-eq: S = {q ∈ qs. μ U_r η ∈ old q ⟶ η ∈ old q}
    and μ U_r η ∈ subfrmlsr φ
    by (auto simp add: create-gba-from-nodes-def)
  have range-subset: range σ ⊆ qs
  proof
    fix q
    assume q∈range σ
    with full-SetCompr-eq[of σ] obtain k where q = σ k by auto
    then show q ∈ qs using σ-valid by auto
  qed
  with limit-nonempty[of σ]
       limit-in-range[of σ]
       finite-subset[OF range-subset]
       inres-SPEC[OF res create-graph-finite]
  obtain q where q-in-limit: q ∈ limit σ and q-is-node: q∈qs
    by auto
  show ∃_∞i. σ i ∈ S
  proof (cases μ U_r η ∈ old q)
    case False
    with S-eq q-in-limit q-is-node
    show ?thesis
      by (auto simp: limit-iff-frequent intro: INFM-mono)
  next
    case True
    obtain k where q-eq: q = σ k using q-in-limit
      unfolding limit-iff-frequent by (metis (lifting, full-types) INFM-nat-le)
    have ∃_∞ k. η ∈ old (σ k)
      unfolding INFM-nat
    proof (rule ccontr)
      assume ¬ (∀ m. ∃ n>m. η ∈ old (σ n))
      then obtain m where ∀ n>m. η ∉ old (σ n) by auto
      moreover
      from q-eq q-in-limit limit-iff-frequent[of q σ] INFM-nat[of λn. σ n = q]
      obtain n where m<n and σn-eq: σ n = σ k by auto
      moreover
```

```
      obtain j where η ∈ old (σ (n+j))
        using σ-vaccpt ‹μ U_r η ∈ old q› unfolding q-eq by (fold σn-eq) force
      ultimately show False by auto
    qed
    then have ∃_∞ k. σ k ∈ qs ∧ η ∈ old (σ k)
      using σ-valid by (auto intro: INF-mono)
    then show ∃_∞ k. σ k ∈ S
      unfolding S-eq by (rule INFM-mono) simp
  qed
 qed
 moreover have gba.L (σ i) (ξ i) for i
 proof −
  from σ-valid have [simp]: σ i ∈ qs by auto
  have ∀ψ∈old (σ i). suffix i ξ ⊨_r ψ
    using σ-valid by auto
  then show ?thesis
    unfolding create-gba-from-nodes-def by auto
 qed
 ultimately show ?thesis
  unfolding gba.accept-def gba.is-acc-run-def by blast
qed

end
end

lemma create-graph--name-ident: create-graph φ ≤ SPEC (λnds. ∀ q∈nds. ∃!q'∈nds.
name q = name q')
 unfolding create-graph-def
 apply (refine-vcg expand-name-propag--name-ident)
 by (auto simp add:expand-new-name-expand-init)

corollary create-graph--name-inj: create-graph φ ≤ SPEC (λnds. inj-on name
nds)
 by (rule order-trans[OF create-graph--name-ident]) (auto intro: inj-onI)


definition
 create-gba φ
  ≡ do { nds ← create-graph_T φ;
       RETURN (create-gba-from-nodes φ nds) }

lemma create-graph-precond: create-graph φ
 ≤ SPEC (create-gba-from-nodes-precond φ)
 apply (clarsimp simp: pw-le-iff create-graph-nofail)
 apply unfold-locales
 apply simp
 done

lemma create-gba--invar: create-gba φ ≤ SPEC gba
```

unfolding *create-gba-def create-graph-eq-create-graph$_T$[symmetric]*
apply (*refine-rcg refine-vcg order-trans*[*OF create-graph-precond*])
by (*rule create-gba-from-nodes-precond.create-gba-from-nodes--invar*)

lemma *create-gba-acc*:
shows *create-gba* $\varphi \leq SPEC(\lambda\mathcal{A}.\ \forall\,\xi.\ gba.accept\ \mathcal{A}\ \xi \longleftrightarrow \xi \models_r \varphi)$
unfolding *create-gba-def create-graph-eq-create-graph$_T$*[*symmetric*]
apply (*refine-rcg refine-vcg order-trans*[*OF create-graph-precond*])
using *create-gba-from-nodes-precond.L4-9*
using *create-gba-from-nodes-precond.L4-10*
by *blast*

lemma *create-gba--name-inj*:
shows *create-gba* $\varphi \leq SPEC(\lambda\mathcal{A}.\ (inj\text{-}on\ name\ (g\text{-}V\ \mathcal{A})))$
unfolding *create-gba-def create-graph-eq-create-graph$_T$*[*symmetric*]
apply (*refine-rcg refine-vcg order-trans*[*OF create-graph--name-inj*])
apply (*auto simp*: *create-gba-from-nodes-def*)
done

lemma *create-gba--fin*:
shows *create-gba* $\varphi \leq SPEC(\lambda\mathcal{A}.\ (finite\ (g\text{-}V\ \mathcal{A})))$
unfolding *create-gba-def create-graph-eq-create-graph$_T$*[*symmetric*]
apply (*refine-rcg refine-vcg order-trans*[*OF create-graph-finite*])
apply (*auto simp*: *create-gba-from-nodes-def*)
done

lemma *create-graph-old-finite*:
create-graph $\varphi \leq SPEC\ (\lambda nds.\ \forall\, nd{\in}nds.\ finite\ (old\ nd))$
proof −
show *?thesis*
unfolding *create-graph-def create-graph-eq-create-graph$_T$*[*symmetric*]
unfolding *expand-def*
apply (*intro refine-vcg*)
apply (*rule-tac order-trans*)
apply (*rule-tac REC-le-RECT*)
apply (*fold expand$_T$-def*)
apply (*rule-tac order-trans*[*OF expand-term-prop*])
apply *auto*[*1*]
apply (*rule-tac SPEC-rule*)
apply *auto*
by (*metis infinite-super subfrmlsr-finite*)
qed

lemma *create-gba--old-fin*:
shows *create-gba* $\varphi \leq SPEC(\lambda\mathcal{A}.\ \forall\, nd{\in}g\text{-}V\ \mathcal{A}.\ finite\ (old\ nd))$
unfolding *create-gba-def create-graph-eq-create-graph$_T$*[*symmetric*]
apply (*refine-rcg refine-vcg order-trans*[*OF create-graph-old-finite*])
apply (*simp add*: *create-gba-from-nodes-def*)
done

lemma *create-gba--incoming-exists*:
shows *create-gba* φ
$\leq$ *SPEC*($\lambda\mathcal{A}$. $\forall$ *nd*$\in$*g-V* $\mathcal{A}$. *incoming nd* $\subseteq$ *insert expand-init* (*name* ‘ (*g-V* $\mathcal{A}$)))
unfolding *create-gba-def create-graph-eq-create-graph*$_T$[*symmetric*]
apply (*refine-rcg refine-vcg order-trans*[*OF create-graph--incoming-name-exist*])
apply (*auto simp add*: *create-gba-from-nodes-def*)
done

lemma *create-gba--no-init*:
shows *create-gba* $\varphi \leq$ *SPEC*($\lambda\mathcal{A}$. *expand-init* $\notin$ *name* ‘ (*g-V* $\mathcal{A}$))
unfolding *create-gba-def create-graph-eq-create-graph*$_T$[*symmetric*]
apply (*refine-rcg refine-vcg order-trans*[*OF create-graph--incoming-name-exist*])
apply (*auto simp add*: *create-gba-from-nodes-def*)
done

definition *nds-invars nds* $\equiv$
inj-on name nds
$\wedge$ *finite nds*
$\wedge$ *expand-init* $\notin$ *name*‘*nds*
$\wedge$ ($\forall$ *nd*$\in$*nds*.
finite (*old nd*)
$\wedge$ *incoming nd* $\subseteq$ *insert expand-init* (*name* ‘ *nds*))

lemma *create-gba-nds-invars*: *create-gba* $\varphi \leq$ *SPEC* ($\lambda\mathcal{A}$. *nds-invars* (*g-V* $\mathcal{A}$))
using *create-gba--name-inj*[*of* φ] *create-gba--fin*[*of* φ]
create-gba--old-fin[*of* φ] *create-gba--incoming-exists*[*of* φ]
create-gba--no-init[*of* φ]
unfolding *nds-invars-def*
by (*simp add*: *pw-le-iff*)

theorem *T4-1*:
create-gba $\varphi \leq$ *SPEC*(
$\lambda\mathcal{A}$. *gba* $\mathcal{A}$
$\wedge$ *finite* (*g-V* $\mathcal{A}$)
$\wedge$ ($\forall\,\xi$. *gba.accept* $\mathcal{A}$ $\xi \longleftrightarrow \xi \models_r \varphi$)
$\wedge$ (*nds-invars* (*g-V* $\mathcal{A}$)))
using *create-gba--invar create-gba--fin create-gba-acc create-gba-nds-invars*
apply (*simp add*: *pw-le-iff*)
apply *blast*
done

definition *create-name-gba* $\varphi \equiv$ *do* {
$G \leftarrow$ *create-gba* φ;
ASSERT (*nds-invars* (*g-V G*));
RETURN (*gba-rename name G*)
}

theorem *create-name-gba-correct*:
create-name-gba $\varphi \leq$ *SPEC*($\lambda\mathcal{A}$. *gba* $\mathcal{A} \wedge$ *finite* (*g-V* $\mathcal{A}$) $\wedge$ ($\forall\xi$. *gba.accept* $\mathcal{A}$ ξ $\longleftrightarrow$ $\xi \models_r \varphi$))
unfolding *create-name-gba-def*
apply (*refine-rcg refine-vcg order-trans*[*OF T4-1*])
apply (*simp-all add*: *nds-invars-def gba-rename-correct*)
done

definition *create-name-igba* :: *'a*::*linorder ltlr* $\Rightarrow$ - **where**
create-name-igba $\varphi \equiv$ *do* {
A $\leftarrow$ *create-name-gba* φ;
A' $\leftarrow$ *gba-to-idx A*;
stat-set-data-nres (*card* (*g-V A*)) (*card* (*g-V0 A'*)) (*igbg-num-acc A'*);
RETURN A'
}

lemma *create-name-igba-correct*: *create-name-igba* $\varphi \leq$ *SPEC* (λG.
igba G $\wedge$ *finite* (*g-V G*) $\wedge$ ($\forall\xi$. *igba.accept G* ξ $\longleftrightarrow$ $\xi \models_r \varphi$))
unfolding *create-name-igba-def*
apply (*refine-rcg*
order-trans[*OF create-name-gba-correct*]
order-trans[*OF gba.gba-to-idx-ext-correct*]
refine-vcg)
apply *clarsimp-all*
proof −
fix *G* :: (*nat*, *'a set*) *gba-rec*
fix *A* :: *nat set*
assume *1*: *gba G*
assume *2*: *finite* (*g-V G*) *A* $\in$ *gbg-F G*
interpret *gba G* **using** *1* **.**
show *finite A* **using** *finite-V-Fe 2* **.**
qed

context
notes [*refine-vcg*] = *order-trans*[*OF create-name-gba-correct*]
begin

lemma *create-name-igba* $\varphi \leq$ *SPEC* (λG. *igba G* $\wedge$ ($\forall\xi$. *igba.accept G* ξ $\longleftrightarrow$ $\xi \models_r \varphi$))
unfolding *create-name-igba-def*
proof (*refine-rcg refine-vcg*, *clarsimp-all*)
fix *G* :: (*nat*, *'a set*) *gba-rec*
assume *gba G*
then interpret *gba G* **.**
note [*refine-vcg*] = *order-trans*[*OF gba-to-idx-ext-correct*]

assume $\forall\xi$. *gba.accept G* $\xi = \xi \models_r \varphi$ *finite* (*g-V G*)
then show *gba-to-idx G* $\leq$ *SPEC* ($\lambda G'$. *igba G'* $\wedge$ ($\forall\xi$. *igba.accept G'* $\xi = \xi \models_r$

φ))
  by (*refine-rcg refine-vcg*) (*auto intro*: *finite-V-Fe*)
qed

end

end

3 Refinement to Efficient Code

theory *LTL-to-GBA-impl*
imports
  LTL-to-GBA
  Deriving.Compare-Instances
  CAVA-Automata.Automata-Impl
  CAVA-Base.CAVA-Code-Target
begin

3.1 Parametricity Setup Boilerplate

3.1.1 LTL Formulas

derive *linorder ltlr*

inductive-set *ltlr-rel* **for** R **where**
  (*True-ltlr*, *True-ltlr*) $\in$ *ltlr-rel R*
| (*False-ltlr*, *False-ltlr*) $\in$ *ltlr-rel R*
| (*a*,*a'*)$\in R \Longrightarrow$ (*Prop-ltlr a*,*Prop-ltlr a'*) $\in$ *ltlr-rel R*
| (*a*,*a'*)$\in R \Longrightarrow$ (*Nprop-ltlr a*,*Nprop-ltlr a'*) $\in$ *ltlr-rel R*
| $[\![$(*a*,*a'*)$\in$*ltlr-rel R*; (*b*,*b'*)$\in$*ltlr-rel R*$]\!]$
  $\Longrightarrow$ (*And-ltlr a b*,*And-ltlr a' b'*) $\in$ *ltlr-rel R*
| $[\![$(*a*,*a'*)$\in$*ltlr-rel R*; (*b*,*b'*)$\in$*ltlr-rel R*$]\!]$
  $\Longrightarrow$ (*Or-ltlr a b*,*Or-ltlr a' b'*) $\in$ *ltlr-rel R*
| $[\![$(*a*,*a'*)$\in$*ltlr-rel R*$]\!] \Longrightarrow$ (*Next-ltlr a*,*Next-ltlr a'*) $\in$ *ltlr-rel R*
| $[\![$(*a*,*a'*)$\in$*ltlr-rel R*; (*b*,*b'*)$\in$*ltlr-rel R*$]\!]$
  $\Longrightarrow$ (*Until-ltlr a b*,*Until-ltlr a' b'*) $\in$ *ltlr-rel R*
| $[\![$(*a*,*a'*)$\in$*ltlr-rel R*; (*b*,*b'*)$\in$*ltlr-rel R*$]\!]$
  $\Longrightarrow$ (*Release-ltlr a b*,*Release-ltlr a' b'*) $\in$ *ltlr-rel R*

lemmas *ltlr-rel-induct*[*induct set*]
  = *ltlr-rel.induct*[*simplified relAPP-def*[*of ltlr-rel*, *symmetric*]]
lemmas *ltlr-rel-cases*[*cases set*]
  = *ltlr-rel.cases*[*simplified relAPP-def*[*of ltlr-rel*, *symmetric*]]
lemmas *ltlr-rel-intros*
  = *ltlr-rel.intros*[*simplified relAPP-def*[*of ltlr-rel*, *symmetric*]]

inductive-simps *ltlr-rel-left-simps*[*simplified relAPP-def*[*of ltlr-rel*, *symmetric*]]:
  (*True-ltlr*,*z*) $\in$ *ltlr-rel R*
  (*False-ltlr*,*z*) $\in$ *ltlr-rel R*

(*Prop-ltlr p, z*) ∈ *ltlr-rel R*
(*Nprop-ltlr p, z*) ∈ *ltlr-rel R*
(*And-ltlr a b, z*) ∈ *ltlr-rel R*
(*Or-ltlr a b, z*) ∈ *ltlr-rel R*
(*Next-ltlr a, z*) ∈ *ltlr-rel R*
(*Until-ltlr a b, z*) ∈ *ltlr-rel R*
(*Release-ltlr a b, z*) ∈ *ltlr-rel R*

lemma *ltlr-rel-sv*[*relator-props*]:
assumes *SV*: *single-valued R*
shows *single-valued* (⟨*R*⟩*ltlr-rel*)
proof (*intro single-valuedI allI impI*)
fix *x y z*
assume (*x, y*) ∈ ⟨*R*⟩*ltlr-rel* (*x, z*) ∈ ⟨*R*⟩*ltlr-rel*
then show *y*=*z*
apply (*induction arbitrary*: *z*)
apply (*simp* (*no-asm-use*) *only*: *ltlr-rel-left-simps*
| *blast intro*: *single-valuedD*[*OF SV*])+
done
qed

lemma *ltlr-rel-id*[*relator-props*]: ⟦ *R* = *Id* ⟧ ⟹ ⟨*R*⟩*ltlr-rel* = *Id*
proof (*intro equalityI subsetI*, *clarsimp-all*)
fix *a b*
assume (*a*,*b*)∈⟨*Id*⟩*ltlr-rel*
then show *a*=*b*
by *induction auto*
next
fix *a*
show (*a*,*a*)∈⟨*Id*⟩*ltlr-rel*
by (*induction a*) (*auto intro*: *ltlr-rel-intros*)
qed

lemma *ltlr-rel-id-simp*[*simp*]: ⟨*Id*⟩*ltlr-rel* = *Id* **by** (*rule ltlr-rel-id*) *simp*

consts *i-ltlr* :: *interface* ⇒ *interface*
lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of ltlr-rel i-ltlr*]

thm *ltlr-rel-intros*[*no-vars*]

lemma *ltlr-con-param*[*param*, *autoref-rules*]:
(*True-ltlr*, *True-ltlr*) ∈ ⟨*R*⟩*ltlr-rel*
(*False-ltlr*, *False-ltlr*) ∈ ⟨*R*⟩*ltlr-rel*
(*Prop-ltlr*, *Prop-ltlr*) ∈ *R* → ⟨*R*⟩*ltlr-rel*
(*Nprop-ltlr*, *Nprop-ltlr*) ∈ *R* → ⟨*R*⟩*ltlr-rel*
(*And-ltlr*, *And-ltlr*) ∈ ⟨*R*⟩*ltlr-rel* → ⟨*R*⟩*ltlr-rel* → ⟨*R*⟩*ltlr-rel*
(*Or-ltlr*, *Or-ltlr*) ∈ ⟨*R*⟩*ltlr-rel* → ⟨*R*⟩*ltlr-rel* → ⟨*R*⟩*ltlr-rel*
(*Next-ltlr*, *Next-ltlr*) ∈ ⟨*R*⟩*ltlr-rel* → ⟨*R*⟩*ltlr-rel*
(*Until-ltlr*, *Until-ltlr*) ∈ ⟨*R*⟩*ltlr-rel* → ⟨*R*⟩*ltlr-rel* → ⟨*R*⟩*ltlr-rel*

(*Release-ltlr*, *Release-ltlr*) ∈ ⟨*R*⟩*ltlr-rel* → ⟨*R*⟩*ltlr-rel* → ⟨*R*⟩*ltlr-rel*
by (*auto intro*: *ltlr-rel-intros*)

lemma *case-ltlr-param*[*param*, *autoref-rules*]:
(*case-ltlr*,*case-ltlr*) ∈ *Rv* → *Rv* → (*R* → *Rv*)
→ (*R* → *Rv*)
→ (⟨*R*⟩*ltlr-rel* → ⟨*R*⟩*ltlr-rel* → *Rv*)
→ (⟨*R*⟩*ltlr-rel* → ⟨*R*⟩*ltlr-rel* → *Rv*)
→ (⟨*R*⟩*ltlr-rel* → *Rv*)
→ (⟨*R*⟩*ltlr-rel* → ⟨*R*⟩*ltlr-rel* → *Rv*)
→ (⟨*R*⟩*ltlr-rel* → ⟨*R*⟩*ltlr-rel* → *Rv*) → ⟨*R*⟩*ltlr-rel* → *Rv*
apply (*clarsimp*)
apply (*case-tac ai*, *simp-all add*: *ltlr-rel-left-simps*)
apply (*clarsimp-all*)
apply *parametricity*+
done

lemma *rec-ltlr-param*[*param*, *autoref-rules*]:
(*rec-ltlr*,*rec-ltlr*) ∈ *Rv* → *Rv* → (*R* → *Rv*)
→ (*R* → *Rv*)
→ (⟨*R*⟩*ltlr-rel* → ⟨*R*⟩*ltlr-rel* → *Rv* → *Rv* → *Rv*)
→ (⟨*R*⟩*ltlr-rel* → ⟨*R*⟩*ltlr-rel* → *Rv* → *Rv* → *Rv*)
→ (⟨*R*⟩*ltlr-rel* → *Rv* → *Rv*)
→ (⟨*R*⟩*ltlr-rel* → ⟨*R*⟩*ltlr-rel* → *Rv* → *Rv* → *Rv*)
→ (⟨*R*⟩*ltlr-rel* → ⟨*R*⟩*ltlr-rel* → *Rv* → *Rv* → *Rv*)
→ ⟨*R*⟩*ltlr-rel* → *Rv*
proof (*clarsimp*, *goal-cases*)
case *prems*: *1*
from *prems*(*10*)
show *?case*
apply (*induction*)
using *prems*(*1*−*9*)
apply *simp-all*
apply *parametricity*+
done
qed

lemma *case-ltlr-mono*[*refine-mono*]:
assumes φ = *True-ltlr* ⟹ *a*≤*a*′
assumes φ = *False-ltlr* ⟹ *b*≤*b*′
assumes ⋀*p*. φ = *Prop-ltlr p* ⟹ *c p*≤*c*′ *p*
assumes ⋀*p*. φ = *Nprop-ltlr p* ⟹ *d p*≤*d*′ *p*
assumes ⋀μ ν. φ = *And-ltlr* μ ν ⟹ *e* μ ν≤*e*′ μ ν
assumes ⋀μ ν. φ = *Or-ltlr* μ ν ⟹ *f* μ ν≤*f*′ μ ν
assumes ⋀μ. φ = *Next-ltlr* μ ⟹ *g* μ≤*g*′ μ
assumes ⋀μ ν. φ = *Until-ltlr* μ ν ⟹ *h* μ ν≤*h*′ μ ν
assumes ⋀μ ν. φ = *Release-ltlr* μ ν ⟹ *i* μ ν≤*i*′ μ ν
shows *case-ltlr a b c d e f g h i* φ ≤ *case-ltlr a*′ *b*′ *c*′ *d*′ *e*′ *f*′ *g*′ *h*′ *i*′ φ
using *assms*

```
  apply (cases φ)
  apply simp-all
  done


primrec ltlr-eq where
  ltlr-eq eq True-ltlr f ⟷ (case f of True-ltlr ⇒ True | - ⇒ False)
| ltlr-eq eq False-ltlr f ⟷ (case f of False-ltlr ⇒ True | - ⇒ False)
| ltlr-eq eq (Prop-ltlr p) f ⟷ (case f of Prop-ltlr p' ⇒ eq p p' | - ⇒ False)
| ltlr-eq eq (Nprop-ltlr p) f ⟷ (case f of Nprop-ltlr p' ⇒ eq p p' | - ⇒ False)
| ltlr-eq eq (And-ltlr μ ν) f
  ⟷ (case f of And-ltlr μ' ν' ⇒ ltlr-eq eq μ μ' ∧ ltlr-eq eq ν ν' | - ⇒ False)
| ltlr-eq eq (Or-ltlr μ ν) f
  ⟷ (case f of Or-ltlr μ' ν' ⇒ ltlr-eq eq μ μ' ∧ ltlr-eq eq ν ν' | - ⇒ False)
| ltlr-eq eq (Next-ltlr φ) f
  ⟷ (case f of Next-ltlr φ' ⇒ ltlr-eq eq φ φ' | - ⇒ False)
| ltlr-eq eq (Until-ltlr μ ν) f
  ⟷ (case f of Until-ltlr μ' ν' ⇒ ltlr-eq eq μ μ' ∧ ltlr-eq eq ν ν' | - ⇒ False)
| ltlr-eq eq (Release-ltlr μ ν) f
  ⟷ (case f of
    Release-ltlr μ' ν' ⇒ ltlr-eq eq μ μ' ∧ ltlr-eq eq ν ν'
  | - ⇒ False)

lemma ltlr-eq-autoref[autoref-rules]:
  assumes EQP: (eq,(=)) ∈ R → R → bool-rel
  shows (ltlr-eq eq, (=)) ∈ ⟨R⟩ltlr-rel → ⟨R⟩ltlr-rel → bool-rel
proof (intro fun-relI)
  fix μ' μ ν' ν
  assume (μ',μ)∈⟨R⟩ltlr-rel and (ν',ν)∈⟨R⟩ltlr-rel
  then show (ltlr-eq eq μ' ν', μ=ν)∈bool-rel
    apply (induction arbitrary: ν' ν)
    apply (erule ltlr-rel-cases, simp-all) []
    apply (erule ltlr-rel-cases, simp-all) []
    apply (erule ltlr-rel-cases,
      simp-all add: EQP[THEN fun-relD, THEN fun-relD, THEN IdD]) []
    apply (erule ltlr-rel-cases,
      simp-all add: EQP[THEN fun-relD, THEN fun-relD, THEN IdD]) []
    apply (rotate-tac -1)
    apply (erule ltlr-rel-cases, simp-all) []
    apply (rotate-tac -1)
    apply (erule ltlr-rel-cases, simp-all) []
    apply (rotate-tac -1)
    apply (erule ltlr-rel-cases, simp-all) []
    apply (rotate-tac -1)
    apply (erule ltlr-rel-cases, simp-all) []
    apply (rotate-tac -1)
    apply (erule ltlr-rel-cases, simp-all) []
    done
qed
```

lemma *ltlr-dflt-cmp*[*autoref-rules-raw*]:
assumes *PREFER-id R*
shows
(*dflt-cmp* (≤) (<), *dflt-cmp* (≤) (<))
∈ ⟨*R*⟩*ltlr-rel* → ⟨*R*⟩*ltlr-rel* → *comp-res-rel*
using *assms*
by *simp*

type-synonym
node-name-impl = *node-name*

abbreviation (*input*) *node-name-rel* ≡ *Id* :: (*node-name-impl*×*node-name*) *set*

lemma *case-ltlr-gtransfer*:
assumes
γ *ai* ≤ *a*
γ *bi* ≤ *b*
⋀*a*. γ (*ci a*) ≤ *c a*
⋀*a*. γ (*di a*) ≤ *d a*
⋀*ltlr1 ltlr2*. γ (*ei ltlr1 ltlr2*) ≤ *e ltlr1 ltlr2*
⋀*ltlr1 ltlr2*. γ (*fi ltlr1 ltlr2*) ≤ *f ltlr1 ltlr2*
⋀*ltlr*. γ (*gi ltlr*) ≤ *g ltlr*
⋀*ltlr1 ltlr2*. γ (*hi ltlr1 ltlr2*) ≤ *h ltlr1 ltlr2*
⋀*ltlr1 ltlr2*. γ (*iiv ltlr1 ltlr2*) ≤ *i ltlr1 ltlr2*
shows γ (*case-ltlr ai bi ci di ei fi gi hi iiv* φ)
≤ (*case-ltlr a b c d e f g h i* φ)
apply (*cases* φ)
apply (*auto intro*: *assms*)
done

lemmas [*refine-transfer*]
= *case-ltlr-gtransfer*[**where** γ=*nres-of*] *case-ltlr-gtransfer*[**where** γ=*RETURN*]

lemma [*refine-transfer*]:
assumes
ai ≠ *dSUCCEED*
bi ≠ *dSUCCEED*
⋀*a*. *ci a* ≠ *dSUCCEED*
⋀*a*. *di a* ≠ *dSUCCEED*
⋀*ltlr1 ltlr2*. *ei ltlr1 ltlr2* ≠ *dSUCCEED*
⋀*ltlr1 ltlr2*. *fi ltlr1 ltlr2* ≠ *dSUCCEED*
⋀*ltlr*. *gi ltlr* ≠ *dSUCCEED*
⋀*ltlr1 ltlr2*. *hi ltlr1 ltlr2* ≠ *dSUCCEED*
⋀*ltlr1 ltlr2*. *iiv ltlr1 ltlr2* ≠ *dSUCCEED*
shows *case-ltlr ai bi ci di ei fi gi hi iiv* φ ≠ *dSUCCEED*
apply (*cases* φ)
apply (*simp-all add*: *assms*)
done

3.1.2 Nodes

record *'a node-impl* =
  name-impl :: *node-name-impl*
  incoming-impl :: (*node-name-impl*,*unit*) *RBT-Impl.rbt*
  new-impl :: *'a frml list*
  old-impl :: *'a frml list*
  next-impl :: *'a frml list*

definition *node-rel* **where** *node-rel-def-internal*: *node-rel Re R* ≡ {(
  (| *name-impl* = *namei*,
    incoming-impl = *inci*,
    new-impl = *newi*,
    old-impl = *oldi*,
    next-impl = *nexti*,
    ... = *morei*
  |),
  (| *name* = *name*,
    incoming = *inc*,
    new=*new*,
    old=*old*,
    next = *next*,
    ... = *more*
  |)) | *namei name inci inc newi new oldi old nexti next morei more*.
    (*namei*,*name*)∈*node-name-rel*
  ∧ (*inci*,*inc*)∈⟨*node-name-rel*⟩*dflt-rs-rel*
  ∧ (*newi*,*new*)∈⟨⟨*R*⟩*ltlr-rel*⟩*lss.rel*
  ∧ (*oldi*,*old*)∈⟨⟨*R*⟩*ltlr-rel*⟩*lss.rel*
  ∧ (*nexti*,*next*)∈⟨⟨*R*⟩*ltlr-rel*⟩*lss.rel*
  ∧ (*morei*,*more*)∈*Re*
  }

lemma *node-rel-def*: ⟨*Re*,*R*⟩*node-rel* = {(
  (| *name-impl* = *namei*,
    incoming-impl = *inci*,
    new-impl = *newi*,
    old-impl = *oldi*,
    next-impl = *nexti*,
    ... = *morei*
  |),
  (| *name* = *name*,
    incoming = *inc*,
    new=*new*,
    old=*old*,
    next = *next*,
    ... = *more*
  |)) | *namei name inci inc newi new oldi old nexti next morei more*.
    (*namei*,*name*)∈*node-name-rel*
  ∧ (*inci*,*inc*)∈⟨*node-name-rel*⟩*dflt-rs-rel*
  ∧ (*newi*,*new*)∈⟨⟨*R*⟩*ltlr-rel*⟩*lss.rel*

```
  ∧ (oldi,old)∈⟨⟨R⟩ltlr-rel⟩lss.rel
  ∧ (nexti,next)∈⟨⟨R⟩ltlr-rel⟩lss.rel
  ∧ (morei,more)∈Re
  } by (simp add: node-rel-def-internal relAPP-def)


lemma node-rel-sv[relator-props]:
  single-valued Re ⟹ single-valued R ⟹ single-valued (⟨Re,R⟩node-rel)
  apply (rule single-valuedI)
  apply (simp add: node-rel-def)
  apply (auto
    dest: single-valuedD lss.rel-sv[OF ltlr-rel-sv] map2set-rel-sv[OF ahm-rel-sv]
    dest: single-valuedD[
      OF map2set-rel-sv[OF rbt-map-rel-sv[OF single-valued-Id single-valued-Id]]
    ])
  done

consts i-node :: interface ⇒ interface ⇒ interface

lemmas [autoref-rel-intf] = REL-INTFI[of node-rel i-node]

lemma [autoref-rules]: (node-impl-ext, node-ext) ∈
  node-name-rel
  → ⟨node-name-rel⟩dflt-rs-rel
  → ⟨⟨R⟩ltlr-rel⟩lss.rel
  → ⟨⟨R⟩ltlr-rel⟩lss.rel
  → ⟨⟨R⟩ltlr-rel⟩lss.rel
  → Re
  → ⟨Re,R⟩node-rel
  unfolding node-rel-def
  by auto

lemma [autoref-rules]:
  (node-impl.name-impl-update,node.name-update)
  ∈ (node-name-rel → node-name-rel) → ⟨Re,R⟩node-rel → ⟨Re,R⟩node-rel
  (node-impl.incoming-impl-update,node.incoming-update)
  ∈ (⟨node-name-rel⟩dflt-rs-rel → ⟨node-name-rel⟩dflt-rs-rel)
    → ⟨Re,R⟩node-rel
    → ⟨Re,R⟩node-rel
  (node-impl.new-impl-update,node.new-update)
  ∈ (⟨⟨R⟩ltlr-rel⟩lss.rel → ⟨⟨R⟩ltlr-rel⟩lss.rel) → ⟨Re,R⟩node-rel → ⟨Re,R⟩node-rel
  (node-impl.old-impl-update,node.old-update)
  ∈ (⟨⟨R⟩ltlr-rel⟩lss.rel → ⟨⟨R⟩ltlr-rel⟩lss.rel) → ⟨Re,R⟩node-rel → ⟨Re,R⟩node-rel
  (node-impl.next-impl-update,node.next-update)
  ∈ (⟨⟨R⟩ltlr-rel⟩lss.rel → ⟨⟨R⟩ltlr-rel⟩lss.rel) → ⟨Re,R⟩node-rel → ⟨Re,R⟩node-rel
  (node-impl.more-update,node.more-update)
  ∈ (Re → Re) → ⟨Re,R⟩node-rel → ⟨Re,R⟩node-rel
  unfolding node-rel-def
  by (auto dest: fun-relD)
```

```
term name
lemma [autoref-rules]:
  (node-impl.name-impl,node.name)∈⟨Re,R⟩node-rel → node-name-rel
  (node-impl.incoming-impl,node.incoming)
  ∈ ⟨Re,R⟩node-rel → ⟨node-name-rel⟩dflt-rs-rel
  (node-impl.new-impl,node.new)∈⟨Re,R⟩node-rel → ⟨⟨R⟩ltlr-rel⟩lss.rel
  (node-impl.old-impl,node.old)∈⟨Re,R⟩node-rel → ⟨⟨R⟩ltlr-rel⟩lss.rel
  (node-impl.next-impl,node.next)∈⟨Re,R⟩node-rel → ⟨⟨R⟩ltlr-rel⟩lss.rel
  (node-impl.more, node.more)∈⟨Re,R⟩node-rel → Re
  unfolding node-rel-def by auto
```

3.2 Massaging the Abstract Algorithm

In a first step, we do some refinement steps on the abstract data structures, with the goal to make the algorithm more efficient.

3.2.1 Creation of the Nodes

In the expand-algorithm, we replace nested conditionals by case-distinctions, and slightly stratify the code.

```
abbreviation (input) expand2 exp n ns φ n1 nx1 n2 ≡ do {
    (nm, nds) ← exp (
      n(|
        new := insert n1 (new n),
        old := insert φ (old n),
        next := nx1 ∪ next n |),
      ns);
    exp (n(| name := nm, new := n2 ∪ new n, old := {φ} ∪ old n |), nds)
  }



definition expand-aimpl ≡ REC_T (λexpand (n,ns).
      if new n = {} then (
        if (∃n'∈ns. old n' = old n ∧ next n' = next n) then
          RETURN (name n, upd-incoming n ns)
        else do {
          ASSERT (n ∉ ns);
          ASSERT (name n ∉ name`ns);
          expand (|
            name=expand-new-name (name n),
            incoming={name n},
            new=next n,
            old={},
            next={} |),
            insert n ns)
        }
```

) *else do* {
$\varphi \leftarrow SPEC\ (\lambda x.\ x{\in}(new\ n))$;
let $n = n(\!|\ new := new\ n - \{\varphi\}\ |\!)$;
case φ *of*
$prop_r(q) \Rightarrow$
if $nprop_r(q){\in}old\ n$ *then* $RETURN\ (name\ n,\ ns)$
else $expand\ (n(\!|\ old := \{\varphi\} \cup old\ n\ |\!),\ ns)$
| $nprop_r(q) \Rightarrow$
if $prop_r(q){\in}old\ n$ *then* $RETURN\ (name\ n,\ ns)$
else $expand\ (n(\!|\ old := \{\varphi\} \cup old\ n\ |\!),\ ns)$
| $true_r \Rightarrow expand\ (n(\!|\ old := \{\varphi\} \cup old\ n\ |\!),\ ns)$
| $false_r \Rightarrow RETURN\ (name\ n,\ ns)$
| $\nu\ and_r\ \mu \Rightarrow expand\ (n(\!|$
$new := insert\ \nu\ (insert\ \mu\ (new\ n))$,
$old := \{\varphi\} \cup old\ n$,
$next := next\ n\ |\!),\ ns)$
| $X_r\ \nu \Rightarrow expand$
$(n(\!|\ new := new\ n,\ old := \{\varphi\} \cup old\ n,\ next := insert\ \nu\ (next\ n)\ |\!),\ ns)$
| $\mu\ or_r\ \nu \Rightarrow expand2\ expand\ n\ ns\ \varphi\ \mu\ \{\}\ \{\nu\}$
| $\mu\ U_r\ \nu \Rightarrow expand2\ expand\ n\ ns\ \varphi\ \mu\ \{\varphi\}\ \{\nu\}$
| $\mu\ R_r\ \nu \Rightarrow expand2\ expand\ n\ ns\ \varphi\ \nu\ \{\varphi\}\ \{\mu,\nu\}$
~~| - ⇒ do { (nm, nds) ← expand (n(| new := new1 φ ∪ new n, old := {φ} ∪ old n, next := next1 φ ∪ next n |), ns); expand (n(| name := nm, new := new2 φ ∪ new n, old := {φ} ∪ old n |), nds) }~~
}
)

lemma *expand-aimpl-refine*:
fixes $n\text{-}ns$ :: $('a\ node \times \text{-})$
defines $R \equiv Id \cap \{(\text{-},(n,ns)).\ \forall n'{\in}ns.\ n > name\ n'\}$
defines $R' \equiv Id \cap \{(\text{-},(n,ns)).\ \forall n'{\in}ns.\ name\ n > name\ n'\}$
assumes [*refine*]: $(n\text{-}ns',n\text{-}ns){\in}R'$
shows $expand\text{-}aimpl\ n\text{-}ns' \leq \Downarrow R\ (expand_T\ n\text{-}ns)$
using [[*goals-limit* = *1*]]
proof −
have [*relator-props*]: *single-valued R*
by (*auto simp add*: *R-def intro*: *single-valuedI*)
have [*relator-props*]: *single-valued R′*
by (*auto simp add*: *R′-def intro*: *single-valuedI*)

{
fix n :: *′a node* **and** *ns* **and** *n′ ns′*
assume $((n',\ ns'),\ (n,\ ns)) \in R'$
then have $(RETURN\ (name\ n',\ ns') \leq \Downarrow R\ (RETURN\ (name\ n,\ ns)))$
by (*auto simp*: *R-def R′-def pw-le-iff refine-pw-simps*)
} **note** *aux* = *this*

show *?thesis*
unfolding *expand-aimpl-def expand$_T$-def*
apply *refine-rcg*
apply (*simp add*: *R-def R′-def*)
apply (*simp add*: *R-def R′-def*)
apply (*auto simp add*: *R-def R′-def upd-incoming-def*) []
apply (*auto simp add*: *R-def R′-def upd-incoming-def*) []
apply (*auto simp add*: *R-def R′-def upd-incoming-def*) []
apply *rprems*
apply (*auto simp*: *R′-def expand-new-name-def*) []
apply (*simp add*: *R′-def*)

apply (*auto split*: *ltlr.split*) []
apply (*fastforce simp*: *R-def R′-def*) []
apply (*fastforce simp*: *R-def R′-def*) []

apply (*auto simp*: *R-def R′-def*) []
apply (*auto simp*: *R-def R′-def*) []
apply (*auto simp*: *R-def R′-def*) []
apply (*auto simp*: *R-def R′-def*) []
apply (*auto simp*: *R-def R′-def*) []
apply (*auto simp*: *R-def R′-def*) []
apply (*auto simp*: *R-def R′-def*) []
apply (*auto simp*: *R-def R′-def*) []
apply (*auto simp*: *R-def R′-def*) []
apply (*refine-rcg*, *rprems*, (*fastforce simp*: *R-def R′-def*)+) []
apply (*fastforce simp*: *R′-def*) []
apply (*refine-rcg*, *rprems*, (*fastforce simp*: *R-def R′-def*)+) []
apply (*refine-rcg*, *rprems*, (*fastforce simp*: *R-def R′-def*)+) []
done
qed

thm *create-graph-def*
definition *create-graph-aimpl* φ = *do* {
(-, *nds*) ←
expand-aimpl
(⦇*name* = *expand-new-name expand-init*, *incoming* = {*expand-init*},
new = {φ}, *old* = {}, *next* = {}⦈,
{});
RETURN nds
}

lemma *create-graph-aimpl-refine*: *create-graph-aimpl* $\varphi \leq \Downarrow Id$ (*create-graph$_T$* φ)
unfolding *create-graph-aimpl-def create-graph$_T$-def*
apply (*refine-rcg expand-aimpl-refine*)
apply *auto*
done

3.2.2 Creation of GBA from Nodes

We summarize creation of the GBA and renaming of the nodes into one step

lemma *create-name-gba-alt*: *create-name-gba* φ = *do* {
  nds ← *create-graph*$_T$ φ;
  ASSERT (*nds-invars nds*);
  RETURN (*gba-rename-ext* (λ-. ()) *name* (*create-gba-from-nodes* φ *nds*))
  }
proof −
  have [*simp*]: ⋀*nds*. *g-V* (*create-gba-from-nodes* φ *nds*) = *nds*
    by (*auto simp*: *create-gba-from-nodes-def*)

  show *?thesis*
    unfolding *create-name-gba-def create-gba-def*
    by *simp*
qed

In the following, we implement the componenents of the renamed GBA separately.

definition *build-succ nds* =
  FOREACH
    nds (λ*q′ s*.
    FOREACH
      (*incoming q′*) (λ*qn s*.
        if qn=*expand-init then*
          RETURN s
        else
          RETURN (*s*(*qn* ↦ *insert* (*name q′*) (*the-default* {} (*s qn*))))
) *s*
) *Map.empty*

lemma *build-succ-aux1*:
  assumes [*simp*]: *finite nds*
  assumes [*simp*]: ⋀*q*. *q*∈*nds* ⟹ *finite* (*incoming q*)
  shows *build-succ nds* ≤ *SPEC* (λ*r*. *r* = (λ*qn*.
  dflt-None-set {*qn′*. ∃ *q′*.
    q′∈*nds* ∧ *qn′* = *name q′* ∧ *qn*∈*incoming q′* ∧ *qn*≠*expand-init*
  }))
  unfolding *build-succ-def*
  apply (*refine-rcg refine-vcg*
    FOREACH-rule[**where**
      I=λ*it s*. *s* = (λ*qn*. *dflt-None-set* {*qn′*. ∃ *q′*. *q′*∈*nds*−*it* ∧ *qn′* = *name q′*
    ∧ *qn*∈*incoming q′* ∧ *qn*≠*expand-init* })])

  apply (*simp-all add*: *dflt-None-set-def*) [*2*]
  apply (*rename-tac q′ it s*)
  apply (*rule-tac I*=λ*it2 s*. *s* =

```
    (λqn. dflt-None-set (
      {qn'. ∃ q'. q'∈nds−it ∧ qn' = name q' ∧ qn∈incoming q' ∧ qn≠expand-init }
∪
      {qn' . qn'=name q' ∧ qn∈incoming q' − it2 ∧ qn≠expand-init} ))
    in FOREACH-rule)

  apply auto []

  apply (simp-all add: dflt-None-set-def)

  apply (refine-rcg refine-vcg)
  apply (auto simp: dflt-None-set-def intro!: ext) []
  apply (rule ext, (auto) [])+
  done

lemma build-succ-aux2:
  assumes NINIT: expand-init ∉ name‘nds
  assumes CL: ⋀nd. nd∈nds ⟹ incoming nd ⊆ insert expand-init (name‘nds)
  shows
  (λqn. dflt-None-set
    {qn'. ∃ q'. q'∈nds ∧ qn' = name q' ∧ qn∈incoming q' ∧ qn≠expand-init })
  = (λqn. dflt-None-set (succ-of-E
      (rename-E name {(q, q'). q ∈ nds ∧ q' ∈ nds ∧ name q ∈ incoming q'}) qn))

  (is (λqn. dflt-None-set (?L qn)) = (λqn. dflt-None-set (?R qn)))
  apply (intro ext)
  apply (fo-rule arg-cong)
proof (intro ext equalityI subsetI)
  fix qn x
  assume x∈?R qn
  then show x∈?L qn using NINIT
    by (force simp: succ-of-E-def)
next
  fix qn x
  assume XL: x∈?L qn
  show x∈?R qn
  proof (cases qn = expand-init)
    case False
    from XL obtain q' where
      A: q'∈nds qn∈incoming q'
      and [simp]: x=name q'
      by auto
    from False obtain q where B: q∈nds and [simp]: qn = name q
      using CL A by auto

    from A B show x∈?R qn
      by (auto simp: succ-of-E-def image-def)
  next
    case [simp]: True
```

from *XL* **show** *x∈?R qn* **by** *simp*
qed
qed

lemma *build-succ-correct*:
assumes *NINIT*: *expand-init ∉ name'nds*
assumes *FIN*: *finite nds*
assumes *CL*: $\bigwedge$*nd. nd∈nds ⟹ incoming nd ⊆ insert expand-init* (*name'nds*)
shows *build-succ nds ≤ SPEC* (*λr.*
E-of-succ (*λqn. the-default* {} (*r qn*))
= rename-E (*λu. name u*) {(*q*, *q'*). *q ∈ nds ∧ q' ∈ nds ∧ name q ∈ incoming q'*})
proof −
from *FIN CL* **have** *FIN'*: $\bigwedge$*q. q∈nds ⟹ finite* (*incoming q*)
by (*metis finite-imageI finite-insert infinite-super*)

note *build-succ-aux1*[*OF FIN FIN'*]
also note *build-succ-aux2*[*OF NINIT CL*]
finally show *?thesis*
by (*rule order-trans*) *auto*
qed

primrec *until-frmlsr* :: *'a frml ⇒* (*'a frml × 'a frml*) *set* **where**
until-frmlsr (*μ and$_r$ ψ*) = (*until-frmlsr μ*) ∪ (*until-frmlsr ψ*)
| *until-frmlsr* (*X$_r$ μ*) = *until-frmlsr μ*
| *until-frmlsr* (*μ U$_r$ ψ*) = *insert* (*μ*, *ψ*) ((*until-frmlsr μ*) ∪ (*until-frmlsr ψ*))
| *until-frmlsr* (*μ R$_r$ ψ*) = (*until-frmlsr μ*) ∪ (*until-frmlsr ψ*)
| *until-frmlsr* (*μ or$_r$ ψ*) = (*until-frmlsr μ*) ∪ (*until-frmlsr ψ*)
| *until-frmlsr* (*true$_r$*) = {}
| *until-frmlsr* (*false$_r$*) = {}
| *until-frmlsr* (*prop$_r$*(-)) = {}
| *until-frmlsr* (*nprop$_r$*(-)) = {}

lemma *until-frmlsr-correct*:
until-frmlsr φ = {(*μ*, *η*). *Until-ltlr μ η ∈ subfrmlsr φ*}
by (*induct φ*) *auto*

definition *build-F nds φ*
≡ (*λ*(*μ*,*η*). *name* ' {*q ∈ nds*. (*Until-ltlr μ η ∈ old q ⟶ η ∈ old q*)}) '
until-frmlsr φ

lemma *build-F-correct*: *build-F nds φ* =
{*name* ' *A* |*A*. ∃ *μ η*. *A* = {*q ∈ nds*. *Until-ltlr μ η ∈ old q ⟶ η ∈ old q*} ∧
Until-ltlr μ η ∈ subfrmlsr φ}
proof −
have {*name* ' *A* |*A*. ∃ *μ η*. *A* = {*q ∈ nds*. *Until-ltlr μ η ∈ old q ⟶ η ∈ old q*} ∧

```
                    Until-ltlr μ η ∈ subfrmlsr φ}
    = (λ(μ,η). name‘{q∈nds. Until-ltlr μ η ∈ old q ⟶ η ∈ old q})
      ‘ {(μ, η). Until-ltlr μ η ∈ subfrmlsr φ}
    by auto
  also have ... = (λ(μ,η). name‘{q∈nds. Until-ltlr μ η ∈ old q ⟶ η ∈ old q})
      ‘ until-frmlsr φ
    unfolding until-frmlsr-correct ..
  finally show ?thesis
    unfolding build-F-def by simp
qed



definition pn-props ps ≡ FOREACHi
  (λit (P,N). P = {p. Prop-ltlr p ∈ ps − it} ∧ N = {p. Nprop-ltlr p ∈ ps − it})
  ps (λp (P,N).
    case p of Prop-ltlr p ⇒ RETURN (insert p P,N)
    | Nprop-ltlr p ⇒ RETURN (P, insert p N)
    | - ⇒ RETURN (P,N)
  ) ({},{})

lemma pn-props-correct:
  assumes [simp]: finite ps
  shows pn-props ps ≤ SPEC(λr. r =
  ({p. Prop-ltlr p ∈ ps}, {p. Nprop-ltlr p ∈ ps}))
  unfolding pn-props-def
  apply (refine-rcg refine-vcg)
  apply (auto split: ltlr.split)
  done

definition pn-map nds ≡ FOREACH nds
  (λnd m. do {
    PN ← pn-props (old nd);
    RETURN (m(name nd ↦ PN))
  }) Map.empty

lemma pn-map-correct:
  assumes [simp]: finite nds
  assumes FIN': ⋀nd. nd∈nds ⟹ finite (old nd)
  assumes INJ: inj-on name nds
  shows pn-map nds ≤ SPEC (λr. ∀ qn.
    case r qn of
      None ⇒ qn ∉ name‘nds
    | Some (P,N) ⇒ qn ∈ name‘nds
      ∧ P = {p. Prop-ltlr p ∈ old (the-inv-into nds name qn)}
      ∧ N = {p. Nprop-ltlr p ∈ old (the-inv-into nds name qn)}
  )
  unfolding pn-map-def
  apply (refine-rcg refine-vcg
```

```
    FOREACH-rule[where I=λit r. ∀ qn.
      case r qn of
        None ⇒ qn ∉ name'(nds − it)
      | Some (P,N) ⇒ qn ∈ name'(nds − it)
        ∧ P = {p. Prop-ltlr p ∈ old (the-inv-into nds name qn)}
        ∧ N = {p. Nprop-ltlr p ∈ old (the-inv-into nds name qn)}]
    order-trans[OF pn-props-correct]
  )
  apply simp-all
  apply (blast dest: subsetD[THEN FIN'])
  apply (force
    split: option.splits
    simp: the-inv-into-f-f[OF INJ] it-step-insert-iff)
  apply (fastforce split: option.splits)
  done



definition cr-rename-gba nds φ ≡ do {
  let V = name ' nds;
  let V0 = name ' {q ∈ nds. expand-init ∈ incoming q};
  succmap ← build-succ nds;
  let E = E-of-succ (the-default {} o succmap);
  let F = build-F nds φ;
  pnm ← pn-map nds;
  let L = (λqn l. case pnm qn of
      None ⇒ False
    | Some (P,N) ⇒ (∀ p∈P. p∈(l:::_r⟨Id⟩fun-set-rel)) ∧ (∀ p∈N. p∉l)
  );
  RETURN (⦇ g-V = V, g-E=E, g-V0=V0, gbg-F = F, gba-L = L ⦈)
}

lemma cr-rename-gba-refine:
  assumes INV: nds-invars nds
  assumes REL[simplified]: (nds',nds)∈Id (φ',φ)∈Id
  shows cr-rename-gba nds' φ'
  ≤ ⇓Id (RETURN (gba-rename-ext (λ-. ()) name (create-gba-from-nodes φ nds)))
  unfolding RETURN-SPEC-conv
proof (rule Id-SPEC-refine)
  from INV have
    NINIT: expand-init ∉ name'nds
    and FIN: finite nds
    and FIN': ⋀nd. nd∈nds ⟹ finite (old nd)
    and CL: ⋀nd. nd∈nds ⟹ incoming nd ⊆ insert expand-init (name'nds)
    and INJ: inj-on name nds
    unfolding nds-invars-def by auto
  show cr-rename-gba nds' φ'
    ≤ SPEC (λx. x = gba-rename-ext (λ-. ()) name (create-gba-from-nodes φ nds))
    unfolding REL
```

```
    unfolding cr-rename-gba-def
    apply (refine-rcg refine-vcg
      order-trans[OF build-succ-correct[OF NINIT FIN CL]]
      order-trans[OF pn-map-correct[OF FIN FIN' INJ]]
    )
    unfolding gba-rename-ecnv-def gb-rename-ecnv-def
      fr-rename-ext-def create-gba-from-nodes-def
    apply simp
    apply (intro conjI)
    apply (simp add: comp-def)
    apply (simp add: build-F-correct)
    apply (intro ext)
    apply (drule-tac x=qn in spec)
    apply (auto simp: the-inv-into-f-f[OF INJ] split: option.split prod.split)
    done
qed

definition create-name-gba-aimpl φ ≡ do {
  nds ← create-graph-aimpl φ;
  ASSERT (nds-invars nds);
  cr-rename-gba nds φ
}


lemma create-name-gba-aimpl-refine:
  create-name-gba-aimpl φ ≤ ⇓Id (create-name-gba φ)
  unfolding create-name-gba-aimpl-def create-name-gba-alt
  apply (refine-rcg create-graph-aimpl-refine cr-rename-gba-refine)
  by auto
```

3.3 Refinement to Efficient Data Structures

3.3.1 Creation of GBA from Nodes

```
schematic-goal until-frmlsr-impl-aux:
  assumes [relator-props, simp]: R=Id
  shows (?c,until-frmlsr)
  ∈ ⟨⟨R::(-×-::linorder) set)⟩ltlr-rel → ⟨⟨R⟩ltlr-rel ×r ⟨R⟩ltlr-rel⟩dflt-rs-rel
  unfolding until-frmlsr-def
  apply (autoref (keep-goal, trace))
  done
concrete-definition until-frmlsr-impl uses until-frmlsr-impl-aux
lemmas [autoref-rules] = until-frmlsr-impl.refine[OF PREFER-id-D]



schematic-goal build-succ-impl-aux:
  shows (?c,build-succ) ∈
```

$\langle\langle Rm,R\rangle node\text{-}rel\rangle list\text{-}set\text{-}rel$
$\rightarrow \langle\langle nat\text{-}rel,\langle nat\text{-}rel\rangle list\text{-}set\text{-}rel\rangle iam\text{-}map\text{-}rel\rangle nres\text{-}rel$
unfolding *build-succ-def*[*abs-def*] *expand-init-def*
using [[*autoref-trace-failed-id*]]
apply (*autoref* (*keep-goal*, *trace*))
done

concrete-definition *build-succ-impl* **uses** *build-succ-impl-aux*
lemmas [*autoref-rules*] = *build-succ-impl.refine*

schematic-goal *build-succ-code-aux*: *RETURN ?c* $\leq$ *build-succ-impl x*
unfolding *build-succ-impl-def*
apply (*refine-transfer* (*post*))
done

concrete-definition *build-succ-code* **uses** *build-succ-code-aux*
lemmas [*refine-transfer*] = *build-succ-code.refine*

schematic-goal *build-F-impl-aux*:
assumes [*relator-props*]: $R = Id$
shows (*?c*,*build-F*) $\in$
$\langle\langle Rm,R\rangle node\text{-}rel\rangle list\text{-}set\text{-}rel \rightarrow \langle R\rangle ltlr\text{-}rel \rightarrow \langle\langle nat\text{-}rel\rangle list\text{-}set\text{-}rel\rangle list\text{-}set\text{-}rel$
unfolding *build-F-def*[*abs-def*]
using [[*autoref-trace-failed-id*]]
apply (*autoref* (*keep-goal*, *trace*))
done

concrete-definition *build-F-impl* **uses** *build-F-impl-aux*
lemmas [*autoref-rules*] = *build-F-impl.refine*[*OF PREFER-id-D*]

schematic-goal *pn-map-impl-aux*:
shows (*?c*,*pn-map*) $\in$
$\langle\langle Rm,Id\rangle node\text{-}rel\rangle list\text{-}set\text{-}rel$
$\rightarrow \langle\langle nat\text{-}rel,\langle Id\rangle list\text{-}set\text{-}rel \times_r \langle Id\rangle list\text{-}set\text{-}rel\rangle iam\text{-}map\text{-}rel\rangle nres\text{-}rel$
unfolding *pn-map-def*[*abs-def*] *pn-props-def*
using [[*autoref-trace-failed-id*]]
apply (*autoref* (*keep-goal*, *trace*))
done

concrete-definition *pn-map-impl* **uses** *pn-map-impl-aux*
lemma *pn-map-impl-autoref*[*autoref-rules*]:
assumes *PREFER-id R*

shows (*pn-map-impl*,*pn-map*) ∈
⟨⟨*Rm*,*R*⟩*node-rel*⟩*list-set-rel*
→ ⟨⟨*nat-rel*,⟨*R*⟩*list-set-rel* ×$_r$ ⟨*R*⟩*list-set-rel*⟩*iam-map-rel*⟩*nres-rel*
using *assms pn-map-impl.refine* **by** *simp*

schematic-goal *pn-map-code-aux*: *RETURN ?c* ≤ *pn-map-impl x*
unfolding *pn-map-impl-def*
apply (*refine-transfer* (*post*))
done
concrete-definition *pn-map-code* **uses** *pn-map-code-aux*
lemmas [*refine-transfer*] = *pn-map-code.refine*

schematic-goal *cr-rename-gba-impl-aux*:
assumes *ID*[*relator-props*]: *R*=*Id*
notes [*autoref-tyrel del*] = *tyrel-dflt-linorder-set*
notes [*autoref-tyrel*] = *ty-REL*[*of* ⟨*nat-rel*⟩*list-set-rel*]
shows (*?c*,*cr-rename-gba*) ∈
⟨⟨*Rm*,*R*⟩*node-rel*⟩*list-set-rel* → ⟨*R*⟩*ltlr-rel* → (*?R*::(*?'c* × -) *set*)
unfolding *ID*
unfolding *cr-rename-gba-def*[*abs-def*] *expand-init-def comp-def*
using [[*autoref-trace-failed-id*]]
apply (*autoref* (*keep-goal*, *trace*))
done
concrete-definition *cr-rename-gba-impl* **uses** *cr-rename-gba-impl-aux*

thm *cr-rename-gba-impl.refine*

lemma *cr-rename-gba-autoref*[*autoref-rules*]:
assumes *PREFER-id R*
shows (*cr-rename-gba-impl*, *cr-rename-gba*) ∈
⟨⟨*Rm*, *R*⟩*node-rel*⟩*list-set-rel* → ⟨*R*⟩*ltlr-rel* →
⟨*gbav-impl-rel-ext unit-rel nat-rel* (⟨*R*⟩*fun-set-rel*)⟩*nres-rel*
using *assms cr-rename-gba-impl.refine*[*of R Rm*] **by** *simp*

schematic-goal *cr-rename-gba-code-aux*: *RETURN ?c* ≤ *cr-rename-gba-impl x y*
unfolding *cr-rename-gba-impl-def*
apply (*refine-transfer* (*post*))
done
concrete-definition *cr-rename-gba-code* **uses** *cr-rename-gba-code-aux*
lemmas [*refine-transfer*] = *cr-rename-gba-code.refine*

3.3.2 Creation of Graph

The implementation of the node-set. The relation enforces that there are no different nodes with the same name. This effectively establishes an additional invariant, made explicit by an assertion in the refined program. This invariant allows for a more efficient implementation.

definition *ls-nds-rel-def-internal*:
ls-nds-rel R ≡ ⟨*R*⟩*list-set-rel* ∩ {(-,*s*). *inj-on name s*}
lemma *ls-nds-rel-def*: ⟨*R*⟩*ls-nds-rel* = ⟨*R*⟩*list-set-rel* ∩ {(-,*s*). *inj-on name s*}
by (*simp add*: *relAPP-def ls-nds-rel-def-internal*)

lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of ls-nds-rel i-set*]

lemma *ls-nds-rel-sv*[*relator-props*]:
assumes *single-valued R*
shows *single-valued* (⟨*R*⟩*ls-nds-rel*)
using *list-set-rel-sv*[*OF assms*]
unfolding *ls-nds-rel-def*
by (*metis inf.cobounded1 single-valued-subset*)

context begin interpretation *autoref-syn* **.**
lemma *lsnds-empty-autoref*[*autoref-rules*]:
assumes *PREFER-id R*
shows ([],{})∈⟨*R*⟩*ls-nds-rel*
using *assms*
apply (*simp add*: *ls-nds-rel-def*)
by *autoref*

lemma *lsnds-insert-autoref*[*autoref-rules*]:
assumes *SIDE-PRECOND* (*name n* ∉ *name‘ns*)
assumes (*n′*,*n*)∈*R*
assumes (*ns′*,*ns*)∈⟨*R*⟩*ls-nds-rel*
shows (*n′*#*ns′*,(*OP insert* ::: *R* → ⟨*R*⟩*ls-nds-rel* → ⟨*R*⟩*ls-nds-rel*)$*n*$*ns*)
∈ ⟨*R*⟩*ls-nds-rel*
using *assms*
unfolding *ls-nds-rel-def*
apply *simp*
proof (*elim conjE*, *rule conjI*)
assume [*autoref-rules*]: (*n′*, *n*) ∈ *R* (*ns′*, *ns*) ∈ ⟨*R*⟩*list-set-rel*
assume *name n* ∉ *name ‘ ns*
and *inj-on name ns*
then have *n* ∉ *ns* **by** (*auto*)
then show (*n′ # ns′*, *insert n ns*) ∈ ⟨*R*⟩*list-set-rel*
by *autoref*
qed *auto*

lemma *ls-nds-image-autoref-aux*:
assumes [*autoref-rules*]: (*fi*,*f*) ∈ *Ra* → *Rb*
assumes (*l*,*s*) ∈ ⟨*Ra*⟩*ls-nds-rel*
assumes [*simp*]: ∀ *x*. *name* (*f x*) = *name x*
shows (*map fi l*, *f‘s*) ∈ ⟨*Rb*⟩*ls-nds-rel*
proof −
from *assms* **have**
[*autoref-rules*]: (*l*,*s*)∈⟨*Ra*⟩*list-set-rel*
and *INJ*: (*inj-on name s*)

by (*auto simp add*: *ls-nds-rel-def*)

have [*simp*]: *inj-on f s*
by (*rule inj-onI*) (*metis INJ assms*(*3*) *inj-on-eq-iff*)

have (*map fi l*, *f*‘*s*) ∈ ⟨*Rb*⟩*list-set-rel*
by (*autoref* (*keep-goal*))
moreover have *inj-on name* (*f*‘*s*)
apply (*rule inj-onI*)
apply (*auto simp*: *image-iff dest*: *inj-onD*[*OF INJ*])
done
ultimately show *?thesis*
by (*auto simp*: *ls-nds-rel-def*)
qed

lemma *ls-nds-image-autoref*[*autoref-rules*]:
assumes (*fi*,*f*) ∈ *Ra* → *Rb*
assumes *SIDE-PRECOND* (∀ *x*. *name* (*f x*) = *name x*)
shows (*map fi*, (*OP* (‘) ::: (*Ra*→*Rb*) → ⟨*Ra*⟩*ls-nds-rel* → ⟨*Rb*⟩*ls-nds-rel*)$*f*)
∈ ⟨*Ra*⟩*ls-nds-rel* → ⟨*Rb*⟩*ls-nds-rel*
using *assms*
unfolding *autoref-tag-defs*
using *ls-nds-image-autoref-aux*
by *blast*

lemma *list-set-autoref-to-list*[*autoref-ga-rules*]:
shows *is-set-to-sorted-list* (λ- -. *True*) *R ls-nds-rel id*
unfolding *is-set-to-list-def is-set-to-sorted-list-def ls-nds-rel-def*
it-to-sorted-list-def list-set-rel-def br-def
by *auto*

end

context begin interpretation *autoref-syn* **.**
lemma [*autoref-itype*]:
upd-incoming
::$_i$ ⟨*Im*, *I*⟩$_i$*i-node* →$_i$ ⟨⟨*Im*′, *I*⟩$_i$*i-node*⟩$_i$*i-set* →$_i$ ⟨⟨*Im*′, *I*⟩$_i$*i-node*⟩$_i$*i-set*
by *simp*
end

term *upd-incoming*
schematic-goal *upd-incoming-impl-aux*:
assumes *REL-IS-ID R*
shows (*?c*, *upd-incoming*)∈⟨*Rm1*,*R*⟩*node-rel*
→ ⟨⟨*Rm2*,*R*⟩*node-rel*⟩*ls-nds-rel*
→ ⟨⟨*Rm2*,*R*⟩*node-rel*⟩*ls-nds-rel*
using *assms* **apply** *simp*
unfolding *upd-incoming-def*[*abs-def*]

using [[*autoref-trace-failed-id*]]
apply (*autoref* (*keep-goal*))
done

concrete-definition *upd-incoming-impl* **uses** *upd-incoming-impl-aux*
lemmas [*autoref-rules*] = *upd-incoming-impl.refine*[*OF PREFER-D*[*of REL-IS-ID*]]

schematic-goal *expand-impl-aux*: (*?c*, *expand-aimpl*) ∈
⟨*unit-rel*,*Id*⟩*node-rel* ×$_r$ ⟨⟨*unit-rel*,*Id*⟩*node-rel*⟩*ls-nds-rel*
→ ⟨*nat-rel* ×$_r$ ⟨⟨*unit-rel*,*Id*⟩*node-rel*⟩*ls-nds-rel*⟩*nres-rel*
unfolding *expand-aimpl-def*[*abs-def*] *expand-new-name-def*
using [[*autoref-trace-failed-id*, *autoref-trace-intf-unif*]]
apply (*autoref* (*trace*,*keep-goal*))
done

concrete-definition *expand-impl* **uses** *expand-impl-aux*

context begin interpretation *autoref-syn* **.**
lemma [*autoref-itype*]: *expand*$_T$
::$_i$ ⟨⟨*i-unit*, *I*⟩$_i$*i-node*, ⟨⟨*i-unit*, *I*⟩$_i$*i-node*⟩$_i$*i-set*⟩$_i$*i-prod*
→$_i$ ⟨⟨*i-nat*,⟨⟨*i-unit*, *I*⟩$_i$*i-node*⟩$_i$*i-set*⟩$_i$*i-prod*⟩$_i$*i-nres* **by** *simp*

lemma *expand-autoref*[*autoref-rules*]:
assumes *ID*: *PREFER-id R*
assumes *A*: (*n-ns′*, *n-ns*)
∈ ⟨*unit-rel*,*R*⟩*node-rel* ×$_r$ ⟨⟨*unit-rel*,*R*⟩*node-rel*⟩*list-set-rel*
assumes *B*: *SIDE-PRECOND* (
let (*n*,*ns*)=*n-ns in inj-on name ns* ∧ (∀ *n′*∈*ns*. *name n* > *name n′*)
)
shows (*expand-impl n-ns′*,
(*OP expand-aimpl*
::: ⟨*unit-rel*,*R*⟩*node-rel* ×$_r$ ⟨⟨*unit-rel*,*R*⟩*node-rel*⟩*list-set-rel*
→ ⟨*nat-rel* ×$_r$ ⟨⟨*unit-rel*,*R*⟩*node-rel*⟩*list-set-rel*⟩*nres-rel*)$*n-ns*)
∈ ⟨*nat-rel* ×$_r$ ⟨⟨*unit-rel*,*R*⟩*node-rel*⟩*list-set-rel*⟩*nres-rel*
proof *simp*
from *ID A B* **have**
1: (*n-ns*, *n-ns*) ∈ *Id* ∩ {(-,(*n*,*ns*)). ∀ *n′*∈*ns*. *name n* > *name n′*}
and *2*: (*n-ns′*, *n-ns*)
∈ ⟨*unit-rel*,*Id*⟩*node-rel* ×$_r$ ⟨⟨*unit-rel*,*Id*⟩*node-rel*⟩*ls-nds-rel*
unfolding *ls-nds-rel-def*
apply −
apply *auto* []
apply (*cases n-ns′*)
apply *auto* []
done

have [*simp*]: *single-valued* (*nat-rel* ×$_r$ ⟨⟨*unit-rel*, *Id*⟩*node-rel*⟩*list-set-rel*)
by *tagged-solver*

```
  have [relator-props]: ⋀R. single-valued (Id ∩ R)
    by (metis IntE single-valuedD single-valuedI single-valued-Id)

  have [simp]: single-valued ((nat-rel ×ᵣ ⟨⟨unit-rel, Id⟩node-rel⟩ls-nds-rel) O
                              (Id ∩ {(-, n, ns). ∀n'∈ns. name n' < n}))
    by (tagged-solver)

  from expand-impl.refine[THEN fun-relD, OF 2, THEN nres-relD]
  show (expand-impl n-ns', expand-aimpl n-ns)
    ∈ ⟨nat-rel ×ᵣ ⟨⟨unit-rel, R⟩node-rel⟩list-set-rel⟩nres-rel
    apply −
    apply (rule nres-relI)
    using ID
    apply (simp add: pw-le-iff refine-pw-simps)
    apply (fastforce simp: ls-nds-rel-def)
    done
qed

end


schematic-goal expand-code-aux: RETURN ?c ≤ expand-impl n-ns
  unfolding expand-impl-def
  by (refine-transfer the-resI)
concrete-definition expand-code uses expand-code-aux
prepare-code-thms expand-code-def
lemmas [refine-transfer] = expand-code.refine


schematic-goal create-graph-impl-aux:
  assumes ID: R=Id
  shows (?c, create-graph-aimpl)
    ∈ ⟨R⟩ltlr-rel → ⟨⟨⟨unit-rel,R⟩node-rel⟩list-set-rel⟩nres-rel
  unfolding ID
  unfolding create-graph-aimpl-def[abs-def] expand-init-def expand-new-name-def
  using [[autoref-trace-failed-id]]
  apply (autoref (keep-goal))
  done
concrete-definition create-graph-impl uses create-graph-impl-aux

lemmas [autoref-rules] = create-graph-impl.refine[OF PREFER-id-D]

schematic-goal create-graph-code-aux: RETURN ?c ≤ create-graph-impl φ
  unfolding create-graph-impl-def
  by refine-transfer
concrete-definition create-graph-code uses create-graph-code-aux
lemmas [refine-transfer] = create-graph-code.refine
```

schematic-goal *create-name-gba-impl-aux*:
(*?c*, (*create-name-gba-aimpl*:: *'a*::*linorder ltlr* ⇒ -))
∈ ⟨*Id*⟩*ltlr-rel* → (*?R*::(*?'c*×-) *set*)
unfolding *create-name-gba-aimpl-def*[*abs-def*]
using [[*autoref-trace-failed-id*]]
apply (*autoref* (*keep-goal*, *trace*))
done
concrete-definition *create-name-gba-impl* **uses** *create-name-gba-impl-aux*

lemma *create-name-gba-autoref*[*autoref-rules*]:
assumes *PREFER-id R*
shows
(*create-name-gba-impl*, *create-name-gba*)
∈ ⟨*R*⟩*ltlr-rel* → ⟨*gbav-impl-rel-ext unit-rel nat-rel* (⟨*R*⟩*fun-set-rel*)⟩*nres-rel*
(**is** -:-→⟨*?R*⟩*nres-rel*)
proof (*intro fun-relI nres-relI*)
fix *φ φ'*
assume *A*: (*φ*,*φ'*)∈⟨*R*⟩*ltlr-rel*
from *assms* **have** *RID*[*simp*]: *R*=*Id* **by** *simp*

note *create-name-gba-impl.refine*[*THEN fun-relD*,*THEN nres-relD*, *OF A*[*unfolded RID*]]
also note *create-name-gba-aimpl-refine*
finally show *create-name-gba-impl φ* ≤ ⇓ *?R* (*create-name-gba φ'*) **by** *simp*
qed

schematic-goal *create-name-gba-code-aux*: *RETURN ?c* ≤ *create-name-gba-impl φ*
unfolding *create-name-gba-impl-def*
by (*refine-transfer* (*post*))
concrete-definition *create-name-gba-code* **uses** *create-name-gba-code-aux*
lemmas [*refine-transfer*] = *create-name-gba-code.refine*

schematic-goal *create-name-igba-impl-aux*:
assumes *RID*: *R*=*Id*
shows (*?c*,*create-name-igba*)∈
⟨*R*⟩*ltlr-rel* → ⟨*igbav-impl-rel-ext unit-rel nat-rel* (⟨*R*⟩*fun-set-rel*)⟩*nres-rel*
unfolding *RID*
unfolding *create-name-igba-def*[*abs-def*]
using [[*autoref-trace-failed-id*]]
apply (*autoref* (*trace*, *keep-goal*))
done
concrete-definition *create-name-igba-impl* **uses** *create-name-igba-impl-aux*
lemmas [*autoref-rules*] = *create-name-igba-impl.refine*[*OF PREFER-id-D*]

schematic-goal *create-name-igba-code-aux*: *RETURN ?c* ≤ *create-name-igba-impl φ*

unfolding *create-name-igba-impl-def*
by (*refine-transfer* (*post*))
concrete-definition *create-name-igba-code* **uses** *create-name-igba-code-aux*
lemmas [*refine-transfer*] = *create-name-igba-code.refine*

export-code *create-name-igba-code* **checking** *SML*

end

References

[1] J. Esparza, P. Lammich, R. Neumann, T. Nipkow, A. Schimpf, and J.-G. Smaus. A fully verified executable ltl model checker. In N. Sharygina and H. Veith, editors, *Computer Aided Verification*, volume 8044 of *Lecture Notes in Computer Science*, pages 463–478. Springer Berlin Heidelberg, 2013.

[2] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV*, pages 3–18, London, UK, UK, 1996. Chapman & Hall, Ltd.

[3] S. Merz. Stuttering equivalence. *Archive of Formal Proofs*, May 2012. http://isa-afp.org/entries/Stuttering_Equivalence.shtml, Formal proof development.