

Bounded-Deducibility Security

Andrei Popescu

Peter Lammich

Contents

1	Introduction	1
2	Trivia	2
3	IO Automaton	2
3.1	Preliminaries	2
3.2	Reachability and invariance	2
3.3	System traces	3
3.4	Traces versus reachability	5
4	BD Security	6
4.1	Definition	6
4.2	Unwinding proof method	8
5	Compositional Reasoning	16
5.1	Preliminaries	16
5.2	Decomposition into an arbitrary network of components	17
5.3	A customization for linear modular reasoning	19
5.4	Instances	21

1 Introduction

This is a formalization of *bounded-deducibility security* (*BD security*), a flexible notion of information-flow security applicable to arbitrary input–output automata. It generalizes Sutherland’s classic notion of nondeducibility [1] by factoring in declassification bounds and triggers—whereas nondeducibility states that, in a system, information cannot flow between specified sources and sinks, BD security indicates upper bounds for the flow and triggers under which these upper bounds are no longer guaranteed.

BD security is introduced and discussed in detail in [2], where an application to the verification of a conference management system is also presented. This formalization only contains the abstract notion and its associated unwinding proof method, as discussed in Sections 4 and 5 from [2].

- [1] D. Sutherland. A model of information. In 9th National Security Conference, pp. 175–183, 1986.
- [2] S. Kanav, P. Lammich and A. Popescu. A Conference Management System with Verified Document Confidentiality. To appear in CAV 2014. Preprint available at <http://www21.in.tum.de/~popescua/pdf/confsys.pdf>

2 Trivia

```

lemma measure-induct2:
fixes meas :: 'a ⇒ 'b ⇒ nat
assumes  $\bigwedge x1\ x2. (\bigwedge y1\ y2. \text{meas } y1\ y2 < \text{meas } x1\ x2 \implies S\ y1\ y2) \implies S\ x1\ x2$ 
shows  $S\ x1\ x2$ 
proof–
  let ?m =  $\lambda\ x1\ x2. \text{meas } (\text{fst } x1\ x2)\ (\text{snd } x1\ x2)$  let ?S =  $\lambda\ x1\ x2. S\ (\text{fst } x1\ x2)\ (\text{snd } x1\ x2)$ 
  have ?S (x1,x2)
  apply(rule measure-induct[of ?m ?S])
  using assms by (metis fst-conv snd-conv)
  thus ?thesis by auto
qed

```

Right cons:

```

abbreviation Rcons (infix ## 70) where xs ## x  $\equiv$  xs @ [x]

```

```

lemma two-singl-Rcons: [a,b] = [a] ## b by auto

```

3 IO Automaton

3.1 Preliminaries

Transitions

```

datatype ('state,'act,'out) trans =
  Trans (srcOf: 'state) (actOf: 'act) (outOf: 'out) (tgtOf: 'state)

```

```

type-synonym ('state,'act,'out) trace = ('state,'act,'out) trans list

```

```

locale IO-Automaton =
fixes istate :: 'state
  and step :: 'state ⇒ 'act ⇒ 'out * 'state
begin

```

3.2 Reachability and invariance

```

inductive reach :: 'state ⇒ bool where
  Istate: reach istate

```

|
Step: $reach\ s \implies reach\ (snd\ (step\ s\ a))$

lemma *reach-PairI*:
assumes $reach\ s$ **and** $step\ s\ a = (ou, s')$
shows $reach\ s'$
by (*metis Step assms assms snd-eqD*)

definition *holdsIstate* :: $('state \Rightarrow bool) \Rightarrow bool$ **where**
holdsIstate $\varphi \equiv \varphi\ ystate$

definition *invar* :: $('state \Rightarrow bool) \Rightarrow bool$ **where**
invar $\varphi \equiv \forall\ s\ a. reach\ s \wedge \varphi\ s \longrightarrow \varphi\ (snd\ (step\ s\ a))$

lemma *holdsIstate-invar*:
assumes h : *holdsIstate* φ **and** i : *invar* φ **and** a : *reach* s
shows $\varphi\ s$
using a **apply** (*induct rule: reach.induct*)
using $h\ i$ **unfolding** *holdsIstate-def invar-def* **by** *auto*

3.3 System traces

definition *out* :: $'state \Rightarrow 'act \Rightarrow 'out$ **where** $out\ s\ a \equiv fst\ (step\ s\ a)$

definition *eff* :: $'state \Rightarrow 'act \Rightarrow 'state$ **where** $eff\ s\ a \equiv snd\ (step\ s\ a)$

primrec *validTrans* :: $('state, 'act, 'out)\ trans \Rightarrow bool$ **where**
validTrans $(Trans\ s\ a\ ou\ s') = (step\ s\ a = (ou, s'))$

lemma *validTrans*:
validTrans $trn =$
 $(step\ (srcOf\ trn)\ (actOf\ trn) = (outOf\ trn, tgtOf\ trn))$
by (*cases trn*) *auto*

inductive *valid* :: $('state, 'act, 'out)\ trace \Rightarrow bool$ **where**
Singl[*simp, intro!*]:
validTrans trn
 \implies
 $valid\ [trn]$
|
Cons[*intro*]:
 $\llbracket validTrans\ trn; tgtOf\ trn = srcOf\ (hd\ tr); valid\ tr \rrbracket$
 \implies
 $valid\ (trn\ \#\ tr)$

inductive-cases *valid-SingleE*[*elim!*]: $valid\ [trn]$

inductive-cases *valid-ConsE*[*elim*]: *valid* (*trn* # *tr*)

inductive *valid2* :: ('*state*, '*act*, '*out*) *trace* \Rightarrow *bool* **where**
Singl[*simp*, *intro!*]:
validTrans *trn*
 \Rightarrow
valid2 [*trn*]
|
Rcons[*intro*]:
[[*valid2* *tr*; *tgtOf* (*last tr*) = *srcOf trn*; *validTrans trn*]]
 \Rightarrow
valid2 (*tr* ## *trn*)

inductive-cases *valid2-SingleE*[*elim!*]: *valid2* [*trn*]
inductive-cases *valid2-RconsE*[*elim*]: *valid2* (*tr* ## *trn*)

lemma *Nil-not-valid*[*simp*]: \neg *valid* []
by (*metis valid.simps neq-Nil-conv*)

lemma *Nil-not-valid2*[*simp*]: \neg *valid2* []
by (*metis valid2.cases append-Nil butlast.simps butlast-snoc not-Cons-self2*)

lemma *valid-Rcons*:
assumes *valid tr* **and** *tgtOf* (*last tr*) = *srcOf trn* **and** *validTrans trn*
shows *valid* (*tr* ## *trn*)
using *assms* **proof**(*induct arbitrary: trn*)
 case (*Cons trn tr trna*)
 thus ?*case* **by** (*cases tr*) (*auto intro: valid.Cons*)
qed(*auto intro: valid.intros*)

lemma *valid-hd-Rcons*[*simp*]:
assumes *valid tr*
shows *hd* (*tr* ## *trn*) = *hd tr*
by (*metis Nil-not-valid assms hd-append*)

lemma *valid2-hd-Rcons*[*simp*]:
assumes *valid2 tr*
shows *hd* (*tr* ## *trn*) = *hd tr*
by (*metis Nil-not-valid2 assms hd-append*)

lemma *valid2-last-Cons*[*simp*]:
assumes *valid2 tr*
shows *last* (*trn* # *tr*) = *last tr*
by (*metis Nil-not-valid2 assms last.simps*)

lemma *valid2-Cons*:
assumes *valid2 tr* **and** *tgtOf trn* = *srcOf* (*hd tr*) **and** *validTrans trn*
shows *valid2* (*trn* # *tr*)
using *assms* **proof**(*induct arbitrary: trn*)

```

  case Singl show ?case
  unfolding two-singl-Rcons using Singl
  by (intro valid2.Rcons) (auto intro: valid2.Singl)
next
  case Rcons show ?case
  unfolding append.append-Cons[symmetric] using Rcons by (intro valid2.Rcons) auto
qed

```

```

lemma valid-valid2: valid = valid2
proof (rule ext, safe)
  fix tr assume valid tr thus valid2 tr
  by (induct) (auto intro: valid2.Singl valid2-Cons)
next
  fix tr assume valid2 tr thus valid tr
  by (induct) (auto intro: valid.Singl valid-Rcons)
qed

```

```

lemmas valid2-valid = valid-valid2[symmetric]

```

```

definition validFrom :: 'state  $\Rightarrow$  ('state,'act,'out) trace  $\Rightarrow$  bool where
validFrom s tr  $\equiv$  tr = []  $\vee$  (valid tr  $\wedge$  srcOf (hd tr) = s)

```

```

lemma validFrom-Nil[simp,intro!]: validFrom s []
unfolding validFrom-def by auto

```

```

lemma validFrom-valid[simp,intro]: valid tr  $\wedge$  srcOf (hd tr) = s  $\implies$  validFrom s tr
unfolding validFrom-def by auto

```

```

lemma validFrom-validTrans[intro]:
assumes validTrans (Trans s a ou s') and validFrom s' tr
shows validFrom s (Trans s a ou s' # tr)
using assms unfolding validFrom-def by auto

```

3.4 Traces versus reachability

```

lemma valid-reach-src-tgt:
assumes valid tr and reach (srcOf (hd tr))
shows reach (tgtOf (last tr))
using assms by induct (auto intro: reach-PairI simp: validTrans)

```

```

lemma valid-init-reach:
assumes valid tr and srcOf (hd tr) = istate
shows reach (tgtOf (last tr))
using valid-reach-src-tgt assms reach.Istate by metis

```

```

lemma Trans-fst-sndI:
valid [Trans s a (fst (step s a)) (snd (step s a))]
by (metis valid.Singl surjective-pairing validTrans.simps)

```

```

lemma reach-init-valid:
assumes reach s
shows
s = istate
∨
(∃ tr. valid tr ∧ srcOf (hd tr) = istate ∧ tgtOf (last tr) = s)
using assms proof induction
case (Step s a)
thus ?case proof(elim disjE exE conjE)
assume s: s = istate
let ?ou = fst (step s a) let ?s' = snd (step s a)
show ?thesis using s
by (intro disjI2 exI[of - [Trans s a ?ou ?s']]) auto
next
fix tr assume v: valid tr and s: srcOf (hd tr) = istate and t: tgtOf (last tr) = s
let ?ou = fst (step s a) let ?s' = snd (step s a)
show ?thesis using v t s
by (intro disjI2 exI[of - tr ## Trans s a ?ou ?s']) (auto intro: valid-Rcons)
qed
qed auto

```

```

lemma reach-validFrom:
assumes reach s'
shows ∃ s tr. s = istate ∧ (s = s' ∨ (validFrom s tr ∧ tgtOf (last tr) = s'))
using reach-init-valid[OF assms] unfolding validFrom-def by auto

```

4 BD Security

4.1 Definition

```

declare Let-def[simp]

```

```

no-notation relcomp (infixr O 75)

```

```

abbreviation never :: ('a ⇒ bool) ⇒ 'a list ⇒ bool where never U ≡ list-all (λ a. ¬ U a)

```

```

function filtermap ::
(('state,'act,'out) trans ⇒ bool) ⇒ (('state,'act,'out) trans ⇒ 'a) ⇒ ('state,'act,'out) trace ⇒ 'a list
where
filtermap pred func [] = []
|
¬ pred trn ⇒ filtermap pred func (trn # tr) = filtermap pred func tr
|
pred trn ⇒ filtermap pred func (trn # tr) = func trn # filtermap pred func tr
by auto (metis list.exhaust)
termination by lexicographic-order

```

```

locale BD-Security = IO-Automaton istate step
  for istate :: 'state and step :: 'state  $\Rightarrow$  'act  $\Rightarrow$  'out  $\times$  'state
  +
fixes
   $\varphi$  :: ('state,'act,'out) trans  $\Rightarrow$  bool and  $f$  :: ('state,'act,'out) trans  $\Rightarrow$  'value
  and
   $\gamma$  :: ('state,'act,'out) trans  $\Rightarrow$  bool and  $g$  :: ('state,'act,'out) trans  $\Rightarrow$  'obs
  and
   $T$  :: ('state,'act,'out) trans  $\Rightarrow$  bool
  and
   $B$  :: 'value list  $\Rightarrow$  'value list  $\Rightarrow$  bool
begin

definition  $V$  :: ('state,'act,'out) trace  $\Rightarrow$  'value list where  $V \equiv \text{filtermap } \varphi f$ 

definition  $O$  :: ('state,'act,'out) trace  $\Rightarrow$  'obs list where  $O \equiv \text{filtermap } \gamma g$ 

lemma V-simps[simp]:
 $V [] = [] \neg \varphi \text{ trn} \Longrightarrow V (\text{trn} \# \text{tr}) = V \text{tr } \varphi \text{ trn} \Longrightarrow V (\text{trn} \# \text{tr}) = f \text{trn} \# V \text{tr}$ 
unfolding V-def by auto

lemma O-simps[simp]:
 $O [] = [] \neg \gamma \text{ trn} \Longrightarrow O (\text{trn} \# \text{tr}) = O \text{tr } \gamma \text{ trn} \Longrightarrow O (\text{trn} \# \text{tr}) = g \text{trn} \# O \text{tr}$ 
unfolding O-def by auto

inductive reachNT:: 'state  $\Rightarrow$  bool where
  Istate: reachNT istate
  |
  Step:
   $\llbracket \text{reachNT } (\text{srcOf } \text{trn}); \text{step } (\text{srcOf } \text{trn}) (\text{actOf } \text{trn}) = (\text{outOf } \text{trn}, \text{tgtOf } \text{trn}); \neg T \text{trn} \rrbracket$ 
   $\Longrightarrow \text{reachNT } (\text{tgtOf } \text{trn})$ 

lemma reachNT-PairI:
assumes reachNT  $s$  and  $\text{step } s a = (ou, s')$  and  $\neg T (\text{Trans } s a ou s')$ 
shows reachNT  $s'$ 
by (metis BD-Security.reachNT.simps assms trans.sel)

lemma reachNT-reach: assumes reachNT  $s$  shows reach  $s$ 
using assms by induct (auto intro: reach.intros, metis reach.Step snd-conv validTrans)

lemma reachNT-stateO-aux:
assumes reachNT  $s$ 
shows  $s = \text{istate} \vee (\exists sh a ou. \text{reach } sh \wedge \text{step } sh a = (ou, s) \wedge \neg T (\text{Trans } sh a ou s))$ 
using assms by induct (clarsimp, metis BD-Security.reachNT-reach trans.exhaust-sel)

lemma reachNT-state-cases[cases set, consumes 1, case-names init step]:
assumes reachNT  $s$ 

```

obtains $s = \text{istate}$
| $sh\ a\ ou$ **where** $\text{reach}\ sh\ step\ sh\ a = (ou, s) \neg T (\text{Trans}\ sh\ a\ ou\ s)$
by (*metis reachNT-stateO-aux[OF assms]*)

definition *invarNT* **where**
 $\text{invarNT}\ Inv \equiv \forall\ s\ a\ ou\ s'. \text{reachNT}\ s \wedge Inv\ s \wedge \neg T (\text{Trans}\ s\ a\ ou\ s') \wedge \text{step}\ s\ a = (ou, s') \longrightarrow Inv\ s'$

lemma *invarNT-disj*:
assumes *invarNT Inv1* **and** *invarNT Inv2*
shows *invarNT* $(\lambda\ s. Inv1\ s \vee Inv2\ s)$
using *assms* **unfolding** *invarNT-def* **by** *blast*

lemma *invarNT-conj*:
assumes *invarNT Inv1* **and** *invarNT Inv2*
shows *invarNT* $(\lambda\ s. Inv1\ s \wedge Inv2\ s)$
using *assms* **unfolding** *invarNT-def* **by** *blast*

lemma *holdsIstate-invarNT*:
assumes $h: \text{holdsIstate}\ Inv$ **and** $i: \text{invarNT}\ Inv$ **and** $a: \text{reachNT}\ s$
shows $Inv\ s$
using a **using** $h\ i$ **unfolding** *holdsIstate-def invarNT-def*
by (*induct rule: reachNT.induct*) (*metis i invarNT-def trans.exhaust-sel*)+

definition *secure* **::** *bool* **where**
 $\text{secure} \equiv$
 $\forall\ tr\ vl\ vl1.$
 $\text{validFrom}\ \text{istate}\ tr \wedge \text{never}\ T\ tr \wedge B\ vl\ vl1 \wedge V\ tr = vl \longrightarrow$
 $(\exists\ tr1. \text{validFrom}\ \text{istate}\ tr1 \wedge O\ tr1 = O\ tr \wedge V\ tr1 = vl1)$

lemma *V-iff-non-φ[simp]*: $V\ (trn\ \# \ tr) = V\ tr \longleftrightarrow \neg\ \varphi\ trn$
by (*cases φ trn*) *auto*

lemma *V-imp-φ*: $V\ (trn\ \# \ tr) = v\ \# \ V\ tr \implies \varphi\ trn$
by (*cases φ trn*) *auto*

lemma *V-imp-Nil*: $V\ (trn\ \# \ tr) = [] \implies V\ tr = []$
by (*metis V-simps list.distinct trans.exhaust*)

lemma *V-iff-Nil[simp]*: $V\ (trn\ \# \ tr) = [] \longleftrightarrow \neg\ \varphi\ trn \wedge V\ tr = []$
by (*metis V-iff-non-φ V-imp-Nil*)

end

4.2 Unwinding proof method

context *BD-Security*
begin

definition *consume* :: ('state,'act,'out) trans \Rightarrow 'value list \Rightarrow 'value list \Rightarrow bool **where**
consume trn vl vl' \equiv
 if φ trn then vl \neq [] \wedge f trn = hd vl \wedge vl' = tl vl
 else vl' = vl

lemma *length-consume*[simp]:
consume trn vl vl' \implies length vl' < Suc (length vl)
unfolding *consume-def* **by** (auto split: if-splits)

lemma *ex-consume- φ* :
assumes $\neg \varphi$ trn
obtains vl' **where** *consume* trn vl vl'
using *assms* **unfolding** *consume-def* **by** auto

lemma *ex-consume-NO*:
assumes vl \neq [] **and** f trn = hd vl
obtains vl' **where** *consume* trn vl vl'
using *assms* **unfolding** *consume-def* **by** (cases φ trn) auto

definition *iaction* **where**
iaction Δ s vl s1 vl1 \equiv
 \exists a1 ou1 s1' vl1'.
 let trn1 = Trans s1 a1 ou1 s1' in
 validTrans trn1 \wedge
 φ trn1 \wedge *consume* trn1 vl1 vl1' \wedge
 $\neg \gamma$ trn1
 \wedge
 Δ s vl s1' vl1'

lemma *iactionI*[intro?]:
assumes step s1 a1 = (ou1, s1') **and** φ (Trans s1 a1 ou1 s1')
and *consume* (Trans s1 a1 ou1 s1') vl1 vl1'
and $\neg \gamma$ (Trans s1 a1 ou1 s1') **and** Δ s vl s1' vl1'
shows *iaction* Δ s vl s1 vl1
unfolding *iaction-def* **using** *assms* **by** auto

definition *match* **where**
match Δ s s1 vl1 a ou s' vl' \equiv
 \exists a1 ou1 s1' vl1'.
 let trn = Trans s a ou s'; trn1 = Trans s1 a1 ou1 s1' in
 validTrans trn1 \wedge
consume trn1 vl1 vl1' \wedge
 γ trn = γ trn1 \wedge (γ trn \longrightarrow g trn = g trn1) \wedge
 Δ s' vl' s1' vl1'

lemma *matchI*[intro?]:
assumes validTrans (Trans s1 a1 ou1 s1')

and *consume* (*Trans* *s1 a1 ou1 s1'*) *vl1 vl1'* **and** γ (*Trans* *s a ou s'*) = γ (*Trans* *s1 a1 ou1 s1'*)
and γ (*Trans* *s a ou s'*) \implies *g* (*Trans* *s a ou s'*) = *g* (*Trans* *s1 a1 ou1 s1'*)
and Δ *s' vl' s1' vl1'*
shows *match* Δ *s s1 vl1 a ou s' vl'*
unfolding *match-def* **using** *assms* **by** *auto*

definition *ignore where*
ignore Δ *s s1 vl1 a ou s' vl'* \equiv
 $\neg \gamma$ (*Trans* *s a ou s'*) \wedge
 Δ *s' vl' s1 vl1*

lemma *ignoreI*[*intro?*]:
assumes $\neg \gamma$ (*Trans* *s a ou s'*) **and** Δ *s' vl' s1 vl1*
shows *ignore* Δ *s s1 vl1 a ou s' vl'*
unfolding *ignore-def* **using** *assms* **by** *auto*

definition *reaction where*
reaction Δ *s vl s1 vl1* \equiv
 \forall *a ou s' vl'*.
let *trn* = *Trans* *s a ou s'* *in*
validTrans *trn* \wedge \neg *T* *trn* \wedge
consume *trn vl vl'*
 \longrightarrow
match Δ *s s1 vl1 a ou s' vl'*
 \vee
ignore Δ *s s1 vl1 a ou s' vl'*

lemma *reactionI*[*intro?*]:
assumes
 \bigwedge *a ou s' vl'*.
 \llbracket *step* *s a* = (*ou*, *s'*); \neg *T* (*Trans* *s a ou s'*);
consume (*Trans* *s a ou s'*) *vl vl'* \rrbracket
 \implies
match Δ *s s1 vl1 a ou s' vl'* \vee *ignore* Δ *s s1 vl1 a ou s' vl'*
shows *reaction* Δ *s vl s1 vl1*
using *assms* **unfolding** *reaction-def* **by** *auto*

definition *exit* :: '*state* \Rightarrow '*value* \Rightarrow *bool* **where**
exit *s v* \equiv \forall *tr trn*. *validFrom* *s* (*tr ## trn*) \wedge *never* *T* (*tr ## trn*) \wedge φ *trn* \longrightarrow *f* *trn* \neq *v*

lemma *exit-coind*:
assumes *K*: *K* *s*
and *I*: \bigwedge *trn*. \llbracket *K* (*srcOf* *trn*); *validTrans* *trn*; \neg *T* *trn* \rrbracket
 \implies (φ *trn* \longrightarrow *f* *trn* \neq *v*) \wedge *K* (*tgtOf* *trn*)
shows *exit* *s v*
using *K* **unfolding** *exit-def* **proof**(*intro allI conjI impI*)
fix *tr trn* **assume** *K* *s* **and** *validFrom* *s* (*tr ## trn*) \wedge *never* *T* (*tr ## trn*) \wedge φ *trn*
thus *f* *trn* \neq *v*

using *I* **unfolding** *validFrom-def* **by** (*induction tr arbitrary: s trn*)
(auto, metis neq-Nil-conv rotate1.simps(2) rotate1-is-Nil-conv valid-ConsE)
qed

definition *noVal* **where**

noVal *K v* \equiv

$\forall s a ou s'. \text{reachNT } s \wedge K s \wedge \text{step } s a = (ou, s') \wedge \varphi (\text{Trans } s a ou s') \longrightarrow f (\text{Trans } s a ou s') \neq v$

lemma *noVal-disj*:

assumes *noVal Inv1 v* **and** *noVal Inv2 v*

shows *noVal* $(\lambda s. \text{Inv1 } s \vee \text{Inv2 } s)$ *v*

using *assms* **unfolding** *noVal-def* **by** *metis*

lemma *noVal-conj*:

assumes *noVal Inv1 v* **and** *noVal Inv2 v*

shows *noVal* $(\lambda s. \text{Inv1 } s \wedge \text{Inv2 } s)$ *v*

using *assms* **unfolding** *noVal-def* **by** *blast*

definition *no φ* **where**

no φ *K* $\equiv \forall s a ou s'. \text{reachNT } s \wedge K s \wedge \text{step } s a = (ou, s') \longrightarrow \neg \varphi (\text{Trans } s a ou s')$

lemma *no φ -noVal*: *no φ* *K* \implies *noVal* *K v*

unfolding *no φ -def* *noVal-def* **by** *auto*

lemma *exitI*[*consumes 2, induct pred: exit*]:

assumes *rs: reachNT s* **and** *K: K s*

and *I*:

$\bigwedge s a ou s'.$

$\llbracket \text{reach } s; \text{reachNT } s; \text{step } s a = (ou, s'); K s \rrbracket$

$\implies (\varphi (\text{Trans } s a ou s') \longrightarrow f (\text{Trans } s a ou s') \neq v) \wedge K s'$

shows *exit s v*

proof–

let *?K* = $\lambda s. \text{reachNT } s \wedge K s$

show *?thesis* **using** *assms* **by** (*intro exit-coind*[*of ?K*])

(*metis BD-Security.reachNT-reach IO-Automaton.validTrans reachNT.Step trans.exhaust-sel*)+

qed

lemma *exitI2*:

assumes *rs: reachNT s* **and** *K: K s*

and *invarNT K* **and** *noVal K v*

shows *exit s v*

proof–

let *?K* = $\lambda s. \text{reachNT } s \wedge K s$

show *?thesis* **using** *assms* **unfolding** *invarNT-def* *noVal-def* **apply**(*intro exit-coind*[*of ?K*])

by *metis* (*metis IO-Automaton.validTrans reachNT.Step trans.exhaust-sel*)

qed

definition *noVal2* **where**

noVal2 $K v \equiv$

$\forall s a ou s'. \text{reachNT } s \wedge K s v \wedge \text{step } s a = (ou, s') \wedge \varphi (\text{Trans } s a ou s') \longrightarrow f (\text{Trans } s a ou s') \neq v$

lemma *noVal2-disj*:

assumes *noVal2* $Inv1 v$ **and** *noVal2* $Inv2 v$

shows *noVal2* $(\lambda s v. Inv1 s v \vee Inv2 s v) v$

using *assms* **unfolding** *noVal2-def* **by** *metis*

lemma *noVal2-conj*:

assumes *noVal2* $Inv1 v$ **and** *noVal2* $Inv2 v$

shows *noVal2* $(\lambda s v. Inv1 s v \wedge Inv2 s v) v$

using *assms* **unfolding** *noVal2-def* **by** *blast*

lemma *noVal-noVal2*: *noVal* $K v \implies \text{noVal2 } (\lambda s v. K s) v$

unfolding *noVal-def* *noVal2-def* **by** *auto*

lemma *exitI-noVal2*[*consumes 2, induct pred: exit*]:

assumes *rs*: *reachNT* s **and** $K: K s v$

and I :

$\bigwedge s a ou s'.$

$\llbracket \text{reach } s; \text{reachNT } s; \text{step } s a = (ou, s'); K s v \rrbracket$

$\implies (\varphi (\text{Trans } s a ou s') \longrightarrow f (\text{Trans } s a ou s') \neq v) \wedge K s' v$

shows *exit* $s v$

proof–

let $?K = \lambda s. \text{reachNT } s \wedge K s v$

show *?thesis* **using** *assms* **by** (*intro exit-coind*[*of ?K*])

(*metis BD-Security.reachNT-reach IO-Automaton.validTrans reachNT.Step trans.exhaust-sel*)+

qed

lemma *exitI2-noVal2*:

assumes *rs*: *reachNT* s **and** $K: K s v$

and *invarNT* $(\lambda s. K s v)$ **and** *noVal2* $K v$

shows *exit* $s v$

proof–

let $?K = \lambda s. \text{reachNT } s \wedge K s v$

show *?thesis* **using** *assms* **unfolding** *invarNT-def* *noVal2-def*

by (*intro exit-coind*[*of ?K*]) (*metis IO-Automaton.validTrans reachNT.Step trans.exhaust-sel*)+

qed

lemma *exit-validFrom*:

assumes $vl: vl \neq []$ **and** $i: \text{exit } s (\text{hd } vl)$ **and** $v: \text{validFrom } s \text{ tr}$ **and** $V: V \text{ tr} = vl$

and $T: \text{never } T \text{ tr}$

shows *False*

using $i v V T$ **proof**(*induction tr arbitrary: s*)

```

case Nil thus ?case by (metis V-simps(1) vl)
next
case (Cons trn tr s)
show ?case
proof(cases  $\varphi$  trn)
  case True
  hence f trn = hd vl using Cons by (metis V-simps(3) hd-Cons-tl list.inject vl)
  moreover have validFrom s [trn] using  $\langle$ validFrom s (trn # tr) $\rangle$ 
  unfolding validFrom-def by auto
  ultimately show ?thesis using Cons True unfolding exit-def
  by (elim allE[of - []]) auto
next
case False
hence V tr = vl using Cons by auto
moreover have never T tr by (metis Cons.prem1 list-all-simps)
moreover from  $\langle$ validFrom s (trn # tr) $\rangle$  have validFrom (tgtOf trn) tr and s: s = srcOf trn
by (metis list.distinct(1) validFrom-def valid-ConsE Cons.prem2)
  IO-Automaton.validFrom-def list.discI list.sel(1)+
moreover have exit (tgtOf trn) (hd vl) using  $\langle$ exit s (hd vl) $\rangle$ 
unfolding exit-def s by simp
  (metis (no-types) Cons.prem2 Cons.prem4 append-Cons list.sel(1)
    list.distinct list-all-simps valid.Cons validFrom-def valid-ConsE)
ultimately show ?thesis using Cons(1) by auto
qed
qed

```

definition unwind **where**

```

unwind  $\Delta \equiv$ 
 $\forall$  s vl s1 vl1.
  reachNT s  $\wedge$  reach s1  $\wedge$   $\Delta$  s vl s1 vl1
   $\longrightarrow$ 
  (vl  $\neq$  []  $\wedge$  exit s (hd vl))
   $\vee$ 
  iaction  $\Delta$  s vl s1 vl1
   $\vee$ 
  ((vl  $\neq$  []  $\vee$  vl1 = [])  $\wedge$  reaction  $\Delta$  s vl s1 vl1)

```

lemma unwindI[*intro?*]:

assumes

```

 $\bigwedge$  s vl s1 vl1.
   $\llbracket$ reachNT s; reach s1;  $\Delta$  s vl s1 vl1 $\rrbracket$ 
   $\implies$ 
  (vl  $\neq$  []  $\wedge$  exit s (hd vl))
   $\vee$ 
  iaction  $\Delta$  s vl s1 vl1
   $\vee$ 
  ((vl  $\neq$  []  $\vee$  vl1 = [])  $\wedge$  reaction  $\Delta$  s vl s1 vl1)

```

shows unwind Δ

using assms **unfolding** unwind-def **by** auto

lemma *unwind-trace*:

assumes *unwind*: *unwind* Δ **and** *reachNT* *s* **and** *reach* *s1* **and** Δ *s vl s1 vl1*

and *validFrom* *s tr* **and** *never* *T tr* **and** *V tr = vl*

shows $\exists tr1. \text{validFrom } s1 \ tr1 \ \wedge \ O \ tr1 = O \ tr \ \wedge \ V \ tr1 = vl1$

proof–

let $?S = \lambda \ tr \ vl1.$

$\forall \ s \ vl \ s1. \text{reachNT } s \ \wedge \ \text{reach } s1 \ \wedge \ \Delta \ s \ vl \ s1 \ vl1 \ \wedge \ \text{validFrom } s \ tr \ \wedge \ \text{never } T \ tr \ \wedge \ V \ tr = vl \ \longrightarrow$

$(\exists tr1. \text{validFrom } s1 \ tr1 \ \wedge \ O \ tr1 = O \ tr \ \wedge \ V \ tr1 = vl1)$

let $?f = \lambda \ tr \ vl1. \text{length } tr + \text{length } vl1$

have $?S \ tr \ vl1$

proof(*induct rule: measure-induct2[of ?f ?S]*)

case $(1 \ tr \ vl1)$

show *?case*

proof(*intro allI impI, elim conjE*)

fix *s vl s1* **assume** *rs: reachNT s* **and** *rs1: reach s1* **and** $\Delta: \Delta \ s \ vl \ s1 \ vl1$

and *v: validFrom s tr* **and** *NT: never T tr* **and** *V: V tr = vl*

hence $(vl \neq [] \ \wedge \ \text{exit } s \ (\text{hd } vl)) \vee$

iaction $\Delta \ s \ vl \ s1 \ vl1 \ \vee$

reaction $\Delta \ s \ vl \ s1 \ vl1 \ \wedge \ \neg \ \text{iaction } \Delta \ s \ vl \ s1 \ vl1)$

(is *?exit* \vee *?iact* \vee *?react* \wedge *-*)

using *unwind unfolding unwind-def by metis*

thus $\exists tr1. \text{validFrom } s1 \ tr1 \ \wedge \ O \ tr1 = O \ tr \ \wedge \ V \ tr1 = vl1$

proof *safe*

assume $vl \neq []$ **and** *exit s (hd vl)*

hence *False* **using** *v V exit-validFrom NT by auto*

thus *?thesis* **by** *auto*

next

assume *?iact*

thus *?thesis* **unfolding** *iaction-def Let-def* **proof** *safe*

fix *a1 :: 'act* **and** *ou1 :: 'out* **and** *s1' :: 'state* **and** *vl1'*

let $?trn1 = \text{Trans } s1 \ a1 \ ou1 \ s1'$

assume *vtrans1: validTrans ?trn1* **and** $\varphi1: \varphi \ ?trn1$

and *c: consume ?trn1 vl1 vl1'* **and** $\gamma: \neg \ \gamma \ ?trn1$ **and** $\Delta: \Delta \ s \ vl \ s1' \ vl1'$

from $\varphi1 \ c$ **obtain** *v1* **where** $vl1: vl1 = v1 \ \# \ vl1'$ **and** $f1: f \ ?trn1 = v1$

unfolding *consume-def* **by** $(\text{cases } vl1)$ *auto*

have *rs1': reach s1'* **using** *rs1 vtrans1* **by** $(\text{auto intro: reach-PairI})$

obtain *tr1* **where** $v1: \text{validFrom } s1' \ tr1$ **and** $O: O \ tr1 = O \ tr$ **and** $V: V \ tr1 = vl1'$

using $1[\text{of } tr \ vl1'] \ rs \ rs1' \ \Delta \ v \ NT \ V$ **unfolding** *vl1* **by** *auto*

show *?thesis*

using $vtrans1 \ v1 \ O \ \gamma \ V \ \varphi1 \ f1$ **unfolding** *vl1* **by** $(\text{intro exI}[\text{of } - \ ?trn1 \ \# \ tr1])$ *auto*

qed

next

assume *react: ?react* **and** *iact: \neg ?iact*

show *?thesis*

proof(*cases tr*)

case *Nil* **note** $tr = Nil$

hence $vl: vl = []$ **using** *V* **by** *simp*

show *?thesis* **proof**(*cases vl1*)

```

    case Nil note vl1 = Nil
    show ?thesis using 1[of tr vl1]  $\Delta$  V NT V unfolding tr vl1 by auto
next
  case Cons
  hence False using vl unwind rs rs1  $\Delta$  iact unfolding unwind-def by auto
  thus ?thesis by auto
qed
next
  case (Cons trn tr') note tr = Cons
  show ?thesis
  proof(cases trn)
    case (Trans ss a ou s') note trn = Trans let ?trn = Trans s a ou s'
    have ss: ss = s using trn v unfolding tr validFrom-def by auto
    have Ta:  $\neg$  T ?trn and s: s = srcOf trn and vtrans: validTrans ?trn
    and v': validFrom s' tr' and NT': never T tr'
    using v NT V unfolding tr validFrom-def trn by auto
    have rs': reachNT s' using rs vtrans Ta by (auto intro: reachNT-PairI)
    {assume  $\varphi$  ?trn hence vl  $\neq$  []  $\wedge$  f ?trn = hd vl using V unfolding tr trn ss by auto
    }
    then obtain vl' where c: consume ?trn vl vl'
    using ex-consume- $\varphi$  ex-consume-NO by metis
    have V': V tr' = vl' using V c unfolding tr trn ss consume-def
    by (cases  $\varphi$  ?trn) (simp-all, metis list.sel(2-3))
    have match  $\Delta$  s s1 vl1 a ou s' vl'  $\vee$  ignore  $\Delta$  s s1 vl1 a ou s' vl' (is ?match  $\vee$  ?ignore)
    using react unfolding reaction-def using vtrans Ta c by auto
    thus ?thesis proof safe
      assume ?match
      thus ?thesis unfolding match-def Let-def proof (elim exE conjE)
        fix a1 :: 'act and ou1 :: 'out and s1' :: 'state and vl1'
        let ?trn1 = Trans s1 a1 ou1 s1'
        assume  $\Delta$ :  $\Delta$  s' vl' s1' vl1' and vtrans1: validTrans ?trn1
        and c1: consume ?trn1 vl1 vl1' and  $\gamma$ :  $\gamma$  ?trn =  $\gamma$  ?trn1
        and g:  $\gamma$  ?trn  $\longrightarrow$  g ?trn = g ?trn1
        have rs1': reach s1' using rs rs1 vtrans vtrans1 by (auto intro: reach-PairI)
        obtain tr1 where v1: validFrom s1' tr1 and O: O tr1 = O tr' and V: V tr1 = vl1'
        using 1[of tr' vl1'] rs' rs1'  $\Delta$  v' NT' V' c1 unfolding tr by auto
        have V (?trn1 # tr1) = vl1
        using c1 V unfolding consume-def by (cases  $\varphi$  ?trn1) auto
        thus ?thesis
        apply(intro exI[of - Trans s1 a1 ou1 s1' # tr1])
        using vtrans1 v1 O  $\gamma$  g V unfolding tr trn ss by auto
      qed
    next
      assume ?ignore
      thus ?thesis unfolding ignore-def Let-def proof (elim exE conjE)
        assume  $\gamma$ :  $\neg$   $\gamma$  ?trn and  $\Delta$ :  $\Delta$  s' vl' s1 vl1
        obtain tr1 where v1: validFrom s1 tr1 and O: O tr1 = O tr' and V: V tr1 = vl1
        using 1[of tr' vl1] rs' rs1  $\Delta$  v' NT' V' c unfolding tr by auto
        show ?thesis

```

```

      apply(intro exI[of - tr1])
      using v1 O V  $\gamma$  unfolding tr trn ss by auto
    qed
  qed
  qed
  qed
  qed
  qed
  qed
  thus ?thesis using assms by auto
qed

```

theorem *unwind-secure*:
assumes *init*: $\bigwedge vl\ vl1. B\ vl\ vl1 \implies \Delta\ istate\ vl\ istate\ vl1$
and *unwind*: *unwind* Δ
shows *secure*
using *assms* *unwind-trace* **unfolding** *secure-def* **by** (*blast* *intro*: *reach.Istate* *reachNT.Istate*)

5 Compositional Reasoning

context *BD-Security* **begin**

5.1 Preliminaries

definition *disjAll* $\Delta s\ s\ vl\ s1\ vl1 \equiv (\exists \Delta \in \Delta s. \Delta\ s\ vl\ s1\ vl1)$

lemma *disjAll-simps[simp]*:
disjAll $\{\}$ $\equiv \lambda - - -. False$
disjAll (*insert* $\Delta\ \Delta s$) $\equiv \lambda s\ vl\ s1\ vl1. \Delta\ s\ vl\ s1\ vl1 \vee disjAll\ \Delta s\ s\ vl\ s1\ vl1$
unfolding *disjAll-def*[*abs-def*] **by** *auto*

lemma *iaction-mono*:
assumes *1*: *iaction* $\Delta\ s\ vl\ s1\ vl1$ **and** *2*: $\bigwedge s\ vl\ s1\ vl1. \Delta\ s\ vl\ s1\ vl1 \implies \Delta'\ s\ vl\ s1\ vl1$
shows *iaction* $\Delta'\ s\ vl\ s1\ vl1$
proof –
obtain *a1* *ou1* *s1'* *vl1'*
where *step* *s1* *a1* = (*ou1*, *s1'*) **and** φ (*Trans* *s1* *a1* *ou1* *s1'*)
and *consume* (*Trans* *s1* *a1* *ou1* *s1'*) *vl1* *vl1'* **and** $\neg \gamma$ (*Trans* *s1* *a1* *ou1* *s1'*)
and $\Delta\ s\ vl\ s1'\ vl1'$ **using** *1* **unfolding** *iaction-def* **by** *auto*
thus ?thesis **unfolding** *iaction-def* **using** *2* **apply** –
by (*rule* *exI*[*of* - *a1*], *rule* *exI*[*of* - *ou1*], *rule* *exI*[*of* - *s1'*], *rule* *exI*[*of* - *vl1'*]) *auto*
qed

lemma *match-mono*:

assumes 1: *match* $\Delta s s1 vl1 a ou s' vl'$ **and** 2: $\bigwedge s vl s1 vl1. \Delta s vl s1 vl1 \implies \Delta' s vl s1 vl1$
shows *match* $\Delta' s s1 vl1 a ou s' vl'$

proof–

obtain $a1 ou1 s1' vl1'$
where $\Delta s' vl' s1' vl1'$
and $step s1 a1 = (ou1, s1')$
and $consume (Trans s1 a1 ou1 s1') vl1 vl1'$
and $\gamma (Trans s a ou s') = \gamma (Trans s1 a1 ou1 s1')$
and $(\gamma (Trans s a ou s') \longrightarrow g (Trans s a ou s')) = g (Trans s1 a1 ou1 s1')$
using 1 **unfolding** *match-def* **by** *auto*
thus *?thesis* **unfolding** *match-def* **using** 2 **apply** –
by (*rule exI[of - a1]*, *rule exI[of - ou1]*, *rule exI[of - s1']*, *rule exI[of - vl1']*) *auto*
qed

lemma *ignore-mono*:

assumes 1: *ignore* $\Delta s s1 vl1 a ou s' vl'$ **and** 2: $\bigwedge s vl s1 vl1. \Delta s vl s1 vl1 \implies \Delta' s vl s1 vl1$
shows *ignore* $\Delta' s s1 vl1 a ou s' vl'$

using *assms* **unfolding** *ignore-def* **by** *auto*

lemma *reaction-mono*:

assumes 1: *reaction* $\Delta s vl s1 vl1$ **and** 2: $\bigwedge s vl s1 vl1. \Delta s vl s1 vl1 \implies \Delta' s vl s1 vl1$
shows *reaction* $\Delta' s vl s1 vl1$

proof

fix $a ou s' vl'$
assume $step s a = (ou, s')$ **and** $\neg T (Trans s a ou s')$ **and** $consume (Trans s a ou s') vl vl'$
hence *match* $\Delta s s1 vl1 a ou s' vl' \vee ignore \Delta s s1 vl1 a ou s' vl'$ (**is** *?m* \vee *?i*)
using 1 **unfolding** *reaction-def* **by** *auto*
thus *match* $\Delta' s s1 vl1 a ou s' vl' \vee ignore \Delta' s s1 vl1 a ou s' vl'$ (**is** *?m'* \vee *?i'*)

proof

assume *?m* **from** *match-mono[OF this 2]* **show** *?thesis* **by** *simp*

next

assume *?i* **from** *ignore-mono[OF this 2]* **show** *?thesis* **by** *simp*

qed

qed

5.2 Decomposition into an arbitrary network of components

definition *unwind-to* **where**

unwind-to $\Delta \Delta s \equiv$

$\forall s vl s1 vl1.$

$reachNT s \wedge reach s1 \wedge \Delta s vl s1 vl1$

\longrightarrow

$vl \neq [] \wedge exit s (hd vl)$

\vee

$iaction (disjAll \Delta s) s vl s1 vl1$

\vee

$(vl \neq [] \vee vl1 = []) \wedge reaction (disjAll \Delta s) s vl s1 vl1$

lemma *unwind-toI[intro?]*:

assumes
 $\bigwedge s \text{ vl } s1 \text{ vl1}.$
 $\llbracket \text{reachNT } s; \text{reach } s1; \Delta \text{ s vl } s1 \text{ vl1} \rrbracket$
 \implies
 $\text{vl} \neq [] \wedge \text{exit } s \text{ (hd vl)}$
 \vee
 $\text{iaction } (\text{disjAll } \Delta s) \text{ s vl } s1 \text{ vl1}$
 \vee
 $(\text{vl} \neq [] \vee \text{vl1} = []) \wedge \text{reaction } (\text{disjAll } \Delta s) \text{ s vl } s1 \text{ vl1}$
shows *unwind-to* $\Delta \Delta s$
using *assms unfolding unwind-to-def by auto*

lemma *unwind-dec*:
assumes *ne*: $\bigwedge \Delta. \Delta \in \Delta s \implies \text{next } \Delta \subseteq \Delta s \wedge \text{unwind-to } \Delta (\text{next } \Delta)$
shows *unwind* $(\text{disjAll } \Delta s)$ (**is** *unwind* $? \Delta$)
proof
fix *s s1* :: 'state **and** *vl vl1* :: 'value list
assume *r*: *reachNT s reach s1* **and** Δ : $? \Delta \text{ s vl } s1 \text{ vl1}$
then obtain Δ **where** Δ : $\Delta \in \Delta s$ **and** Δ : $\Delta \text{ s vl } s1 \text{ vl1}$ **unfolding** *disjAll-def by auto*
let $? \Delta s' = \text{next } \Delta$ **let** $? \Delta' = \text{disjAll } ? \Delta s'$
have $(\text{vl} \neq [] \wedge \text{exit } s \text{ (hd vl)}) \vee$
 $\text{iaction } ? \Delta' \text{ s vl } s1 \text{ vl1} \vee$
 $((\text{vl} \neq [] \vee \text{vl1} = []) \wedge \text{reaction } ? \Delta' \text{ s vl } s1 \text{ vl1})$
using Δ *ne r* **unfolding** *unwind-to-def by auto*
moreover have $\bigwedge s \text{ vl } s1 \text{ vl1}. ? \Delta' \text{ s vl } s1 \text{ vl1} \implies ? \Delta \text{ s vl } s1 \text{ vl1}$
using *ne[OF Δ]* **unfolding** *disjAll-def by auto*
ultimately show
 $(\text{vl} \neq [] \wedge \text{exit } s \text{ (hd vl)}) \vee$
 $\text{iaction } ? \Delta \text{ s vl } s1 \text{ vl1} \vee$
 $((\text{vl} \neq [] \vee \text{vl1} = []) \wedge \text{reaction } ? \Delta \text{ s vl } s1 \text{ vl1})$
using *iaction-mono[of $? \Delta'$ - - - $? \Delta$]* *reaction-mono[of $? \Delta'$ - - - $? \Delta$]* **by** *blast*
qed

lemma *init-dec*:
assumes $\Delta 0$: $\Delta 0 \in \Delta s$
and *i*: $\bigwedge \text{vl } \text{vl1}. B \text{ vl } \text{vl1} \implies \Delta 0 \text{ ivate vl ivate vl1}$
shows $\forall \text{vl } \text{vl1}. B \text{ vl } \text{vl1} \longrightarrow \text{disjAll } \Delta s \text{ ivate vl ivate vl1}$
using *assms unfolding disjAll-def by auto*

theorem *unwind-dec-secure*:
assumes $\Delta 0$: $\Delta 0 \in \Delta s$
and *i*: $\bigwedge \text{vl } \text{vl1}. B \text{ vl } \text{vl1} \implies \Delta 0 \text{ ivate vl ivate vl1}$
and *ne*: $\bigwedge \Delta. \Delta \in \Delta s \implies \text{next } \Delta \subseteq \Delta s \wedge \text{unwind-to } \Delta (\text{next } \Delta)$
shows *secure*
using *init-dec[OF $\Delta 0$ i]* *unwind-dec[OF ne]* *unwind-secure by metis*

5.3 A customization for linear modular reasoning

definition *unwind-cont* **where**

unwind-cont $\Delta \Delta s \equiv$
 $\forall s \text{ vl } s1 \text{ vl1}.$
 $\text{reachNT } s \wedge \text{reach } s1 \wedge \Delta s \text{ vl } s1 \text{ vl1}$
 \longrightarrow
 $\text{iaction } (\text{disjAll } \Delta s) s \text{ vl } s1 \text{ vl1}$
 \vee
 $((\text{vl} \neq [] \vee \text{vl1} = []) \wedge \text{reaction } (\text{disjAll } \Delta s) s \text{ vl } s1 \text{ vl1})$

lemma *unwind-contI*[*intro?*]:

assumes

$\bigwedge s \text{ vl } s1 \text{ vl1}.$
 $\llbracket \text{reachNT } s; \text{reach } s1; \Delta s \text{ vl } s1 \text{ vl1} \rrbracket$
 \implies
 $\text{iaction } (\text{disjAll } \Delta s) s \text{ vl } s1 \text{ vl1}$
 \vee
 $((\text{vl} \neq [] \vee \text{vl1} = []) \wedge \text{reaction } (\text{disjAll } \Delta s) s \text{ vl } s1 \text{ vl1})$

shows *unwind-cont* $\Delta \Delta s$

using *assms unfolding unwind-cont-def* **by** *auto*

definition *unwind-exit* **where**

unwind-exit $\Delta e \equiv$
 $\forall s \text{ vl } s1 \text{ vl1}.$
 $\text{reachNT } s \wedge \text{reach } s1 \wedge \Delta e s \text{ vl } s1 \text{ vl1}$
 \longrightarrow
 $\text{vl} \neq [] \wedge \text{exit } s (\text{hd } \text{vl})$

lemma *unwind-exitI*[*intro?*]:

assumes

$\bigwedge s \text{ vl } s1 \text{ vl1}.$
 $\llbracket \text{reachNT } s; \text{reach } s1; \Delta e s \text{ vl } s1 \text{ vl1} \rrbracket$
 \implies
 $\text{vl} \neq [] \wedge \text{exit } s (\text{hd } \text{vl})$

shows *unwind-exit* Δe

using *assms unfolding unwind-exit-def* **by** *auto*

fun *allConsec* :: *'a list* \Rightarrow (*'a * 'a*) *set* **where**

$\text{allConsec } [] = \{\}$
 $| \text{allConsec } [a] = \{\}$
 $| \text{allConsec } (a \# b \# as) = \text{insert } (a,b) (\text{allConsec } (b\#as))$

lemma *set-allConsec*:

assumes $\Delta \in \text{set } \Delta s'$ **and** $\Delta s = \Delta s' \#\# \Delta 1$

shows $\exists \Delta 2. (\Delta, \Delta 2) \in \text{allConsec } \Delta s$

using *assms proof* (*induction* $\Delta s'$ *arbitrary*: Δs)

case *Nil* **thus** *?case* **by** *auto*

next

```

case (Cons  $\Delta_3$   $\Delta s'$   $\Delta s$ )
show ?case proof(cases  $\Delta = \Delta_3$ )
  case True
  show ?thesis proof(cases  $\Delta s'$ )
    case Nil
    show ?thesis unfolding  $\langle \Delta s = (\Delta_3 \# \Delta s') \#\# \Delta_1 \rangle$  Nil True by (rule exI[of -  $\Delta_1$ ]) simp
  next
  case (Cons  $\Delta_4$   $\Delta s''$ )
  show ?thesis unfolding  $\langle \Delta s = (\Delta_3 \# \Delta s') \#\# \Delta_1 \rangle$  Cons True by (rule exI[of -  $\Delta_4$ ]) simp
qed
next
case False hence  $\Delta \in \text{set } \Delta s'$  using Cons by auto
then obtain  $\Delta_2$  where  $(\Delta, \Delta_2) \in \text{allConsec } (\Delta s' \#\# \Delta_1)$  using Cons by auto
thus ?thesis unfolding  $\langle \Delta s = (\Delta_3 \# \Delta s') \#\# \Delta_1 \rangle$  by (intro exI[of -  $\Delta_2$ ]) (cases  $\Delta s'$ , auto)
qed
qed

```

lemma *allConsec-set*:
assumes $(\Delta_1, \Delta_2) \in \text{allConsec } \Delta s$
shows $\Delta_1 \in \text{set } \Delta s \wedge \Delta_2 \in \text{set } \Delta s$
using *assms* **by** (induct Δs rule: *allConsec.induct*) auto

theorem *unwind-decomp-secure*:
assumes $n: \Delta s \neq []$
and $i: \bigwedge vl\ vl1. B\ vl\ vl1 \implies \text{hd } \Delta s\ \text{istate}\ vl\ \text{istate}\ vl1$
and $c: \bigwedge \Delta_1\ \Delta_2. (\Delta_1, \Delta_2) \in \text{allConsec } \Delta s \implies \text{unwind-cont } \Delta_1\ \{\Delta_1, \Delta_2, \Delta e\}$
and $l: \text{unwind-cont } (\text{last } \Delta s)\ \{\text{last } \Delta s, \Delta e\}$
and $e: \text{unwind-exit } \Delta e$
shows *secure*
proof–
let $?\Delta_0 = \text{hd } \Delta s$ **let** $?\Delta s = \text{insert } \Delta e\ (\text{set } \Delta s)$
define *next* **where** $\text{next } \Delta_1 =$
 (*if* $\Delta_1 = \Delta e$ *then* $\{\}$
 else if $\Delta_1 = \text{last } \Delta s$ *then* $\{\Delta_1, \Delta e\}$
 else $\{\Delta_1, \text{SOME } \Delta_2. (\Delta_1, \Delta_2) \in \text{allConsec } \Delta s, \Delta e\}$) **for** Δ_1
show ?thesis
proof(rule *unwind-dec-secure*)
show $?\Delta_0 \in ?\Delta s$ **using** n **by** auto
next
fix $vl\ vl1$ **assume** $B\ vl\ vl1$
thus $?\Delta_0\ \text{istate}\ vl\ \text{istate}\ vl1$ **by** *fact*
next
fix Δ
assume $1: \Delta \in ?\Delta s$ **show** $\text{next } \Delta \subseteq ?\Delta s \wedge \text{unwind-to } \Delta\ (\text{next } \Delta)$
proof–
 {**assume** $\Delta = \Delta e$
 hence ?thesis **using** e **unfolding** *next-def* *unwind-exit-def* *unwind-to-def* **by** auto
 }
}

```

moreover
{assume  $\Delta = \text{last } \Delta s$  and  $\Delta \neq \Delta e$ 
  hence ?thesis using n l unfolding next-def unwind-cont-def unwind-to-def by simp
}
moreover
{assume 1:  $\Delta \in \text{set } \Delta s$  and 2:  $\Delta \neq \text{last } \Delta s$   $\Delta \neq \Delta e$ 
  then obtain  $\Delta' \Delta s'$  where  $\Delta s: \Delta s = \Delta s' \#\# \Delta'$  and  $\Delta: \Delta \in \text{set } \Delta s'$ 
  by (metis (no-types) append-Cons append-assoc in-set-conv-decomp last-snoc rev-exhaust)
  have  $\exists \Delta 2. (\Delta, \Delta 2) \in \text{allConsec } \Delta s$  using set-allConsec[OF \Delta \Delta s] .
  hence  $(\Delta, \text{SOME } \Delta 2. (\Delta, \Delta 2) \in \text{allConsec } \Delta s) \in \text{allConsec } \Delta s$  by (metis (lifting) someI-ex)
  hence ?thesis using 1 2 c unfolding next-def unwind-cont-def unwind-to-def
  by simp (metis (no-types) allConsec-set)
}
ultimately show ?thesis using 1 by blast
qed
qed
qed

```

5.4 Instances

corollary *unwind-decomp3-secure*:

assumes

i: $\bigwedge vl vl1. B vl vl1 \implies \Delta 1 \text{ istate } vl \text{ istate } vl1$

and *c1*: *unwind-cont* $\Delta 1 \{\Delta 1, \Delta 2, \Delta e\}$

and *c2*: *unwind-cont* $\Delta 2 \{\Delta 2, \Delta 3, \Delta e\}$

and *l*: *unwind-cont* $\Delta 3 \{\Delta 3, \Delta e\}$

and *e*: *unwind-exit* Δe

shows *secure*

apply(*rule* *unwind-decomp-secure*[*of* $[\Delta 1, \Delta 2, \Delta 3] \Delta e$])

using *assms* **by** *auto*

corollary *unwind-decomp4-secure*:

assumes

i: $\bigwedge vl vl1. B vl vl1 \implies \Delta 1 \text{ istate } vl \text{ istate } vl1$

and *c1*: *unwind-cont* $\Delta 1 \{\Delta 1, \Delta 2, \Delta e\}$

and *c2*: *unwind-cont* $\Delta 2 \{\Delta 2, \Delta 3, \Delta e\}$

and *c3*: *unwind-cont* $\Delta 3 \{\Delta 3, \Delta 4, \Delta e\}$

and *l*: *unwind-cont* $\Delta 4 \{\Delta 4, \Delta e\}$

and *e*: *unwind-exit* Δe

shows *secure*

apply(*rule* *unwind-decomp-secure*[*of* $[\Delta 1, \Delta 2, \Delta 3, \Delta 4] \Delta e$])

using *assms* **by** *auto*

corollary *unwind-decomp5-secure*:

assumes

i: $\bigwedge vl vl1. B vl vl1 \implies \Delta 1 \text{ istate } vl \text{ istate } vl1$

and *c1*: *unwind-cont* $\Delta 1 \{\Delta 1, \Delta 2, \Delta e\}$

and *c2*: *unwind-cont* $\Delta 2 \{\Delta 2, \Delta 3, \Delta e\}$

and *c3*: *unwind-cont* $\Delta 3 \{\Delta 3, \Delta 4, \Delta e\}$

```
and  $c_4$ : unwind-cont  $\Delta_4$  { $\Delta_4$ ,  $\Delta_5$ ,  $\Delta e$ }  
and  $l$ : unwind-cont  $\Delta_5$  { $\Delta_5$ ,  $\Delta e$ }  
and  $e$ : unwind-exit  $\Delta e$   
shows secure  
apply(rule unwind-decomp-secure[of [ $\Delta_1$ ,  $\Delta_2$ ,  $\Delta_3$ ,  $\Delta_4$ ,  $\Delta_5$ ]  $\Delta e$ ])  
using assms by auto
```