

A Formalization of Declassification with WHAT&WHERE-Security

Sylvia Grewe, Alexander Lux, Heiko Mantel, Jens Sauer

December 14, 2021

Abstract

Research in information-flow security aims at developing methods to identify undesired information leaks within programs from private sources to public sinks. Noninterference captures this intuition by requiring that no information whatsoever flows from private sources to public sinks. However, in practice this definition is often too strict: Depending on the intuitive desired security policy, the controlled declassification of certain private information (WHAT) at certain points in the program (WHERE) might not result in an undesired information leak.

We present an Isabelle/HOL formalization of such a security property for controlled declassification, namely WHAT&WHERE-security from [2]. The formalization includes compositionality proofs for and a soundness proof for a security type system that checks for programs in a simple while language with dynamic thread creation.

Our formalization of the security type system is abstract in the language for expressions and in the semantic side conditions for expressions. It can easily be instantiated with different syntactic approximations for these side conditions. The soundness proof of such an instantiation boils down to showing that these syntactic approximations imply the semantic side conditions.

This Isabelle/HOL formalization uses theories from the entry Strong-Security (see proof document for details).

Contents

| | | |
|----------|--|----------|
| 1 | Preliminary definitions | 2 |
| 1.1 | Type synonyms | 2 |
| 2 | WHAT&WHERE-security | 4 |
| 2.1 | Definition of WHAT&WHERE-security | 4 |
| 2.2 | Proof technique for compositionality results | 7 |
| 2.3 | Proof of parallel compositionality | 11 |

| | | |
|----------|---|-----------|
| 3 | Example language and compositionality proofs | 17 |
| 3.1 | Example language with dynamic thread creation | 17 |
| 3.2 | Proofs of atomic compositionality results | 21 |
| 3.3 | Proofs of non-atomic compositionality results | 26 |
| 4 | Security type system | 49 |
| 4.1 | Abstract security type system with soundness proof | 49 |
| 4.2 | Example language for Boolean and arithmetic expressions . . . | 52 |
| 4.3 | Example interpretation of abstract security type system . . . | 53 |

1 Preliminary definitions

1.1 Type synonyms

The formalization is parametric in different aspects. Notably, it is parametric in the security lattice it supports.

For better readability, we use the following type synonyms in our formalization (from the entry Strong-Security):

```
theory Types
imports Main
begin
```

- type parameters:
- 'exp: expressions (arithmetic, boolean...)
- 'val: values
- 'id: identifier names
- 'com: commands
- 'd: domains

This is a collection of type synonyms. Note that not all of these type synonyms are used within Strong-Security - some are used in WHATandWHERE-Security.

```
type-synonym ('id, 'val) State = 'id  $\Rightarrow$  'val
```

- type for evaluation functions mapping expressions to a values depending on a state

```
type-synonym ('exp, 'id, 'val) Evalfunction =
  'exp  $\Rightarrow$  ('id, 'val) State  $\Rightarrow$  'val
```

- define configurations with threads as pair of commands and states

```
type-synonym ('id, 'val, 'com) TConfig = 'com  $\times$  ('id, 'val) State
```

- define configurations with thread pools as pair of command lists (thread pool) and states

```
type-synonym ('id, 'val, 'com) TPConfig =
```

$(\text{'com list}) \times (\text{'id, 'val}) \text{ State}$

— type for program states (including the set of commands and a symbol for terminating - None)

type-synonym $\text{'com ProgramState} = \text{'com option}$

— type for configurations with program states

type-synonym $(\text{'id, 'val, 'com}) \text{ PConfig} =$
 $\text{'com ProgramState} \times (\text{'id, 'val}) \text{ State}$

— type for labels with a list of spawned threads

type-synonym $\text{'com Label} = \text{'com list}$

— type for step relations from single commands to a program state, with a label

type-synonym $(\text{'exp, 'id, 'val, 'com}) \text{ TLSteps} =$
 $((\text{'id, 'val, 'com}) \text{ TConfig} \times \text{'com Label}$
 $\times (\text{'id, 'val, 'com}) \text{ PConfig}) \text{ set}$

— curried version of previously defined type

type-synonym $(\text{'exp, 'id, 'val, 'com}) \text{ TLSteps-curry} =$
 $\text{'com} \Rightarrow (\text{'id, 'val}) \text{ State} \Rightarrow \text{'com Label} \Rightarrow \text{'com ProgramState}$
 $\Rightarrow (\text{'id, 'val}) \text{ State} \Rightarrow \text{bool}$

— type for step relations from thread pools to thread pools

type-synonym $(\text{'exp, 'id, 'val, 'com}) \text{ TPSteps} =$
 $((\text{'id, 'val, 'com}) \text{ TPCConfig} \times (\text{'id, 'val, 'com}) \text{ TPCConfig}) \text{ set}$

— curried version of previously defined type

type-synonym $(\text{'exp, 'id, 'val, 'com}) \text{ TPSteps-curry} =$
 $\text{'com list} \Rightarrow (\text{'id, 'val}) \text{ State} \Rightarrow \text{'com list} \Rightarrow (\text{'id, 'val}) \text{ State} \Rightarrow \text{bool}$

— define type of step relations for single threads to thread pools

type-synonym $(\text{'exp, 'id, 'val, 'com}) \text{ TSteps} =$
 $((\text{'id, 'val, 'com}) \text{ TConfig} \times (\text{'id, 'val, 'com}) \text{ TPCConfig}) \text{ set}$

— define the same type as TSteps, but in a curried version (allowing syntax abbreviations)

type-synonym $(\text{'exp, 'id, 'val, 'com}) \text{ TSteps-curry} =$
 $\text{'com} \Rightarrow (\text{'id, 'val}) \text{ State} \Rightarrow \text{'com list} \Rightarrow (\text{'id, 'val}) \text{ State} \Rightarrow \text{bool}$

— type for simple domain assignments; 'd has to be an instance of order (partial order)

type-synonym $(\text{'id, 'd}) \text{ DomainAssignment} = \text{'id} \Rightarrow \text{'d::order}$

type-synonym $\text{'com Bisimulation-type} = ((\text{'com list}) \times (\text{'com list})) \text{ set}$

— type for escape hatches

type-synonym $(\text{'d, 'exp}) \text{ Hatch} = \text{'d} \times \text{'exp}$

— type for sets of escape hatches
type-synonym ('d, 'exp) *Hatches* = (('d, 'exp) *Hatch*) *set*

— type for local escape hatches
type-synonym ('d, 'exp) *lHatch* = 'd × 'exp × *nat*

— type for sets of local escape hatches
type-synonym ('d, 'exp) *lHatches* = (('d, 'exp) *lHatch*) *set*

end

2 WHAT&WHERE-security

2.1 Definition of WHAT&WHERE-security

The definition of WHAT&WHERE-security is parametric in a security lattice ('d) and in a programming language ('com).

theory *WHATWHERE-Security*
imports *Strong-Security.Types*
begin

locale *WHATWHERE* =
fixes *SR* :: ('exp, 'id, 'val, 'com) *TLSteps*
and *E* :: ('exp, 'id, 'val) *Evalfunction*
and *pp* :: 'com ⇒ *nat*
and *DA* :: ('id, 'd::order) *DomainAssignment*
and *lH* :: ('d::order, 'exp) *lHatches*

begin

— define when two states are indistinguishable for an observer on domain d

definition *d-equal* :: 'd::order ⇒ ('id, 'val) *State*
⇒ ('id, 'val) *State* ⇒ *bool*

where

$d\text{-equal } d \ m \ m' \equiv \forall x. ((DA \ x) \leq d \longrightarrow (m \ x) = (m' \ x))$

abbreviation *d-equal'* :: ('id, 'val) *State*
⇒ 'd::order ⇒ ('id, 'val) *State* ⇒ *bool*
((- =_ -))

where

$m =_d m' \equiv d\text{-equal } d \ m \ m'$

— transitivity of d-equality

lemma *d-equal-trans*:

$\llbracket m =_d m'; m' =_d m'' \rrbracket \Longrightarrow m =_d m''$

by (*simp add: d-equal-def*)

abbreviation $SRabbr :: ('exp, 'id, 'val, 'com) TLSteps\text{-}curry$
 $((1\langle -,/- \rangle) \rightarrow \triangleleft - \triangleright / (1\langle -,/- \rangle)) [0,0,0,0,0] 81)$
where
 $\langle c,m \rangle \rightarrow \triangleleft \alpha \triangleright \langle p,m' \rangle \equiv ((c,m),\alpha,(p,m')) \in SR$

— function for obtaining the unique memory (state) after one step for a command and a memory (state)

definition $NextMem :: 'com \Rightarrow ('id, 'val) State \Rightarrow ('id, 'val) State$
 $(\llbracket - \rrbracket'(-))$
where
 $\llbracket c \rrbracket(m) \equiv (THE\ m'. (\exists p\ \alpha. \langle c,m \rangle \rightarrow \triangleleft \alpha \triangleright \langle p,m' \rangle))$

— function getting all escape hatches for some location

definition $htchLoc :: nat \Rightarrow ('d, 'exp) Hatches$
where
 $htchLoc\ \iota \equiv \{(d,e). (d,e,\iota) \in lH\}$

— function for getting all escape hatches for some set of locations

definition $htchLocSet :: nat\ set \Rightarrow ('d, 'exp) Hatches$
where
 $htchLocSet\ PP \equiv \bigcup \{h. (\exists \iota \in PP. h = htchLoc\ \iota)\}$

— predicate for (d,H)-equality

definition $dH\text{-}equal :: 'd \Rightarrow ('d, 'exp) Hatches$
 $\Rightarrow ('id, 'val) State \Rightarrow ('id, 'val) State \Rightarrow bool$
where
 $dH\text{-}equal\ d\ H\ m\ m' \equiv (m =_d m' \wedge$
 $(\forall (d',e) \in H. (d' \leq d \longrightarrow (E\ e\ m = E\ e\ m'))))$

abbreviation $dH\text{-}equal' :: ('id, 'val) State \Rightarrow 'd \Rightarrow ('d, 'exp) Hatches$
 $\Rightarrow ('id, 'val) State \Rightarrow bool$
 $((-\ \sim_{-,H}\ -))$

where
 $m \sim_{d,H} m' \equiv dH\text{-}equal\ d\ H\ m\ m'$

— predicate indicating that a command is not a d-declassification command

definition $NDC :: 'd \Rightarrow 'com \Rightarrow bool$
where
 $NDC\ d\ c \equiv (\forall m\ m'. m =_d m' \longrightarrow \llbracket c \rrbracket(m) =_d \llbracket c \rrbracket(m'))$

— predicate indicating an 'immediate d-declassification command' for a set of escape hatches

definition $IDC :: 'd \Rightarrow 'com \Rightarrow ('d, 'exp) Hatches \Rightarrow bool$
where
 $IDC\ d\ c\ H \equiv (\exists m\ m'. m =_d m' \wedge$
 $(\neg \llbracket c \rrbracket(m) =_d \llbracket c \rrbracket(m'))))$

$$\wedge (\forall m m'. m \sim_{d,H} m' \longrightarrow \llbracket c \rrbracket(m) =_d \llbracket c \rrbracket(m'))$$

definition *stepResultsinR* :: 'com ProgramState \Rightarrow 'com ProgramState
 \Rightarrow 'com Bisimulation-type \Rightarrow bool

where

$$\text{stepResultsinR } p \ p' \ R \equiv (p = \text{None} \wedge p' = \text{None}) \vee \\ (\exists c \ c'. (p = \text{Some } c \wedge p' = \text{Some } c' \wedge ([c], [c']) \in R))$$

definition *dhequality-alternative* :: 'd \Rightarrow nat set \Rightarrow nat
 \Rightarrow ('id, 'val) State \Rightarrow ('id, 'val) State \Rightarrow bool

where

$$\text{dhequality-alternative } d \ PP \ \iota \ m \ m' \equiv m \sim_{d,(\text{htchLocSet } PP)} m' \vee \\ (\neg (\text{htchLoc } \iota) \subseteq (\text{htchLocSet } PP))$$

definition *Strong-dlHPP-Bisimulation* :: 'd \Rightarrow nat set
 \Rightarrow 'com Bisimulation-type \Rightarrow bool

where

$$\text{Strong-dlHPP-Bisimulation } d \ PP \ R \equiv \\ (\text{sym } R) \wedge (\text{trans } R) \wedge \\ (\forall (V, V') \in R. \text{length } V = \text{length } V') \wedge \\ (\forall (V, V') \in R. \forall i < \text{length } V. \\ ((\text{NDC } d \ (V!i)) \vee \\ (\text{IDC } d \ (V!i) \ (\text{htchLoc } (\text{pp } (V!i)))))) \wedge \\ (\forall (V, V') \in R. \forall i < \text{length } V. \forall m1 \ m1' \ m2 \ \alpha \ p. \\ (\langle V!i, m1 \rangle \rightarrow \triangleleft \alpha \triangleright \langle p, m2 \rangle \wedge m1 \sim_{d,(\text{htchLocSet } PP)} m1') \\ \longrightarrow (\exists p' \ \alpha' \ m2'. (\langle V!i, m1' \rangle \rightarrow \triangleleft \alpha' \triangleright \langle p', m2' \rangle \wedge \\ (\text{stepResultsinR } p \ p' \ R) \wedge (\alpha, \alpha') \in R \wedge \\ (\text{dhequality-alternative } d \ PP \ (\text{pp } (V!i)) \ m2 \ m2'))))$$

— predicate to define when a program is strongly secure

definition *WHATWHERE-Secure* :: 'com list \Rightarrow bool

where

$$\text{WHATWHERE-Secure } V \equiv (\forall d \ PP. \\ (\exists R. \text{Strong-dlHPP-Bisimulation } d \ PP \ R \wedge (V, V) \in R))$$

— auxiliary lemma to obtain central strong (d,H,PP)-Bisimulation property as Lemma in meta logic (allows instantiating all the variables manually if necessary)

lemma *strongdlHPPB-aux*:

$$\wedge V \ V' \ m1 \ m1' \ m2 \ p \ i \ \alpha. \llbracket \text{Strong-dlHPP-Bisimulation } d \ PP \ R; \\ i < \text{length } V; (V, V') \in R; \\ \langle V!i, m1 \rangle \rightarrow \triangleleft \alpha \triangleright \langle p, m2 \rangle; m1 \sim_{d,(\text{htchLocSet } PP)} m1' \rrbracket \\ \Longrightarrow (\exists p' \ \alpha' \ m2'. \langle V!i, m1' \rangle \rightarrow \triangleleft \alpha' \triangleright \langle p', m2' \rangle \\ \wedge \text{stepResultsinR } p \ p' \ R \wedge (\alpha, \alpha') \in R \wedge \\ (\text{dhequality-alternative } d \ PP \ (\text{pp } (V!i)) \ m2 \ m2')) \\ \text{by (simp add: Strong-dlHPP-Bisimulation-def, fastforce)}$$

— auxiliary lemma to obtain 'NDC or IDC' from strong (d,lH,PP)-Bisimulation as lemma in meta logic allowing instantiation of the variables

lemma *strongdlHPPB-NDCIDCaux*:

$\bigwedge V V' i. \llbracket \text{Strong-dlHPP-Bisimulation } d \text{ PP } R; \text{ (} V, V') \in R; i < \text{length } V \rrbracket$
 $\implies (\text{NDC } d (V!i) \vee \text{IDC } d (V!i) (\text{htchLoc } (pp (V!i))))$
by (*simp add: Strong-dlHPP-Bisimulation-def, auto*)

lemma *WHATWHERE-empty*:

WHATWHERE-Secure \square
by (*simp add: WHATWHERE-Secure-def, auto,*
rule-tac x={([],[])} in exI,
simp add: Strong-dlHPP-Bisimulation-def sym-def trans-def)

end

end

2.2 Proof technique for compositionality results

For proving compositionality results for WHAT&WHERE-security, we formalize the following “up-to technique” and prove it sound:

theory *Up-To-Technique*

imports *WHATWHERE-Security*

begin

context *WHATWHERE*

begin

abbreviation *SdlHPPB* **where** *SdlHPPB* \equiv *Strong-dlHPP-Bisimulation*

— define the ‘reflexive part’ of a relation (sets of elements which are related with themselves by the given relation)

definition *Areft* $:: ('a \times 'a) \text{ set} \Rightarrow 'a \text{ set}$

where

Areft *R* = {*e.* (*e,e*) \in *R*}

lemma *commonAreft-subset-commonDomain*:

$(\text{Areft } R1 \cap \text{Areft } R2) \subseteq (\text{Domain } R1 \cap \text{Domain } R2)$

by (*simp add: Areft-def, auto*)

— define disjoint strong (d,lH,PP)-bisimulation up-to-R’ for a relation R

definition *disj-dlHPP-Bisimulation-Up-To-R'* $::$

$'d \Rightarrow \text{nat set} \Rightarrow 'com \text{ Bisimulation-type}$

$\Rightarrow 'com \text{ Bisimulation-type} \Rightarrow \text{bool}$

where

disj-dlHPP-Bisimulation-Up-To-R' *d PP R' R* \equiv

$$\begin{aligned}
& SdlHPPB \ d \ PP \ R' \wedge (sym \ R) \wedge (trans \ R) \\
& \wedge (\forall (V, V') \in R. length \ V = length \ V') \wedge \\
& (\forall (V, V') \in R. \forall i < length \ V. \\
& \quad ((NDC \ d \ (V!i)) \vee \\
& \quad (IDC \ d \ (V!i) \ (htchLoc \ (pp \ (V!i)))))) \wedge \\
& (\forall (V, V') \in R. \forall i < length \ V. \forall m1 \ m1' \ m2 \ \alpha \ p. \\
& \quad (\langle V!i, m1 \rangle \rightarrow \triangleleft \alpha \triangleright \langle p, m2 \rangle \wedge m1 \sim_{d, (htchLocSet \ PP)} m1') \\
& \quad \rightarrow (\exists p' \ \alpha' \ m2'. \langle V!i, m1' \rangle \rightarrow \triangleleft \alpha' \triangleright \langle p', m2' \rangle \wedge \\
& \quad \quad (stepResultsinR \ p \ p' \ (R \cup R')) \wedge (\alpha, \alpha') \in (R \cup R') \wedge \\
& \quad \quad (dhequality-alternative \ d \ PP \ (pp \ (V!i)) \ m2 \ m2'))
\end{aligned}$$

— lemma about the transitivity of the union of symmetric and transitive relations under certain circumstances

lemma *trans-RuR'*:

assumes *transR*: *trans R*
assumes *symR*: *sym R*
assumes *transR'*: *trans R'*
assumes *symR'*: *sym R'*
assumes *eqlenR*: $\forall (V, V') \in R. length \ V = length \ V'$
assumes *eqlenR'*: $\forall (V, V') \in R'. length \ V = length \ V'$
assumes *Areflassump*: $(Arefl \ R \cap Arefl \ R') \subseteq \{\emptyset\}$
shows *trans* $(R \cup R')$

proof —

```

{
  fix V V' V''
  assume p1: (V, V') ∈ (R ∪ R')
  assume p2: (V', V'') ∈ (R ∪ R')

  from p1 p2 have (V, V'') ∈ (R ∪ R')
  proof (auto)
    assume inR1: (V, V') ∈ R
    assume inR2: (V', V'') ∈ R
    from inR1 inR2 transR show (V, V'') ∈ R
      unfolding trans-def
      by blast
  next
    assume inR'1: (V, V') ∈ R'
    assume inR'2: (V', V'') ∈ R'
    assume notinR': (V, V'') ∉ R'
    from inR'1 inR'2 transR' have inR: (V, V'') ∈ R
      unfolding trans-def
      by blast
    from notinR' inR have False
      by auto
    thus (V, V'') ∈ R ..
  next
    assume inR1: (V, V') ∈ R
    assume inR'2: (V', V'') ∈ R'
    from inR1 symR transR have (V, V) ∈ R ∧ (V', V') ∈ R

```



```

    unfolding sym-def trans-def
    by blast
  hence AreflR:  $\{V, V'\} \subseteq \text{Arefl } R$  by (simp add: Arefl-def)
  from inR'2 symR' transR' have  $(V', V') \in R' \wedge (V'', V'') \in R'$ 
    unfolding sym-def trans-def
    by blast
  hence AreflR':  $\{V', V''\} \subseteq \text{Arefl } R'$  by (simp add: Arefl-def)

  from AreflR AreflR' Areflassump have V'empt:  $V' = \square$ 
    by (simp add: Arefl-def, blast)
  with inR'2 eqLenR' have  $V' = V''$  by auto
  with inR1 show  $(V, V'') \in R$  by auto
next
  assume inR'1:  $(V, V') \in R'$ 
  assume inR2:  $(V', V'') \in R$ 
  from inR'1 symR' transR' have  $(V, V) \in R' \wedge (V', V') \in R'$ 
    unfolding sym-def trans-def
    by blast
  hence AreflR':  $\{V, V'\} \subseteq \text{Arefl } R'$  by (simp add: Arefl-def)
  from inR2 symR transR have  $(V', V') \in R \wedge (V'', V'') \in R$ 
    unfolding sym-def trans-def
    by blast
  hence AreflR:  $\{V', V''\} \subseteq \text{Arefl } R$  by (simp add: Arefl-def)

  from AreflR AreflR' Areflassump have V'empt:  $V' = \square$ 
    by (simp add: Arefl-def, blast)
  with inR'1 eqLenR' have  $V' = V$  by auto
  with inR2 show  $(V, V'') \in R$  by auto
qed
}
thus ?thesis unfolding trans-def by force

```

qed

lemma *Up-To-Technique*:

$\llbracket \text{disj-dlHPP-Bisimulation-Up-To-}R' \text{ d PP } R' R; \text{Arefl } R \cap \text{Arefl } R' \subseteq \{\square\} \rrbracket$

$\implies \text{SdlHPPB d PP } (R \cup R')$

proof (simp add: disj-dlHPP-Bisimulation-Up-To- R' -def

Strong-dlHPP-Bisimulation-def, auto)

assume symR': sym R'

assume symR: sym R

with symR' show sym $(R \cup R')$

by (simp add: sym-def)

next

assume symR': sym R'

assume symR: sym R

assume transR': trans R'

assume transR: trans R

assume $eqlenR'$: $\forall (V, V') \in R'. \text{length } V = \text{length } V'$
assume $eqlenR$: $\forall (V, V') \in R. \text{length } V = \text{length } V'$
assume $areflassump$: $\text{Areft } R \cap \text{Areft } R' \subseteq \{\emptyset\}$
from $\text{sym}R' \text{ sym}R \text{ trans}R' \text{ trans}R \text{ eqlen}R' \text{ eqlen}R \text{ areflassump } \text{trans-Ru}R'$
show $\text{trans } (R \cup R')$
by *blast*
— condition about IDC and NDC and equal length already proven above by auto
tactic!
next
fix $V V' i m1 m1' m2 \alpha p$
assume $\text{in}R'$: $(V, V') \in R'$
assume irange : $i < \text{length } V$
assume step : $\langle V!i, m1 \rangle \rightarrow \langle \alpha \rangle \langle p, m2 \rangle$
assume dhequal : $m1 \sim_{d, (\text{htchLocSet } PP)} m1'$
assume disjBisimUpTo : $\forall (V, V') \in R'. \forall i < \text{length } V. \forall m1 m1' m2 \alpha p.$
 $\langle V!i, m1 \rangle \rightarrow \langle \alpha \rangle \langle p, m2 \rangle \wedge m1 \sim_{d, (\text{htchLocSet } PP)} m1' \longrightarrow$
 $(\exists p' \alpha' m2'. \langle V!i, m1' \rangle \rightarrow \langle \alpha' \rangle \langle p', m2' \rangle \wedge$
 $\text{stepResultsin}R \text{ p } p' R' \wedge (\alpha, \alpha') \in R' \wedge$
 $\text{dhequality-alternative } d \text{ PP } (\text{pp } (V!i)) \text{ m2 } m2')$
from $\text{in}R' \text{ irange } \text{step } \text{dhequal } \text{disjBisimUpTo}$ **show** $\exists p' \alpha' m2'.$
 $\langle V!i, m1' \rangle \rightarrow \langle \alpha' \rangle \langle p', m2' \rangle \wedge \text{stepResultsin}R \text{ p } p' (R \cup R') \wedge$
 $((\alpha, \alpha') \in R \vee (\alpha, \alpha') \in R') \wedge$
 $\text{dhequality-alternative } d \text{ PP } (\text{pp } (V!i)) \text{ m2 } m2'$
by (*simp add: stepResultsinR-def, fastforce*)
qed

lemma *Union-Strong-dlHPP-Bisim*:
 $\llbracket \text{SdlHPPB } d \text{ PP } R; \text{SdlHPPB } d \text{ PP } R';$
 $\text{Areft } R \cap \text{Areft } R' \subseteq \{\emptyset\} \rrbracket$
 $\implies \text{SdlHPPB } d \text{ PP } (R \cup R')$
by (*rule Up-To-Technique, simp-all add:*
disj-dlHPP-Bisimulation-Up-To-R'-def
Strong-dlHPP-Bisimulation-def stepResultsinR-def, fastforce)

lemma *adding-emptypair-keeps-SdlHPPB*:
assumes SdlHPP : $\text{SdlHPPB } d \text{ PP } R$
shows $\text{SdlHPPB } d \text{ PP } (R \cup \{(\emptyset, \emptyset)\})$
proof —
have SdlHPPemp : $\text{SdlHPPB } d \text{ PP } \{(\emptyset, \emptyset)\}$
by (*simp add: Strong-dlHPP-Bisimulation-def sym-def trans-def*)

have commonDom : $\text{Domain } R \cap \text{Domain } \{(\emptyset, \emptyset)\} \subseteq \{\emptyset\}$
by *auto*

with $\text{commonAreft-subset-commonDomain}$ **have** Areflassump :
 $\text{Areft } R \cap \text{Areft } \{(\emptyset, \emptyset)\} \subseteq \{\emptyset\}$
by *force*

```

with SdlHPP SdlHPPemp Union-Strong-dlHPP-Bisim show
  SdlHPPB d PP (R  $\cup$   $\{(\[], \[])\}$ )
  by force
qed

end

end

```

2.3 Proof of parallel compositionality

We prove that WHAT&WHERE-security is preserved under composition of WHAT&WHERE-secure threads.

```

theory Parallel-Composition
imports Up-To-Technique MWLs
begin

locale WHATWHERE-Secure-Programs =
  L? : MWLs-semantics E BMap
+ WWS? : WHATWHERE MWLsSteps-det E pp DA lH
for E :: ('exp, 'id, 'val) Evalfunction
and BMap :: 'val  $\Rightarrow$  bool
and DA :: ('id, 'd::order) DomainAssignment
and lH :: ('d, 'exp) lHatches
begin

lemma SdlHPPB-restricted-on-PP-is-SdlHPPB:
  assumes SdlHPPB: SdlHPPB d PP R'
  assumes inR': (V, V)  $\in$  R'
  assumes Rdef: R =  $\{(V', V''). (V', V'') \in R'$ 
     $\wedge$  set (PPV V')  $\subseteq$  set (PPV V)
     $\wedge$  set (PPV V'')  $\subseteq$  set (PPV V) $\}$ 
  shows SdlHPPB d PP R
proof (simp add: Strong-dlHPP-Bisimulation-def, auto)
  from SdlHPPB have sym R'
    by (simp add: Strong-dlHPP-Bisimulation-def)
  with Rdef show sym R
    by (simp add: sym-def)
next
  from SdlHPPB have trans R'
    by (simp add: Strong-dlHPP-Bisimulation-def)
  with Rdef show trans R
    by (simp add: trans-def, auto)
next
  fix V' V''
  assume inR-part: (V', V'')  $\in$  R
  with SdlHPPB Rdef show length V' = length V''

```

```

    by (simp add: Strong-dlHPP-Bisimulation-def, auto)
next
fix V' V'' i
assume inR-part: (V', V'') ∈ R
assume irange: i < length V'
assume notIDC:
  ¬ IDC d (V'!i) (htchLoc (pp (V'!i)))
with SdlHPPB inR-part irange Rdef
show NDC d (V'!i)
  by (simp add: Strong-dlHPP-Bisimulation-def, auto)
next
fix V' V'' i α p m1 m1' m2
assume inR-part: (V', V'') ∈ R
assume irange: i < length V'
assume step: ⟨V'!i, m1⟩ →⟨α⟩ ⟨p, m2⟩
assume dhequal: m1 ∼d, (htchLocSet PP) m1'

from inR-part SdlHPPB Rdef have eqlen: length V' = length V''
  by (simp add: Strong-dlHPP-Bisimulation-def, auto)

from inR-part Rdef
have set (PPV V') ⊆ set (PPV V) ∧ set (PPV V'') ⊆ set (PPV V)
  by auto

with irange PPc-in-PPV-version eqlen
have PPc-Vs-at-i:
  set (PPc (V'!i)) ⊆ set (PPV V) ∧ set (PPc (V''!i)) ⊆ set (PPV V)
  by (metis subset-trans)

from SdlHPPB inR-part Rdef irange step dhequal
strongdlHPPB-aux[of d PP R' i
  V' V'' m1 α p m2 m1']
obtain p' α' m2' where stepreq: ⟨V''!i, m1'⟩ →⟨α'⟩ ⟨p', m2'⟩ ∧
  stepResultsinR p p' R' ∧ (α, α') ∈ R' ∧
  dhequality-alternative d PP (pp (V'!i)) m2 m2'
  by auto
have Rpp': stepResultsinR p p' R
proof -
{
  fix c c'
  assume step1: ⟨V'!i, m1⟩ →⟨α⟩ ⟨Some c, m2⟩
  assume step2: ⟨V''!i, m1'⟩ →⟨α'⟩ ⟨Some c', m2'⟩
  assume inR'-res: ([c], [c']) ∈ R'

  from PPc-Vs-at-i step1 step2 PPsc-of-step
  have set (PPc c) ⊆ set (PPV V) ∧ set (PPc c') ⊆ set (PPV V)
    by (metis (no-types) option.sel xt1(6))

  with inR'-res Rdef have ([c], [c']) ∈ R

```

```

    by auto
  }
  thus ?thesis
    by (metis step stepResultsinR-def stepreq)
qed

have Rαα': (α,α') ∈ R
proof -
  from PPc-Vs-at-i step stepreq PPα-of-step have
    set (PPV α) ⊆ set (PPV V) ∧ set (PPV α') ⊆ set (PPV V)
  by (metis (no-types) xt1(6))
  with stepreq Rdef show ?thesis
    by auto
qed

from stepreq Rpp' Rαα' show
  ∃ p' α' m2'. ⟨V'!i, m1⟩ →⟨α'⟩ ⟨p', m2⟩ ∧
  stepResultsinR p p' R ∧ (α,α') ∈ R ∧
  dhequality-alternative d PP (pp (V'!i)) m2 m2'
  by auto
qed

theorem parallel-composition:
  [ [ ∀ i < length V. WHATWHERE-Secure [V!i]; unique-PPV V ] ]
  ⇒ WHATWHERE-Secure V
proof (simp add: WHATWHERE-Secure-def, induct V, auto)
  fix d PP
  from WHATWHERE-empty
  show ∃ R. SdlHPPB d PP R ∧ ([],[]) ∈ R
    by (simp add: WHATWHERE-Secure-def)
next
  fix c V d PP
  assume IH: [ [ ∀ i < length V.
    ∃ d PP. ∃ R. SdlHPPB d PP R ∧ ([V!i],[V!i]) ∈ R;
    unique-PPV V ] ]
    ⇒ ∃ d PP. ∃ R. SdlHPPB d PP R ∧ (V,V) ∈ R
  assume ISassump: ∀ i < Suc (length V).
    ∃ d PP. ∃ R. SdlHPPB d PP R ∧ ([c#V]!i,[c#V]!i) ∈ R
  assume uniPPcV: unique-PPV (c#V)

  hence IHassump1: unique-PPV V
    by (simp add: unique-PPV-def)

  from uniPPcV have nocommonPP: set (PPc c) ∩ set (PPV V) = {}
    by (simp add: unique-PPV-def)

  from ISassump have IHassump2: ∀ i < length V.
    ∃ d PP. ∃ R. SdlHPPB d PP R ∧ ([V!i],[V!i]) ∈ R

```

by *auto*

with *IHassump1 IH* **obtain** RV' **where** RV' *assump*:
 SdlHPPB d PP $RV' \wedge (V, V) \in RV'$
 by *blast*

define RV **where** $RV = \{(V', V''). (V', V'') \in RV' \wedge \text{set } (PPV V') \subseteq \text{set } (PPV V)\}$
 $\wedge \text{set } (PPV V'') \subseteq \text{set } (PPV V)\}$

with RV' *assump* RV -*def SdlHPPB-restricted-on-PP-is-SdlHPPB*
have *SdlHPPRV: SdlHPPB d PP RV*
 by *force*

from *ISassump* **obtain** Rc' **where** Rc' *assump*:
 SdlHPPB d PP $Rc' \wedge ([c], [c]) \in Rc'$
 by (*metis append-Nil drop-Nil neq0-conv not-Cons-self*
 nth-append-length Cons-nth-drop-Suc zero-less-Suc)

define Rc **where** $Rc = \{(V', V''). (V', V'') \in Rc' \wedge \text{set } (PPV V') \subseteq \text{set } (PPc c)\}$
 $\wedge \text{set } (PPV V'') \subseteq \text{set } (PPc c)\}$

with Rc' *assump* Rc -*def SdlHPPB-restricted-on-PP-is-SdlHPPB*
have *SdlHPPRc: SdlHPPB d PP Rc*
 by *force*

from *nocommonPP* **have** $\text{Domain } RV \cap \text{Domain } Rc \subseteq \{\}\}$
 by (*simp add: RV-def Rc-def, auto,*
 metis Int-mono inf-commute inf-idem le-bot nocommonPP unique-V-uneq)

with *commonArefl-subset-commonDomain*
have *Areflassump1: Arefl* $RV \cap \text{Arefl } Rc \subseteq \{\}\}$
 by *force*

define R **where** $R = \{(V', V''). \exists c c' W W'. V' = c\#W \wedge V'' = c'\#W' \wedge W \neq \{\}\} \wedge W' \neq \{\} \wedge ([c], [c']) \in Rc \wedge (W, W') \in RV\}$

with RV -*def* RV' *assump* Rc -*def* Rc' *assump* **have** *inR*:
 $V \neq \{\} \implies (c\#V, c\#V) \in R$
 by *auto*

from R -*def* Rc -*def* RV -*def* *nocommonPP*
have $\text{Domain } R \cap \text{Domain } (Rc \cup RV) = \{\}$
 by (*simp add: R-def Rc-def RV-def, auto,*
 metis inf-bot-right le-inf-iff subset-empty unique-V-uneq,
 metis (opaque-lifting, no-types) inf-absorb1 inf-bot-right le-inf-iff unique-c-uneq)

```

with commonArefl-subset-commonDomain
have Areflassump2:  $Arefl\ R \cap Arefl\ (Rc \cup RV) \subseteq \{\emptyset\}$ 
  by force

have disjuptoR:
  disj-dlHPP-Bisimulation-Up-To-R' d PP (Rc \cup RV) R
proof (simp add: disj-dlHPP-Bisimulation-Up-To-R'-def, auto)
  from Areflassump1 SdlHPPRc SdlHPPRV Union-Strong-dlHPP-Bisim
  show SdlHPPB d PP (Rc \cup RV)
    by force
next
from SdlHPPRV have symRV: sym RV
  by (simp add: Strong-dlHPP-Bisimulation-def)
from SdlHPPRc have symRc: sym Rc
  by (simp add: Strong-dlHPP-Bisimulation-def)
with symRV R-def show sym R
  by (simp add: sym-def, auto)
next
from SdlHPPRV have transRV: trans RV
  by (simp add: Strong-dlHPP-Bisimulation-def)
from SdlHPPRc have transRc: trans Rc
  by (simp add: Strong-dlHPP-Bisimulation-def)
show trans R
  proof -
  {
  fix V V' V''
  assume p1:  $(V, V') \in R$ 
  assume p2:  $(V', V'') \in R$ 
  have  $(V, V'') \in R$ 
  proof -
  from p1 R-def obtain c c' W W' where p1assump:
     $V = c\#\ W \wedge V' = c'\#\ W' \wedge W \neq \square \wedge W' \neq \square \wedge$ 
     $([c],[c']) \in Rc \wedge (W, W') \in RV$ 
  by auto
  with p2 R-def obtain c'' W'' where p2assump:
     $V'' = c''\#\ W'' \wedge W'' \neq \square \wedge$ 
     $([c],[c'']) \in Rc \wedge (W', W'') \in RV$ 
  by auto
  with p1assump transRc transRV have
    trans-assump:  $([c],[c'']) \in Rc \wedge (W, W'') \in RV$ 
  by (simp add: trans-def, blast)
  with p1assump p2assump R-def show ?thesis
  by auto
  qed
  }
  thus ?thesis unfolding trans-def by blast
  qed
next
fix V V'

```

```

assume  $(V, V') \in R$ 
with  $R\text{-def SdlHPPRV}$  show  $\text{length } V = \text{length } V'$ 
  by (simp add: Strong-dlHPP-Bisimulation-def, auto)
next
fix  $V V' i$ 
assume  $\text{inR}: (V, V') \in R$ 
assume  $\text{irange}: i < \text{length } V$ 
assume  $\text{notIDC}: \neg \text{IDC } d (V!i)$ 
  (htchLoc (pp (V!i)))
from  $\text{inR } R\text{-def}$  obtain  $c c' W W'$  where  $VV'\text{assump}:$ 
   $V = c\#W \wedge V'=c'\#W' \wedge W \neq [] \wedge W' \neq [] \wedge$ 
   $([c],[c']) \in Rc \wedge (W, W') \in RV$ 
  by auto
  — Case separation for i
from  $VV'\text{assump SdlHPPRc}$  have  $\text{Case-}i0:$ 
   $i = 0 \implies (\text{NDC } d (V!i) \vee$ 
   $\text{IDC } d (V!i) (\text{htchLoc (pp (V!i))}))$ 
  by (simp add: Strong-dlHPP-Bisimulation-def, auto)

from  $VV'\text{assump SdlHPPRV}$  have  $\forall i < \text{length } W.$ 
   $(\text{NDC } d (W!i) \vee$ 
   $\text{IDC } d (W!i) (\text{htchLoc (pp (W!i))}))$ 
  by (simp add: Strong-dlHPP-Bisimulation-def, auto)

with  $\text{irange } VV'\text{assump}$  have  $\text{Case-}in0:$ 
   $i > 0 \implies (\text{NDC } d (V!i) \vee$ 
   $\text{IDC } d (V!i) (\text{htchLoc (pp (V!i))}))$ 
  by simp
from  $\text{notIDC Case-}i0 \text{ Case-}in0$ 
show  $\text{NDC } d (V!i)$ 
  by auto
next
fix  $V V' m1 m1' m2 \alpha p i$ 
assume  $\text{inR}: (V, V') \in R$ 
assume  $\text{irange}: i < \text{length } V$ 
assume  $\text{step}: \langle V!i, m1 \rangle \rightarrow \langle \alpha \rangle \langle p, m2 \rangle$ 
assume  $\text{dhequal}: m1 \sim_d, (\text{htchLocSet } PP) m1'$ 

from  $\text{inR } R\text{-def}$  obtain  $c c' W W'$  where  $VV'\text{assump}:$ 
   $V = c\#W \wedge V'=c'\#W' \wedge W \neq [] \wedge W' \neq [] \wedge$ 
   $([c],[c']) \in Rc \wedge (W, W') \in RV$ 
  by auto
  — Case separation for i
from  $VV'\text{assump SdlHPPRc strongdlHPPB-aux}$  [of d PP]
   $Rc 0 [c] [c'] \text{ step dhequal}$ 
have  $\text{Case-}i0:$ 
   $i = 0 \implies \exists p' \alpha' m2'.$ 
   $\langle V!i, m1' \rangle \rightarrow \langle \alpha' \rangle \langle p', m2' \rangle \wedge$ 
   $\text{stepResultsinR } p p' (R \cup (Rc \cup RV)) \wedge$ 

```


$((\alpha, \alpha') \in R \vee (\alpha, \alpha') \in Rc \vee (\alpha, \alpha') \in RV) \wedge$
dhequality-alternative d PP (pp (V!i)) m2 m2'
by (*simp add: stepResultsinR-def, blast*)

from *step VV'assump irange have rewV:*
 $i > 0 \implies (i - \text{Suc } 0) < \text{length } W \wedge V!i = W!(i - \text{Suc } 0)$
by *simp*

with *irange VV'assump step dhequal SdlHPPRV*
strongdlHPPB-aux[of d PP RV - W W']
have *Case-in0:*
 $i > 0 \implies \exists p' \alpha' m2'.$
 $\langle V!i, m1 \rangle \rightarrow \langle \alpha' \rangle \langle p', m2 \rangle \wedge$
 $\text{stepResultsinR } p \ p' (R \cup (Rc \cup RV)) \wedge$
 $((\alpha, \alpha') \in R \vee (\alpha, \alpha') \in Rc \vee (\alpha, \alpha') \in RV) \wedge$
dhequality-alternative d PP (pp (V!i)) m2 m2'
by (*simp add: stepResultsinR-def, blast*)

from *Case-i0 Case-in0*
show $\exists p' \alpha' m2'.$
 $\langle V!i, m1 \rangle \rightarrow \langle \alpha' \rangle \langle p', m2 \rangle \wedge$
 $\text{stepResultsinR } p \ p' (R \cup (Rc \cup RV)) \wedge$
 $((\alpha, \alpha') \in R \vee (\alpha, \alpha') \in Rc \vee (\alpha, \alpha') \in RV) \wedge$
dhequality-alternative d PP (pp (V!i)) m2 m2'
by *auto*

qed

with *Areflassump2 Rc'assump Up-To-Technique*
show $\exists R. \text{SdlHPPB } d \text{ PP } R \wedge (c \# V, c \# V) \in R$
by (*metis UnCI inR*)

qed

end

end

3 Example language and compositionality proofs

3.1 Example language with dynamic thread creation

As in [2], we instantiate the language with a simple while language that supports dynamic thread creation via a spawn command (Multi-threaded While Language with spawn, MWLs). Note that the language is still parametric in the language used for Boolean and arithmetic expressions (*'exp*).

```

theory MWLs
imports Strong-Security.Types
begin
  
```

— type parameters not instantiated:
 — `exp`: expressions (arithmetic, boolean...)
 — `val`: numbers, boolean constants....
 — `id`: identifier names

— SYNTAX

```
datatype ('exp, 'id) MWLsCom
  = Skip nat (skip_ [50] 70)
  | Assign 'id nat 'exp
    (:=_ - [70,50,70] 70)

  | Seq ('exp, 'id) MWLsCom
    ('exp, 'id) MWLsCom
    (-; - [61,60] 60)

  | If-Else nat 'exp ('exp, 'id) MWLsCom
    ('exp, 'id) MWLsCom
    (if_ - then - else - fi [50,80,79,79] 70)

  | While-Do nat 'exp ('exp, 'id) MWLsCom
    (while_ - do - od [50,80,79] 70)

  | Spawn nat (('exp, 'id) MWLsCom) list
    (spawn_ - [50,70] 70)
```

— function for obtaining the program point of some MWLsloc command

```
primrec pp :: ('exp, 'id) MWLsCom  $\Rightarrow$  nat
where
  pp (skip $\iota$ ) =  $\iota$  |
  pp (x := $\iota$  e) =  $\iota$  |
  pp (c1;c2) = pp c1 |
  pp (if $\iota$  b then c1 else c2 fi) =  $\iota$  |
  pp (while $\iota$  b do c od) =  $\iota$  |
  pp (spawn $\iota$  V) =  $\iota$ 
```

— mutually recursive functions to collect program points of commands and thread pools

```
primrec PPc :: ('exp,'id) MWLsCom  $\Rightarrow$  nat list
and PPV :: ('exp,'id) MWLsCom list  $\Rightarrow$  nat list
where
  PPc (skip $\iota$ ) = [ $\iota$ ] |
  PPc (x := $\iota$  e) = [ $\iota$ ] |
  PPc (c1;c2) = (PPc c1) @ (PPc c2) |
  PPc (if $\iota$  b then c1 else c2 fi) = [ $\iota$ ] @ (PPc c1) @ (PPc c2) |
  PPc (while $\iota$  b do c od) = [ $\iota$ ] @ (PPc c) |
  PPc (spawn $\iota$  V) = [ $\iota$ ] @ (PPV V) |
```

$PPV [] = []$
 $PPV (c\#V) = (PPc\ c) @ (PPV\ V)$

— predicate indicating that a command only contains unique program points

definition $unique\text{-}PPc :: ('exp, 'id)\ MWLsCom \Rightarrow bool$

where

$unique\text{-}PPc\ c = distinct\ (PPc\ c)$

— predicate indicating that a thread pool only contains unique program points

definition $unique\text{-}PPV :: ('exp, 'id)\ MWLsCom\ list \Rightarrow bool$

where

$unique\text{-}PPV\ V = distinct\ (PPV\ V)$

lemma $PPc\text{-}nonempty: PPc\ c \neq []$

by $(induct\ c)\ auto$

lemma $unique\text{-}c\text{-}uneq: set\ (PPc\ c) \cap set\ (PPc\ c') = \{\} \Longrightarrow c \neq c'$

by $(insert\ PPc\text{-}nonempty, force)$

lemma $V\text{-}nonempty\text{-}PPV\text{-}nonempty: V \neq [] \Longrightarrow PPV\ V \neq []$

by $(auto, induct\ V, simp\text{-}all, insert\ PPc\text{-}nonempty, force)$

lemma $unique\text{-}V\text{-}uneq:$

$\llbracket V \neq []; V' \neq []; set\ (PPV\ V) \cap set\ (PPV\ V') = \{\} \rrbracket \Longrightarrow V \neq V'$

by $(auto, induct\ V, simp\text{-}all, insert\ V\text{-}nonempty\text{-}PPV\text{-}nonempty, auto)$

lemma $PPc\text{-}in\text{-}PPV: c \in set\ V \Longrightarrow set\ (PPc\ c) \subseteq set\ (PPV\ V)$

by $(induct\ V, auto)$

lemma $listindices\text{-}aux: i < length\ V \Longrightarrow (V!i) \in set\ V$

by $(metis\ nth\text{-}mem)$

lemma $PPc\text{-}in\text{-}PPV\text{-}version:$

$i < length\ V \Longrightarrow set\ (PPc\ (V!i)) \subseteq set\ (PPV\ V)$

by $(rule\ PPc\text{-}in\text{-}PPV, erule\ listindices\text{-}aux)$

lemma $uniPPV\text{-}uniPPc: unique\text{-}PPV\ V \Longrightarrow (\forall i < length\ V. unique\text{-}PPc\ (V!i))$

by $(auto, simp\ add: unique\text{-}PPV\text{-}def, induct\ V,$

$auto\ simp\ add: unique\text{-}PPc\text{-}def,$

$metis\ in\text{-}set\ conv\text{-}nth\ length\text{-}Suc\ conv\ set\ ConsD)$

— SEMANTICS

locale $MWLs\text{-}semantics =$

fixes $E :: ('exp, 'id, 'val)\ Evalfunction$

and $BMap :: 'val \Rightarrow bool$

begin

— steps semantics, set of deterministic steps from commands to program states

inductive-set $MWLSSteps-det ::$ $(\text{'exp, 'id, 'val, ('exp, 'id) MWLsCom}) TLSteps$ **and** $MWLSlocSteps-det' ::$ $(\text{'exp, 'id, 'val, ('exp, 'id) MWLsCom}) TLSteps-curry$ $((1\langle -,/- \rangle) \rightarrow \triangleleft \triangleright / (1\langle -,/- \rangle) [0,0,0,0,0] 81)$ **where** $\langle c1, m1 \rangle \rightarrow \triangleleft \alpha \triangleright \langle c2, m2 \rangle \equiv ((c1, m1), \alpha, (c2, m2)) \in MWLSSteps-det \mid$ $skip: \langle skip_{\iota}, m \rangle \rightarrow \triangleleft \square \triangleright \langle None, m \rangle \mid$ $assign: (E e m) = v \implies$ $\langle x :=_{\iota} e, m \rangle \rightarrow \triangleleft \square \triangleright \langle None, m(x := v) \rangle \mid$ $seq1: \langle c1, m \rangle \rightarrow \triangleleft \alpha \triangleright \langle None, m^{\wedge} \rangle \implies$ $\langle c1; c2, m \rangle \rightarrow \triangleleft \alpha \triangleright \langle Some\ c2, m^{\wedge} \rangle \mid$ $seq2: \langle c1, m \rangle \rightarrow \triangleleft \alpha \triangleright \langle Some\ c1', m^{\wedge} \rangle \implies$ $\langle c1; c2, m \rangle \rightarrow \triangleleft \alpha \triangleright \langle Some\ (c1'; c2), m^{\wedge} \rangle \mid$ $iftrue: BMap (E b m) = True \implies$ $\langle if_{\iota} b \text{ then } c1 \text{ else } c2\ fi, m \rangle \rightarrow \triangleleft \square \triangleright \langle Some\ c1, m \rangle \mid$ $iffalse: BMap (E b m) = False \implies$ $\langle if_{\iota} b \text{ then } c1 \text{ else } c2\ fi, m \rangle \rightarrow \triangleleft \square \triangleright \langle Some\ c2, m \rangle \mid$ $whiletrue: BMap (E b m) = True \implies$ $\langle while_{\iota} b \text{ do } c\ od, m \rangle \rightarrow \triangleleft \square \triangleright \langle Some\ (c; (while_{\iota} b \text{ do } c\ od)), m \rangle \mid$ $whilefalse: BMap (E b m) = False \implies$ $\langle while_{\iota} b \text{ do } c\ od, m \rangle \rightarrow \triangleleft \square \triangleright \langle None, m \rangle \mid$ $spawn: \langle spawn_{\iota} V, m \rangle \rightarrow \triangleleft V \triangleright \langle None, m \rangle$ **inductive-cases** $MWLSSteps-det-cases:$ $\langle skip_{\iota}, m \rangle \rightarrow \triangleleft \alpha \triangleright \langle p, m^{\wedge} \rangle$ $\langle x :=_{\iota} e, m \rangle \rightarrow \triangleleft \alpha \triangleright \langle p, m^{\wedge} \rangle$ $\langle c1; c2, m \rangle \rightarrow \triangleleft \alpha \triangleright \langle p, m^{\wedge} \rangle$ $\langle if_{\iota} b \text{ then } c1 \text{ else } c2\ fi, m \rangle \rightarrow \triangleleft \alpha \triangleright \langle p, m^{\wedge} \rangle$ $\langle while_{\iota} b \text{ do } c\ od, m \rangle \rightarrow \triangleleft \alpha \triangleright \langle p, m^{\wedge} \rangle$ $\langle spawn_{\iota} V, m \rangle \rightarrow \triangleleft \alpha \triangleright \langle p, m^{\wedge} \rangle$

— non-deterministic, possibilistic system step (added for intuition, not used in the proofs)

inductive-set $MWLSSteps-ndet ::$ $(\text{'exp, 'id, 'val, ('exp, 'id) MWLsCom}) TPSteps$ **and** $MWLSSteps-ndet' ::$ $(\text{'exp, 'id, 'val, ('exp, 'id) MWLsCom}) TPSteps-curry$ $((1\langle -,/- \rangle) \Rightarrow / (1\langle -,/- \rangle) [0,0,0,0] 81)$ **where** $\langle V, m \rangle \Rightarrow \langle V', m^{\wedge} \rangle \equiv ((V, m), (V', m^{\wedge})) \in MWLSSteps-ndet \mid$ $stepthreadi1: \langle ci, m \rangle \rightarrow \triangleleft \alpha \triangleright \langle None, m^{\wedge} \rangle \implies$ $\langle cf @ [ci] @ ca, m \rangle \Rightarrow \langle cf @ \alpha @ ca, m^{\wedge} \rangle \mid$ $stepthreadi2: \langle ci, m \rangle \rightarrow \triangleleft \alpha \triangleright \langle Some\ c', m^{\wedge} \rangle \implies$ $\langle cf @ [ci] @ ca, m \rangle \Rightarrow \langle cf @ [c'] @ \alpha @ ca, m \rangle$

— lemma about existence and uniqueness of next memory of a step

lemma *nextmem-exists-and-unique*:

$\exists m' p \alpha. \langle c, m \rangle \rightarrow \triangleleft \alpha \triangleright \langle p, m' \rangle$
 $\wedge (\forall m''. (\exists p \alpha. \langle c, m \rangle \rightarrow \triangleleft \alpha \triangleright \langle p, m'' \rangle) \longrightarrow m'' = m')$
by (*induct* *c*, *auto*, *metis* *MWLSSteps-det.skip* *MWLSSteps-det-cases(1)*,
metis *MWLSSteps-det-cases(2)* *MWLSSteps-det.assign*,
metis (*no-types*) *MWLSSteps-det.seq1* *MWLSSteps-det.seq2*
MWLSSteps-det-cases(3) *not-Some-eq*,
metis *MWLSSteps-det.iffalse* *MWLSSteps-det.iftrue*
MWLSSteps-det-cases(4),
metis *MWLSSteps-det.whilefalse* *MWLSSteps-det.whiletrue*
MWLSSteps-det-cases(5),
metis *MWLSSteps-det.spawn* *MWLSSteps-det-cases(6)*)

lemma *PPsc-of-step*:

$\llbracket \langle c, m \rangle \rightarrow \triangleleft \alpha \triangleright \langle p, m' \rangle; \exists c'. p = \text{Some } c' \rrbracket$
 $\implies \text{set } (PPc \text{ (the } p)) \subseteq \text{set } (PPc \ c)$
by (*induct rule*: *MWLSSteps-det.induct*, *auto*)

lemma *PPV α -of-step*:

$\langle c, m \rangle \rightarrow \triangleleft \alpha \triangleright \langle p, m' \rangle$
 $\implies \text{set } (PPV \ \alpha) \subseteq \text{set } (PPc \ c)$
by (*induct rule*: *MWLSSteps-det.induct*, *auto*)

end

end

3.2 Proofs of atomic compositionality results

We prove for each atomic command of our example programming language (i.e. a command that is not composed out of other commands) that it is strongly secure if the expressions involved are indistinguishable for an observer on security level d .

theory *WHATWHERE-Secure-Skip-Assign*

imports *Parallel-Composition*

begin

context *WHATWHERE-Secure-Programs*

begin

abbreviation *NextMem'*

$\llbracket _ \rrbracket'(-)$

where

$\llbracket c \rrbracket(m) \equiv \text{NextMem } c \ m$

— define when two expressions are indistinguishable with respect to a domain d

definition $d\text{-indistinguishable} :: 'd::order \Rightarrow 'exp \Rightarrow 'exp \Rightarrow bool$

where

$d\text{-indistinguishable } d \ e1 \ e2 \equiv \forall m \ m'. ((m =_d m') \rightarrow ((E \ e1 \ m) = (E \ e2 \ m')))$

abbreviation $d\text{-indistinguishable}' :: 'exp \Rightarrow 'd::order \Rightarrow 'exp \Rightarrow bool$

$((- \equiv_d -))$

where

$e1 \equiv_d e2 \equiv d\text{-indistinguishable } d \ e1 \ e2$

— symmetry of d -indistinguishable

lemma $d\text{-indistinguishable-sym}$:

$e \equiv_d e' \Longrightarrow e' \equiv_d e$

by (*simp add: d-indistinguishable-def d-equal-def, metis*)

— transitivity of d -indistinguishable

lemma $d\text{-indistinguishable-trans}$:

$\llbracket e \equiv_d e'; e' \equiv_d e'' \rrbracket \Longrightarrow e \equiv_d e''$

by (*simp add: d-indistinguishable-def d-equal-def, metis*)

— predicate for dH -indistinguishable

definition $dH\text{-indistinguishable} :: 'd \Rightarrow ('d, 'exp) \text{ Hatches}$

$\Rightarrow 'exp \Rightarrow 'exp \Rightarrow bool$

where

$dH\text{-indistinguishable } d \ H \ e1 \ e2 \equiv (\forall m \ m'. m \sim_{d,H} m' \rightarrow (E \ e1 \ m = E \ e2 \ m'))$

abbreviation $dH\text{-indistinguishable}' :: 'exp \Rightarrow 'd$

$\Rightarrow ('d, 'exp) \text{ Hatches} \Rightarrow 'exp \Rightarrow bool$

$((- \equiv_{d,H} -))$

where

$e1 \equiv_{d,H} e2 \equiv dH\text{-indistinguishable } d \ H \ e1 \ e2$

lemma $empH\text{-implies-dHindistinguishable-eq-dindistinguishable}$:

$(e \equiv_{d,\{\}} e') = (e \equiv_d e')$

by (*simp add: d-indistinguishable-def dH-indistinguishable-def dH-equal-def d-equal-def*)

theorem $WHATWHERE\text{-Secure-Skip}$:

$WHATWHERE\text{-Secure } [skip_i]$

proof (*simp add: WHATWHERE-Secure-def, auto*)

fix $d \ PP$

define R **where** $R = \{(V::('exp,'id) \text{ MWLsCom } list, V'::('exp,'id) \text{ MWLsCom } list).$

$V = V' \wedge (V = [] \vee V = [skip_i])\}$

have $inR: ([skip_\iota], [skip_\iota]) \in R$
by (*simp add: R-def*)

have $SdlHPPB\ d\ PP\ R$

proof (*simp add: Strong-dlHPP-Bisimulation-def R-def sym-def trans-def NDC-def NextMem-def, auto*)

fix $m1\ m1'$

assume $dequal: m1 =_d m1'$

have $nextm1: (THE\ m2.\ (\exists p\ \alpha.\ \langle skip_\iota, m1 \rangle \rightarrow \triangleleft \alpha \triangleright \langle p, m2 \rangle)) = m1$

by (*insert MWLsSteps-det.simps[of skip_\iota m1], force*)

have $nextm1'$:

$(THE\ m2'.\ (\exists p\ \alpha.\ \langle skip_\iota, m1' \rangle \rightarrow \triangleleft \alpha \triangleright \langle p, m2' \rangle)) = m1'$

by (*insert MWLsSteps-det.simps[of skip_\iota m1'], force*)

with $dequal\ nextm1$ **show**

$THE\ m2.\ \exists p\ \alpha.\ \langle skip_\iota, m1 \rangle \rightarrow \triangleleft \alpha \triangleright \langle p, m2 \rangle =_d$

$THE\ m2'.\ \exists p\ \alpha.\ \langle skip_\iota, m1' \rangle \rightarrow \triangleleft \alpha \triangleright \langle p, m2' \rangle$

by *auto*

next

fix $p\ m1\ m1'\ m2\ \alpha$

assume $dequal: m1 \sim_d (htchLocSet\ PP)\ m1'$

assume $skipstep: \langle skip_\iota, m1 \rangle \rightarrow \triangleleft \alpha \triangleright \langle p, m2 \rangle$

with $MWLsSteps-det.simps[of\ skip_\iota\ m1\ \alpha\ p\ m2]$

have $aux: p = None \wedge m2 = m1 \wedge \alpha = []$

by *auto*

with $dequal\ skipstep\ MWLsSteps-det.skip$

show $\exists p'\ m2'$.

$\langle skip_\iota, m1' \rangle \rightarrow \triangleleft \alpha \triangleright \langle p', m2' \rangle \wedge$

$stepResultsinR\ p\ p'\ \{(V, V').\ V = V' \wedge$

$(V = [] \vee V = [skip_\iota])\} \wedge$

$(\alpha = [] \vee \alpha = [skip_\iota]) \wedge$

$dhequality-alternative\ d\ PP\ \iota\ m2\ m2'$

by (*simp add: stepResultsinR-def dhequality-alternative-def, fastforce*)

qed

with inR **show** $\exists R.\ SdlHPPB\ d\ PP\ R \wedge ([skip_\iota], [skip_\iota]) \in R$

by *auto*

qed

— auxiliary lemma for assign side condition (lemma 9 in original paper)

lemma *semAssignSC-aux*:

assumes $dhind: e \equiv_{DA} x, (htchLoc\ \iota)\ e$

shows $NDC\ d\ (x :=_\iota e) \vee IDC\ d\ (x :=_\iota e)\ (htchLoc\ (pp\ (x :=_\iota e)))$

proof (*simp add: IDC-def, auto, simp add: NDC-def*)

fix $m1\ m1'$
assume $dhequal: m1 \sim_{d, htchLoc\ \iota} m1'$
hence $dequal: m1 =_d m1'$
by (*simp add: dH-equal-def*)

obtain v **where** $veq: E\ e\ m1 = v$ **by** *auto*
hence $m2eq: \llbracket x :=_{\iota} e \rrbracket(m1) = m1(x := v)$
by (*simp add: NextMem-def*,
insert MWLsSteps-det.simps[of x :=_ι e m1],
force)

obtain v' **where** $v'eq: E\ e\ m1' = v'$ **by** *auto*
hence $m2'eq: \llbracket x :=_{\iota} e \rrbracket(m1') = m1'(x := v')$
by (*simp add: NextMem-def*,
insert MWLsSteps-det.simps[of x :=_ι e m1'],
force)

from $dhequal$ **have** *shiftdomain*:
 $DA\ x \leq d \implies m1 \sim_{DA\ x, (htchLoc\ \iota)} m1'$
by (*simp add: dH-equal-def d-equal-def htchLoc-def*)

with $veq\ v'eq$ *dhind*
have $(DA\ x) \leq d \implies v = v'$
by (*simp add: dH-indistinguishable-def*)

with $dequal\ m2eq\ m2'eq$
show $\llbracket x :=_{\iota} e \rrbracket(m1) =_d \llbracket x :=_{\iota} e \rrbracket(m1')$
by (*simp add: d-equal-def*)

qed

theorem *WHATWHERE-Secure-Assign*:
assumes $dhind: e \equiv_{DA\ x, (htchLoc\ \iota)} e$
assumes $dheq-imp: \forall m\ m'\ d\ \iota'. (m \sim_{d, (htchLoc\ \iota')} m' \wedge$
 $\llbracket x :=_{\iota} e \rrbracket(m) =_d \llbracket x :=_{\iota} e \rrbracket(m'))$
 $\longrightarrow \llbracket x :=_{\iota} e \rrbracket(m) \sim_{d, (htchLoc\ \iota')} \llbracket x :=_{\iota} e \rrbracket(m')$
shows *WHATWHERE-Secure* $\llbracket x :=_{\iota} e \rrbracket$
proof (*simp add: WHATWHERE-Secure-def, auto*)
fix $d\ PP$
define R **where** $R = \{(V::('exp, 'id)\ MWLsCom\ list, V'::('exp, 'id)\ MWLsCom\ list).$
 $V = V' \wedge (V = [] \vee V = [x :=_{\iota} e])\}$

have $inR: (\llbracket x :=_{\iota} e \rrbracket, \llbracket x :=_{\iota} e \rrbracket) \in R$
by (*simp add: R-def*)

have *SdlHPPB* $d\ PP\ R$
proof (*simp add: Strong-dlHPP-Bisimulation-def R-def*)


```

    sym-def trans-def, auto)
  assume notIDC:  $\neg IDC\ d\ (x :=_{\iota}\ e)\ (htchLoc\ \iota)$ 
  with dhind semAssignSC-aux
  show NDC  $d\ (x :=_{\iota}\ e)$ 
    by force
next
fix m1 m1' m2 p  $\alpha$ 
assume assignstep:  $\langle x :=_{\iota}\ e, m1 \rangle \rightarrow \langle \alpha \rangle \langle p, m2 \rangle$ 
assume dhequal:  $m1 \sim_{d, htchLocSet\ PP}\ m1'$ 

from assignstep have nextm1:
   $p = None \wedge \alpha = [] \wedge \llbracket x :=_{\iota}\ e \rrbracket(m1) = m2$ 
  by (simp add: NextMem-def,
      insert MWLsSteps-det.simps[of  $x :=_{\iota}\ e\ m1$ ], force)

obtain m2' where nextm1':
   $\langle x :=_{\iota}\ e, m1' \rangle \rightarrow \langle [] \rangle \langle None, m2' \rangle \wedge \llbracket x :=_{\iota}\ e \rrbracket(m1') = m2'$ 
  by (simp add: NextMem-def,
      insert MWLsSteps-det.simps[of  $x :=_{\iota}\ e\ m1'$ ], force)

from dhequal have dhequal- $\iota$ :  $\bigwedge \iota. htchLoc\ \iota \subseteq htchLocSet\ PP$ 
 $\implies m1 \sim_{d, (htchLoc\ \iota)} m1'$ 
  by (simp add: dH-equal-def, auto)

with dhind semAssignSC-aux
have htchLoc  $\iota \subseteq htchLocSet\ PP \implies$ 
 $\llbracket x :=_{\iota}\ e \rrbracket(m1) =_d \llbracket x :=_{\iota}\ e \rrbracket(m1')$ 
  by (simp add: NDC-def IDC-def dH-equal-def, fastforce)

with dhind dheq-imp dhequal- $\iota$ 
have htchLoc  $\iota \subseteq htchLocSet\ PP \implies$ 
 $\llbracket x :=_{\iota}\ e \rrbracket(m1) \sim_{d, (htchLocSet\ PP)} \llbracket x :=_{\iota}\ e \rrbracket(m1')$ 
  by (simp add: htchLocSet-def dH-equal-def, blast)

with nextm1 nextm1' assignstep dhequal
show  $\exists p'\ m2'$ .
 $\langle x :=_{\iota}\ e, m1' \rangle \rightarrow \langle \alpha \rangle \langle p', m2' \rangle \wedge$ 
 $stepResultsinR\ p\ p'\ \{(V, V').\ V = V' \wedge (V = [] \vee V = [x :=_{\iota}\ e])\} \wedge$ 
 $(\alpha = [] \vee \alpha = [x :=_{\iota}\ e]) \wedge dhequality-alternative\ d\ PP\ \iota\ m2\ m2'$ 
  by (auto simp add: stepResultsinR-def dhequality-alternative-def)
qed

with inR show  $\exists R. SdlHPPB\ d\ PP\ R \wedge ([x :=_{\iota}\ e], [x :=_{\iota}\ e]) \in R$ 
  by auto
qed

end

end

```

3.3 Proofs of non-atomic compositionality results

We prove compositionality results for each non-atomic command of our example programming language (i.e. a command that is composed out of other commands): If the components are strongly secure and the expressions involved indistinguishable for an observer on security level d , then the composed command is also strongly secure.

```

theory Language-Composition
imports WHATWHERE-Secure-Skip-Assign
begin

context WHATWHERE-Secure-Programs
begin

theorem Compositionality-Seq:
  assumes WWs-part1: WHATWHERE-Secure [c1]
  assumes WWs-part2: WHATWHERE-Secure [c2]
  assumes uniPPc1c2: unique-PPc (c1;c2)
  shows WHATWHERE-Secure [c1;c2]
proof (simp add: WHATWHERE-Secure-def, auto)
  fix d PP

  from uniPPc1c2 have nocommonPP:  $set (PPc\ c1) \cap set (PPc\ c2) = \{\}$ 
    by (simp add: unique-PPV-def unique-PPc-def)

  from WWs-part1 obtain R1' where R1'assump:
    SdlHPPB d PP R1' \wedge ([c1],[c1]) \in R1'
    by (simp add: WHATWHERE-Secure-def, auto)

  define R1 where  $R1 = \{(V, V'). (V, V') \in R1' \wedge set (PPV\ V) \subseteq set (PPc\ c1) \wedge set (PPV\ V') \subseteq set (PPc\ c1)\}$ 

  from R1'assump R1-def SdlHPPB-restricted-on-PP-is-SdlHPPB
  have SdlHPPR1: SdlHPPB d PP R1
    by force

  from WWs-part2 obtain R2' where R2'assump:
    SdlHPPB d PP R2' \wedge ([c2],[c2]) \in R2'
    by (simp add: WHATWHERE-Secure-def, auto)

  define R2 where  $R2 = \{(V, V'). (V, V') \in R2' \wedge set (PPV\ V) \subseteq set (PPc\ c2) \wedge set (PPV\ V') \subseteq set (PPc\ c2)\}$ 

  from R2'assump R2-def SdlHPPB-restricted-on-PP-is-SdlHPPB
  have SdlHPPR2: SdlHPPB d PP R2
    by force

```

```

from nocommonPP have nocommonDomain:  $\text{Domain } R1 \cap \text{Domain } R2 \subseteq \{\emptyset\}$ 
  by (simp add: R1-def R2-def, auto,
    metis inf-greatest inf-idem le-bot unique-V-uneq)

with commonArefl-subset-commonDomain
have Areflassump1:  $\text{Arefl } R1 \cap \text{Arefl } R2 \subseteq \{\emptyset\}$ 
  by force

define R0 where  $R0 = \{(s1,s2). \exists c1\ c1'\ c2\ c2'. s1 = [c1;c2] \wedge s2 = [c1';c2']$ 
^
   $([c1],[c1']) \in R1 \wedge ([c2],[c2']) \in R2\}$ 

with R1-def R1'-assump R2-def R2'-assump
have inR0:  $([c1;c2],[c1';c2']) \in R0$ 
  by auto

have  $\text{Domain } R0 \cap \text{Domain } (R1 \cup R2) = \{\}$ 
  by (simp add: R0-def R1-def R2-def, auto, metis Int-absorb1 Int-assoc Int-empty-left

    nocommonPP unique-c-uneq, metis Int-absorb Int-absorb1
    Int-assoc Int-empty-left nocommonPP unique-c-uneq)

with commonArefl-subset-commonDomain
have Areflassump2:  $\text{Arefl } R0 \cap \text{Arefl } (R1 \cup R2) \subseteq \{\emptyset\}$ 
  by force

have disjuptoR0:
  disj-dlHPP-Bisimulation-Up-To-R' d PP (R1 ∪ R2) R0
  proof (simp add: disj-dlHPP-Bisimulation-Up-To-R'-def, auto)
    from Areflassump1 SdlHPPR1 SdlHPPR2 Union-Strong-dlHPP-Bisim
    show SdlHPPB d PP (R1 ∪ R2)
      by metis
    next
      from SdlHPPR1 have symR1: sym R1
        by (simp add: Strong-dlHPP-Bisimulation-def)
      from SdlHPPR2 have symR2: sym R2
        by (simp add: Strong-dlHPP-Bisimulation-def)
      with symR1 R0-def show sym R0
        by (simp add: sym-def, auto)
      next
        from SdlHPPR1 have transR1: trans R1
          by (simp add: Strong-dlHPP-Bisimulation-def)
        from SdlHPPR2 have transR2: trans R2
          by (simp add: Strong-dlHPP-Bisimulation-def)
        show trans R0
          proof –
            {
              fix V V' V''

```

```

assume  $p1: (V, V') \in R0$ 
assume  $p2: (V', V'') \in R0$ 
have  $(V, V'') \in R0$ 
proof –
  from  $p1$  R0-def obtain  $c1\ c2\ c1'\ c2'$  where  $p1assump$ :
     $V = [c1;c2] \wedge V' = [c1';c2'] \wedge$ 
     $([c1],[c1']) \in R1 \wedge ([c2],[c2']) \in R2$ 
    by auto
  with  $p2$  R0-def obtain  $c1''\ c2''$  where  $p2assump$ :
     $V'' = [c1'';c2''] \wedge$ 
     $([c1'],[c1'']) \in R1 \wedge ([c2'],[c2'']) \in R2$ 
    by auto
  with  $p1assump$  transR1 transR2 have
     $trans-assump: ([c1],[c1'']) \in R1 \wedge ([c2],[c2'']) \in R2$ 
    by (simp add: trans-def, blast)
  with  $p1assump\ p2assump\ R0-def$  show ?thesis
    by auto
  qed
}
thus ?thesis unfolding trans-def by blast
qed
next
fix  $V\ V'$ 
assume  $(V, V') \in R0$ 
with R0-def show  $length\ V = length\ V'$ 
  by auto
next
fix  $V\ V'\ i$ 
assume  $inR0: (V, V') \in R0$ 
assume  $irange: i < length\ V$ 
assume  $notIDC: \neg IDC\ d\ (V!i)\ (htchLoc\ (pp\ (V!i)))$ 
from  $inR0$  R0-def obtain  $c1\ c2\ c1'\ c2'$  where  $VV'assump$ :
   $V = [c1;c2] \wedge V' = [c1';c2'] \wedge$ 
   $([c1],[c1']) \in R1 \wedge ([c2],[c2']) \in R2$ 
  by auto
have  $eqnextmem: \bigwedge m. \llbracket c1;c2 \rrbracket(m) = \llbracket c1 \rrbracket(m)$ 
proof –
  fix  $m$ 
from nextmem-exists-and-unique obtain  $m'$  where  $c1nextmem$ :
     $\exists p\ \alpha. \langle c1, m \rangle \rightarrow \triangleleft \alpha \triangleright \langle p, m' \rangle$ 
     $\wedge (\forall m''. (\exists p\ \alpha. \langle c1, m \rangle \rightarrow \triangleleft \alpha \triangleright \langle p, m'' \rangle) \longrightarrow m'' = m')$ 
    by force

  hence  $eqdir1: \llbracket c1 \rrbracket(m) = m'$ 
    by (simp add: NextMem-def, auto)

from  $c1nextmem$  obtain  $p\ \alpha$  where  $\langle c1, m \rangle \rightarrow \triangleleft \alpha \triangleright \langle p, m' \rangle$ 
  by auto

```

with $c1nextmem$ **have** $\exists p \alpha. \langle c1;c2,m \rangle \rightarrow \langle \alpha \rangle \langle p,m \rangle$
 $\wedge (\forall m''. (\exists p \alpha. \langle c1;c2,m \rangle \rightarrow \langle \alpha \rangle \langle p,m'' \rangle) \longrightarrow m'' = m)$
by (*auto*, *metis MWLsSteps-det.seq1 MWLsSteps-det.seq2*
option.exhaust, *metis MWLsSteps-det-cases(3)*)

hence $eqdir2: \llbracket c1;c2 \rrbracket(m) = m'$
by (*simp add: NextMem-def*, *auto*)

with $eqdir1$ **show** $\llbracket c1;c2 \rrbracket(m) = \llbracket c1 \rrbracket(m)$

by *auto*

qed

have $eqpp: pp (c1;c2) = pp c1$

by *simp*

from $VV'assump SdlHPPR1$ **have** $IDC d c1 (htchLoc (pp c1))$
 $\vee NDC d c1$

by (*simp add: Strong-dlHPP-Bisimulation-def*, *auto*)

with $eqnextmem eqpp$ **have** $IDC d (c1;c2)$

$(htchLoc (pp (c1;c2))) \vee NDC d (c1;c2)$

by (*simp add: IDC-def NDC-def*)

with $inR0 irange notIDC VV'assump$

show $NDC d (V!i)$

by (*simp add: IDC-def*, *auto*)

next

fix $V V' m1 m1' m2 \alpha p i$

assume $inR0: (V, V') \in R0$

assume $irange: i < length V$

assume $step: \langle V!i, m1 \rangle \rightarrow \langle \alpha \rangle \langle p, m2 \rangle$

assume $dhequal: m1 \sim_{d, htchLocSet PP} m1'$

from $inR0 R0-def$ **obtain** $c1 c1' c2 c2'$ **where** $R0pair:$

$V = [c1;c2] \wedge V' = [c1';c2'] \wedge ([c1],[c1']) \in R1$

$\wedge ([c2],[c2']) \in R2$

by *auto*

from $R0pair irange$ **have** $i0: i = 0$ **by** *simp*

have $eqpp: pp (c1;c2) = pp c1$

by *simp*

— get the two different cases:

from $R0pair step i0$ **obtain** $c3$ **where** *case-distinction:*

$(p = Some c2 \wedge \langle c1, m1 \rangle \rightarrow \langle \alpha \rangle \langle None, m2 \rangle)$

$\vee (p = Some (c3;c2) \wedge \langle c1, m1 \rangle \rightarrow \langle \alpha \rangle \langle Some c3, m2 \rangle)$

by (*simp*, *insert MWLsSteps-det.simps[of c1;c2 m1]*,

auto)

moreover

— Case 1: first command terminates

{

assume *passump*: $p = \text{Some } c2$
assume *StepAssump*: $\langle c1, m1 \rangle \rightarrow \triangleleft \alpha \triangleright \langle \text{None}, m2 \rangle$
hence *Vstep-case1*:
 $\langle c1; c2, m1 \rangle \rightarrow \triangleleft \alpha \triangleright \langle \text{Some } c2, m2 \rangle$
by (*simp add: MWLsSteps-det.seq1*)

from *SdlHPPR1 StepAssump R0pair dhequal*
strongdlHPPB-aux[of *d PP*
 $R1\ 0\ [c1]\ [c1^\wedge]\ m1\ \alpha\ \text{None}\ m2\ m1^\wedge]$
obtain $p'\ \alpha'\ m2'$ **where** *c1c1'reason*:
 $p' = \text{None} \wedge \langle c1', m1^\wedge \rangle \rightarrow \triangleleft \alpha' \triangleright \langle p', m2^\wedge \rangle \wedge (\alpha, \alpha') \in R1 \wedge$
dhequality-alternative d PP (pp c1) m2 m2'
by (*simp add: stepResultsinR-def, fastforce*)

with *eqpp c1c1'reason have conclpart*:
 $\langle c1'; c2', m1^\wedge \rangle \rightarrow \triangleleft \alpha' \triangleright \langle \text{Some } c2', m2^\wedge \rangle \wedge$
dhequality-alternative d PP (pp (c1;c2)) m2 m2'
by (*auto, simp add: MWLsSteps-det.seq1*)

with *passump R0pair c1c1'reason i0*
have *case1-concl*:
 $\exists p'\ \alpha'\ m2'$.
 $\langle V!i, m1^\wedge \rangle \rightarrow \triangleleft \alpha' \triangleright \langle p', m2^\wedge \rangle \wedge$
stepResultsinR p p' (R0 \cup (R1 \cup R2)) \wedge
 $((\alpha, \alpha') \in R0 \vee (\alpha, \alpha') \in R1 \vee (\alpha, \alpha') \in R2) \wedge$
dhequality-alternative d PP (pp (V!i)) m2 m2'
by (*simp add: stepResultsinR-def, auto*)

}
moreover
— Case 2: first command does not terminate
{

assume *passump*: $p = \text{Some } (c3; c2)$
assume *StepAssump*: $\langle c1, m1 \rangle \rightarrow \triangleleft \alpha \triangleright \langle \text{Some } c3, m2 \rangle$

hence *Vstep-case2*: $\langle c1; c2, m1 \rangle \rightarrow \triangleleft \alpha \triangleright \langle \text{Some } (c3; c2), m2 \rangle$
by (*simp add: MWLsSteps-det.seq2*)

from *SdlHPPR1 StepAssump R0pair dhequal*
strongdlHPPB-aux[of *d PP*
 $R1\ 0\ [c1]\ [c1^\wedge]\ m1\ \alpha\ \text{Some } c3\ m2\ m1^\wedge]$
obtain $p'\ c3'\ \alpha'\ m2'$ **where** *c1c1'reason*:
 $p' = \text{Some } c3' \wedge \langle c1', m1^\wedge \rangle \rightarrow \triangleleft \alpha' \triangleright \langle p', m2^\wedge \rangle \wedge$
 $([c3'], [c3^\wedge]) \in R1 \wedge (\alpha, \alpha') \in R1 \wedge$
dhequality-alternative d PP (pp c1) m2 m2'
by (*simp add: stepResultsinR-def, fastforce*)

with *eqpp c1c1'reason have conclpart*:
 $\langle c1'; c2', m1^\wedge \rangle \rightarrow \triangleleft \alpha' \triangleright \langle \text{Some } (c3'; c2'), m2^\wedge \rangle \wedge$
dhequality-alternative d PP (pp (c1;c2)) m2 m2'

by (auto, simp add: MWLsSteps-det.seq2)

from $c1c1'$ reason $R0$ pair $R0$ -def have
($[c3;c2],[c3';c2'] \in R0$)
by auto

with $R0$ pair conclpart passump $c1c1'$ reason $i0$
have case1-concl:
 $\exists p' \alpha' m2'. \langle V!i, m1 \rangle \rightarrow \langle \alpha' \rangle \langle p', m2 \rangle \wedge$
 $stepResultsinR p p' (R0 \cup (R1 \cup R2)) \wedge$
 $((\alpha, \alpha') \in R0 \vee (\alpha, \alpha') \in R1 \vee (\alpha, \alpha') \in R2) \wedge$
 $dhequality-alternative d PP (pp (V!i)) m2 m2'$
by (simp add: stepResultsinR-def, auto)

}

ultimately

show $\exists p' \alpha' m2'. \langle V!i, m1 \rangle \rightarrow \langle \alpha' \rangle \langle p', m2 \rangle \wedge$
 $stepResultsinR p p' (R0 \cup (R1 \cup R2)) \wedge$
 $((\alpha, \alpha') \in R0 \vee (\alpha, \alpha') \in R1 \vee (\alpha, \alpha') \in R2) \wedge$
 $dhequality-alternative d PP (pp (V!i)) m2 m2'$
by blast

qed

with $inR0$ Areflassump2 Up-To-Technique
have $SdlHPPB d PP (R0 \cup (R1 \cup R2))$
by auto

with $inR0$ show $\exists R. SdlHPPB d PP R \wedge ([c1;c2],[c1';c2']) \in R$
by auto

qed

theorem *Compositionality-Spawn:*

assumes *WWs-threads: WHATWHERE-Secure V*

assumes *uniPPspawn: unique-PPc (spawn_i V)*

shows *WHATWHERE-Secure [spawn_i V]*

proof (simp add: *WHATWHERE-Secure-def, auto*)

fix $d PP$

from *uniPPspawn* have *pp-difference: $\iota \notin \text{set} (PPV V)$*
by (simp add: *unique-PPc-def*)

— Step 1

from *WWs-threads* obtain R' where R' assump:

$SdlHPPB d PP R' \wedge (V, V) \in R'$

by (simp add: *WHATWHERE-Secure-def, auto*)

define R where $R = \{(V', V''). (V', V'') \in R' \wedge \text{set} (PPV V') \subseteq \text{set} (PPV V) \wedge \text{set} (PPV V'') \subseteq \text{set} (PPV V)\}$

from R' assump R -def $SdlHPPB$ -restricted-on- PP -is- $SdlHPPB$
have $SdlHPPR$: $SdlHPPB$ d PP R
by *force*

— Step 2

define $R0$ **where** $R0 = \{(sp1, sp2). \exists \iota' \iota'' V' V''.$
 $sp1 = [spawn_{\iota'} V'] \wedge sp2 = [spawn_{\iota''} V'']$
 $\wedge \iota' \notin set (PPV V) \wedge \iota'' \notin set (PPV V) \wedge (V', V'') \in R\}$

with R -def R' assump pp -difference **have** $inR0$:
 $([spawn_{\iota} V], [spawn_{\iota} V]) \in R0$
by *auto*

— Step 3

from $R0$ -def R -def R' assump **have** $Domain R0 \cap Domain R = \{\}$
by *auto*

with $commonArefl$ -subset- $commonDomain$
have $Areflassump$: $Arefl R0 \cap Arefl R \subseteq \{\emptyset\}$
by *force*

— Step 4

have $disjuptoR0$:

$disj$ - $dlHPP$ - $Bisimulation$ - Up - To - R' d PP R $R0$

proof (*simp add: disj-dlHPP-Bisimulation-Up-To-R'-def, auto*)

from $SdlHPPR$ **show** $SdlHPPB$ d PP R

by *auto*

next

from $SdlHPPR$ **have** $symR$: $sym R$

by (*simp add: Strong-dlHPP-Bisimulation-def*)

with $R0$ -def **show** $sym R0$

by (*simp add: sym-def, auto*)

next

from $SdlHPPR$ **have** $transR$: $trans R$

by (*simp add: Strong-dlHPP-Bisimulation-def*)

with $R0$ -def **show** $trans R0$

proof —

{

fix $V1 V2 V3$

assume $inR1$: $(V1, V2) \in R0$

assume $inR2$: $(V2, V3) \in R0$

from $inR1$ $R0$ -def **obtain** $W W' \iota \iota'$ **where** $p1$: $V1 = [spawn_{\iota} W]$

$\wedge V2 = [spawn_{\iota'} W'] \wedge \iota \notin set (PPV V) \wedge \iota' \notin set (PPV V)$

$\wedge (W, W') \in R$

by *auto*

with $inR2$ $R0$ -def **obtain** $W'' \iota''$ **where** $p2$: $V3 = [spawn_{\iota''} W'']$

$\wedge \iota'' \notin set (PPV V) \wedge (W', W'') \in R$

by *auto*


```

    from p1 p2 transR have (W, W'') ∈ R
      by (simp add: trans-def, auto)
    with p1 p2 R0-def have (V1, V3) ∈ R0
      by auto
  }
  thus ?thesis unfolding trans-def by blast
qed
next
fix V' V''
from SdlHPPR have eqLenR: (V', V'') ∈ R ⇒ length V' = length V''
  by (simp add: Strong-dlHPP-Bisimulation-def, auto)
with R0-def show (V', V'') ∈ R0 ⇒ length V' = length V''
  by auto
next
fix V' V'' i
assume inR0: (V', V'') ∈ R0
assume irange: i < length V'
from inR0 R0-def obtain ι' ι'' W' W''
  where R0pair: V' = [spawnι' W'] ∧ V'' = [spawnι'' W'']
  by auto
{
  fix m
  from spawnnextmem-exists-and-unique obtain m' where spawnnextmem:
    ∃ p α. ⟨spawnι' W', m⟩ →⟨α⟩ ⟨p, m⟩
    ∧ (∀ m''. (∃ p α. ⟨spawnι' W', m⟩ →⟨α⟩ ⟨p, m''⟩) → m'' = m)
  by force

  hence m = m'
    by (metis MWLsSteps-det.spawn)

  with spawnnextmem have eqnextmem:
    ⟦spawnι' W'⟧(m) = m
    by (simp add: NextMem-def, auto)
}

with R0pair irange show NDC d (V'!i)
  by (simp add: NDC-def)
next
fix V' V'' i m1 m1' m2 α p
assume inR0: (V', V'') ∈ R0
assume irange: i < length V'
assume step: ⟨V'!i, m1⟩ →⟨α⟩ ⟨p, m2⟩
assume dhequal: m1 ∼d, htchLocSet PP m1'

from inR0 R0-def obtain ι' ι'' W' W''
  where R0pair: V' = [spawnι' W']
  ∧ V'' = [spawnι'' W''] ∧ (W', W'') ∈ R
  by auto

```

with *step irange*
have *conc-step1*: $\alpha = W' \wedge p = \text{None} \wedge m2 = m1$
by (*simp, metis MWLsSteps-det-cases(6)*)

from *R0pair irange*
obtain $p' \alpha' m2'$ **where** *conc-step2*: $\langle V''!i, m1 \rangle \rightarrow \langle \alpha' \rangle \triangleright \langle p', m2 \rangle$
 $\wedge p' = \text{None} \wedge \alpha' = W'' \wedge m2' = m1'$
by (*simp, metis MWLsSteps-det.spawn*)

with *R0pair conc-step1 dhequal irange*
show $\exists p' \alpha' m2'. \langle V''!i, m1 \rangle \rightarrow \langle \alpha' \rangle \triangleright \langle p', m2 \rangle \wedge$
stepResultsinR $p p' (R0 \cup R) \wedge$
 $((\alpha, \alpha') \in R0 \vee (\alpha, \alpha') \in R) \wedge$
dhequality-alternative d PP (pp (V''!i)) m2 m2'
by (*simp add: stepResultsinR-def*
dhequality-alternative-def, auto)

qed
— Step 5

with *Areflassump Up-To-Technique*
have *SdlHPPB d PP (R0 \cup R)*
by *auto*

with *inR0* **show** $\exists R. \text{SdlHPPB } d \text{ PP } R \wedge$
 $([\text{spawn}_\iota V], [\text{spawn}_\iota V]) \in R$
by *auto*

qed

theorem *Compositionality-If*:
assumes *dind*: $\forall d. b \equiv_d b$
assumes *WWs-branch1*: *WHATWHERE-Secure [c1]*
assumes *WWs-branch2*: *WHATWHERE-Secure [c2]*
assumes *uniPPif*: *unique-PPc (if _{ι} b then c1 else c2 fi)*
shows *WHATWHERE-Secure [if _{ι} b then c1 else c2 fi]*
proof (*simp add: WHATWHERE-Secure-def, auto*)
fix *d PP*
from *uniPPif* **have** *nocommonPP*: $\text{set } (PPc \ c1) \cap \text{set } (PPc \ c2) = \{\}$
by (*simp add: unique-PPc-def*)

from *uniPPif* **have** *pp-difference*: $\iota \notin \text{set } (PPc \ c1) \cup \text{set } (PPc \ c2)$
by (*simp add: unique-PPc-def*)

from *WWs-branch1* **obtain** *R1'* **where** *R1'assump*:
SdlHPPB d PP R1' \wedge ([c1], [c1]) \in R1'
by (*simp add: WHATWHERE-Secure-def, auto*)

define *R1* **where** $R1 = \{(V, V'). (V, V') \in R1' \wedge \text{set } (PPV \ V) \subseteq \text{set } (PPc \ c1)$
 $\wedge \text{set } (PPV \ V') \subseteq \text{set } (PPc \ c1)\}$

from $R1'$ *assump* $R1$ -*def* $SdlHPPB$ -*restricted-on-PP-is-SdlHPPB*
have $SdlHPPR1$: $SdlHPPB$ d PP $R1$
by *force*

from WVs -*branch2* **obtain** $R2'$ **where** $R2'$ *assump*:
 $SdlHPPB$ d PP $R2' \wedge ([c2],[c2]) \in R2'$
by (*simp add: WHATWHERE-Secure-def, auto*)

define $R2$ **where** $R2 = \{(V, V'). (V, V') \in R2' \wedge set (PPV V) \subseteq set (PPc c2)$
 $\wedge set (PPV V') \subseteq set (PPc c2)\}$

from $R2'$ *assump* $R2$ -*def* $SdlHPPB$ -*restricted-on-PP-is-SdlHPPB*
have $SdlHPPR2$: $SdlHPPB$ d PP $R2$
by *force*

from $nocommonPP$ **have** $Domain R1 \cap Domain R2 \subseteq \{\emptyset\}$

by (*simp add: R1-def R2-def, auto,*
metis empty-subsetI inf-idem inf-mono set-eq-subset unique-V-uneq)

with $commonArefl$ -*subset-commonDomain*
have $Areflassump1$: $Arefl R1 \cap Arefl R2 \subseteq \{\emptyset\}$
by *force*

with $SdlHPPR1$ $SdlHPPR2$ $Union$ - $Strong$ - $dlHPP$ - $Bisim$ **have** $SdlHPPR1R2$:
 $SdlHPPB$ d PP $(R1 \cup R2)$
by *force*

define R **where** $R = (R1 \cup R2) \cup \{(\emptyset, \emptyset)\}$

define $R0$ **where** $R0 = \{(i1, i2). \exists \iota' \iota'' b' b'' c1' c1'' c2' c2''.$
 $i1 = [if_{\iota'} b' then c1' else c2' fi]$
 $\wedge i2 = [if_{\iota''} b'' then c1'' else c2'' fi]$
 $\wedge \iota' \notin (set (PPc c1) \cup set (PPc c2))$
 $\wedge \iota'' \notin (set (PPc c1) \cup set (PPc c2)) \wedge ([c1'], [c1'']) \in R1$
 $\wedge ([c2'], [c2'']) \in R2 \wedge b' \equiv_d b''\}$

with R -*def* $R1$ -*def* $R1'$ *assump* $R2$ -*def* $R2'$ *assump* pp -*difference* $dind$
have $inR0$: $([if_{\iota} b then c1 else c2 fi], [if_{\iota} b then c1 else c2 fi]) \in R0$
by *auto*

from $R0$ -*def* R -*def* $R1$ -*def* $R2$ -*def*
have $Domain R0 \cap Domain R = \{\}$
by *auto*

with $commonArefl$ -*subset-commonDomain*
have $Areflassump2$: $Arefl R0 \cap Arefl R \subseteq \{\emptyset\}$

by force

have *disjuptoR0*:

disj-dlHPP-Bisimulation-Up-To-R' d PP R R0

proof (simp add: *disj-dlHPP-Bisimulation-Up-To-R'-def*, auto)

from *SdlHPPR1R2 adding-emptypair-keeps-SdlHPPB*

show *SdlHPPB d PP R*

by (simp add: *R-def*)

next

from *SdlHPPR1* have *symR1: sym R1*

by (simp add: *Strong-dlHPP-Bisimulation-def*)

from *SdlHPPR2* have *symR2: sym R2*

by (simp add: *Strong-dlHPP-Bisimulation-def*)

from *symR1 symR2 d-indistinguishable-sym R0-def* show *sym R0*

by (simp add: *sym-def*, *fastforce*)

next

from *SdlHPPR1* have *transR1: trans R1*

by (simp add: *Strong-dlHPP-Bisimulation-def*)

from *SdlHPPR2* have *transR2: trans R2*

by (simp add: *Strong-dlHPP-Bisimulation-def*)

show *trans R0*

proof –

{

fix $V' V'' V'''$

assume $p1: (V', V'') \in R0$

assume $p2: (V'', V''') \in R0$

from $p1$ *R0-def* obtain $\iota' \iota'' b' b'' c1' c1'' c2' c2''$ where

passump1: $V' = [if_{\iota'} b' then c1' else c2' fi]$

$\wedge V'' = [if_{\iota''} b'' then c1'' else c2'' fi]$

$\wedge \iota' \notin (set (PPc c1) \cup set (PPc c2))$

$\wedge \iota'' \notin (set (PPc c1) \cup set (PPc c2))$

$\wedge ([c1'], [c1'']) \in R1 \wedge ([c2'], [c2'']) \in R2$

$\wedge b' \equiv_d b''$

by force

with $p2$ *R0-def* obtain $\iota''' b''' c1''' c2'''$ where

passump2: $V''' = [if_{\iota'''} b''' then c1''' else c2''' fi]$

$\wedge \iota''' \notin (set (PPc c1) \cup set (PPc c2))$

$\wedge ([c1'''], [c1''']) \in R1 \wedge ([c2'''], [c2''']) \in R2$

$\wedge b'' \equiv_d b'''$

by force

with *passump1* *transR1* *transR2* *d-indistinguishable-trans*

have $([c1'], [c1''']) \in R1 \wedge ([c2'], [c2''']) \in R2$

$\wedge b' \equiv_d b'''$

by (*metis transD*)

with *passump1* *passump2* *R0-def* have $(V', V''') \in R0$

```

    by auto
  }
  thus ?thesis unfolding trans-def by blast
qed
next
fix V V'
assume inR0: (V, V') ∈ R0
with R0-def show length V = length V' by auto
next
fix V' V'' i
assume inR0: (V', V'') ∈ R0
assume irange: i < length V'
assume notIDC: ¬ IDC d (V!i) (htchLoc (pp (V!i)))

from inR0 R0-def obtain ι' ι'' b' b'' c1' c1'' c2' c2''
  where R0pair: V' = [ifι' b' then c1' else c2' fi]
  ∧ V'' = [ifι'' b'' then c1'' else c2'' fi]
  ∧ ι' ∉ set (PPc c1) ∪ set (PPc c2)
  ∧ ι'' ∉ set (PPc c1) ∪ set (PPc c2)
  ∧ ([c1↑], [c1'↑]) ∈ R1 ∧ ([c2↑], [c2'↑]) ∈ R2
  ∧ b' ≡d b''
  by force

have NDC d (ifι' b' then c1' else c2' fi)
proof -
  {
    fix m
    from nextmem-exists-and-unique obtain m' p α where ifnextmem:
      ⟨ifι' b' then c1' else c2' fi, m⟩ →⟨α⟩ ⟨p, m↑⟩
      ∧ (∀ m''. (∃ p α. ⟨ifι' b' then c1' else c2' fi, m⟩ →⟨α⟩ ⟨p, m''↑⟩)
        → m'' = m')
      by blast

    hence m = m'
      by (metis MWLsSteps-det.iffalse MWLsSteps-det.iftrue)

    with ifnextmem have eqnextmem:
      [[ifι' b' then c1' else c2' fi]](m) = m
      by (simp add: NextMem-def, auto)
  }
  thus ?thesis
  by (simp add: NDC-def)
qed

with R0pair irange show NDC d (V!i)
  by simp
next
fix V' V'' i m1 m1' m2 α p
assume inR0: (V', V'') ∈ R0

```

assume *irange*: $i < \text{length } V'$
assume *step*: $\langle V'!i, m1 \rangle \rightarrow \langle \alpha \rangle \langle p, m2 \rangle$
assume *dhequal*: $m1 \sim_{d, \text{htchLocSet } PP} m1'$

from *inR0 R0-def* **obtain** $\iota' \iota'' b' b'' c1' c1'' c2' c2''$
where *R0pair*: $V' = [\text{if } \iota' b' \text{ then } c1' \text{ else } c2' \text{ fi}]$
 $\wedge V'' = [\text{if } \iota'' b'' \text{ then } c1'' \text{ else } c2'' \text{ fi}]$
 $\wedge \iota' \notin \text{set } (PPc \ c1) \cup \text{set } (PPc \ c2)$
 $\wedge \iota'' \notin \text{set } (PPc \ c1) \cup \text{set } (PPc \ c2)$
 $\wedge ([c1', [c1'']] \in R1 \wedge ([c2', [c2'']] \in R2$
 $\wedge b' \equiv_d b''$
by *force*

with *R0pair irange step* **have** *case-distinction*:
 $(p = \text{Some } c1' \wedge BMap (E \ b' \ m1) = \text{True})$
 $\vee (p = \text{Some } c2' \wedge BMap (E \ b' \ m1) = \text{False})$
by (*simp, metis MWLsSteps-det-cases(4)*)

moreover
— Case 1: b evaluates to True
{
assume *passump*: $p = \text{Some } c1'$
assume *eval*: $BMap (E \ b' \ m1) = \text{True}$

from *R0pair step irange* **have** *stepconcl*: $\alpha = [] \wedge m2 = m1$
by (*simp, metis MWLs-semantics.MWLsSteps-det-cases(4)*)

from *eval R0pair dhequal* **have** *eval'*: $BMap (E \ b'' \ m1') = \text{True}$
by (*simp add: d-indistinguishable-def dH-equal-def, auto*)

hence *step'*: $\langle \text{if } \iota'' b'' \text{ then } c1'' \text{ else } c2'' \text{ fi}, m1' \rangle \rightarrow \langle [] \rangle$
 $\langle \text{Some } c1'', m1' \rangle$
by (*simp add: MWLsSteps-det.iftrue*)

with *passump R0pair R-def dhequal stepconcl irange*
have $\exists p' \alpha' m2'. \langle V'!i, m1' \rangle \rightarrow \langle \alpha' \rangle \langle p', m2' \rangle \wedge$
 $\text{stepResultsinR } p \ p' (R0 \cup R) \wedge$
 $((\alpha, \alpha') \in R0 \vee (\alpha, \alpha') \in R) \wedge$
 $\text{dhequality-alternative } d \ PP (pp (V'!i)) \ m2 \ m2'$
by (*simp add: stepResultsinR-def dhequality-alternative-def, auto*)

}

moreover
— Case 2: b evaluates to False
{
assume *passump*: $p = \text{Some } c2'$
assume *eval*: $BMap (E \ b' \ m1) = \text{False}$

from *R0pair step irange* **have** *stepconcl*: $\alpha = [] \wedge m2 = m1$
by (*simp, metis MWLs-semantics.MWLsSteps-det-cases(4)*)

from *eval R0pair dhequal* **have** *eval': BMap (E b'' m1') = False*
by (*simp add: d-indistinguishable-def dH-equal-def, auto*)

hence *step': ⟨if_{l''} b'' then c1'' else c2'' fi, m1'⟩ →<□>*
⟨Some c2'', m1'⟩
by (*simp add: MWLsSteps-det.iffalse*)

with *passump R0pair R-def dhequal stepconcl irange*
have $\exists p' \alpha' m2'. \langle V''!i, m1' \rangle \rightarrow \langle \alpha' \rangle \langle p', m2' \rangle \wedge$
stepResultsinR p p' (R0 ∪ R) ∧
 $((\alpha, \alpha') \in R0 \vee (\alpha, \alpha') \in R) \wedge$
dhequality-alternative d PP (pp (V''!i)) m2 m2'
by (*simp add: stepResultsinR-def dhequality-alternative-def,*
auto)

}

ultimately

show $\exists p' \alpha' m2'. \langle V''!i, m1' \rangle \rightarrow \langle \alpha' \rangle \langle p', m2' \rangle \wedge$
stepResultsinR p p' (R0 ∪ R) ∧
 $((\alpha, \alpha') \in R0 \vee (\alpha, \alpha') \in R) \wedge$
dhequality-alternative d PP (pp (V''!i)) m2 m2'
by *auto*

qed

with *Areflassump2 Up-To-Technique*
have *SdlHPPB d PP (R0 ∪ R)*
by *auto*

with *inR0* **show** $\exists R. \text{SdlHPPB } d \text{ PP } R$
 $\wedge ((\text{if}_l b \text{ then } c1 \text{ else } c2 \text{ fi}), (\text{if}_l b \text{ then } c1 \text{ else } c2 \text{ fi})) \in R$
by *auto*

qed

theorem *Compositionality-While:*

assumes *dind: $\forall d. b \equiv_d b$*

assumes *WWs-body: WHATWHERE-Secure [c]*

assumes *uniPPwhile: unique-PPc (while_l b do c od)*

shows *WHATWHERE-Secure [while_l b do c od]*

proof (*simp add: WHATWHERE-Secure-def, auto*)

fix *d PP*

from *uniPPwhile* **have** *pp-difference: $\iota \notin \text{set } (PPc \ c)$*
by (*simp add: unique-PPc-def*)

from *WWs-body* **obtain** *R' where R'assump:*

SdlHPPB d PP R' ∧ ([c], [c]) ∈ R'

by (*simp add: WHATWHERE-Secure-def, auto*)

— add the empty pair because it is needed later in the proof
define R **where** $R = \{(V, V'). (V, V') \in R' \wedge \text{set}(PPV V) \subseteq \text{set}(PPc c) \wedge \text{set}(PPV V') \subseteq \text{set}(PPc c)\} \cup \{([], [])\}$

with R' *assump SdlHPPB-restricted-on-PP-is-SdlHPPB*
adding-emptypair-keeps-SdlHPPB

have $SdlHPPR$: $SdlHPPB$ d PP R

proof —

from R' *assump SdlHPPB-restricted-on-PP-is-SdlHPPB* **have**

$SdlHPPB$ d PP

$\{(V, V'). (V, V') \in R' \wedge \text{set}(PPV V) \subseteq \text{set}(PPc c) \wedge \text{set}(PPV V') \subseteq \text{set}(PPc c)\}$

by *force*

with *adding-emptypair-keeps-SdlHPPB* **have**

$SdlHPPB$ d PP

$\{(V, V'). (V, V') \in R' \wedge \text{set}(PPV V) \subseteq \text{set}(PPc c) \wedge \text{set}(PPV V') \subseteq \text{set}(PPc c)\} \cup \{([], [])\}$

by *auto*

with R -*def* **show** *?thesis*

by *auto*

qed

define $R1$ **where** $R1 = \{(w1, w2). \exists \iota \iota' b b' c1 c1' c2 c2'\}$

$w1 = [c1; (\text{while}_{\iota} b \text{ do } c2 \text{ od})]$

$\wedge w2 = [c1'; (\text{while}_{\iota'} b' \text{ do } c2' \text{ od})]$

$\wedge \iota \notin \text{set}(PPc c) \wedge \iota' \notin \text{set}(PPc c)$

$\wedge ([c1], [c1']) \in R \wedge ([c2], [c2']) \in R$

$\wedge b \equiv_d b'\}$

define $R2$ **where** $R2 = \{(w1, w2). \exists \iota \iota' b b' c1 c1'\}$

$w1 = [\text{while}_{\iota} b \text{ do } c1 \text{ od}]$

$\wedge w2 = [\text{while}_{\iota'} b' \text{ do } c1' \text{ od}]$

$\wedge \iota \notin \text{set}(PPc c) \wedge \iota' \notin \text{set}(PPc c) \wedge$

$([c1], [c1']) \in R \wedge b \equiv_d b'\}$

define $R0$ **where** $R0 = R1 \cup R2$

from $R2$ -*def* R -*def* R' *assump pp-difference dind*

have $\text{in}R2$: $([\text{while}_{\iota} b \text{ do } c \text{ od}], [\text{while}_{\iota} b \text{ do } c \text{ od}]) \in R2$

by *auto*

from $R0$ -*def* $R1$ -*def* $R2$ -*def* R -*def* R' *assump* **have**

$\text{Domain } R0 \cap \text{Domain } R = \{\}$

by *auto*

with *commonArefl-subset-commonDomain*

have $\text{Arefl} \text{assump}$: $\text{Arefl } R0 \cap \text{Arefl } R \subseteq \{\}$

by force

— show some facts about R1 and R2 needed later in the proof at several positions

```
from SdlHPPR have symR: sym R
  by (simp add: Strong-dlHPP-Bisimulation-def)
from symR R1-def d-indistinguishable-sym have symR1: sym R1
  by (simp add: sym-def, fastforce)
from symR R2-def d-indistinguishable-sym have symR2: sym R2
  by (simp add: sym-def, fastforce)
```

```
have disjuptoR1R2:
  disj-dlHPP-Bisimulation-Up-To-R' d PP R R0
proof (simp add: disj-dlHPP-Bisimulation-Up-To-R'-def, auto)
  from SdlHPPR show SdlHPPB d PP R
    by auto
next
  from symR1 symR2 R0-def show sym R0
    by (simp add: sym-def)
next
  from SdlHPPR have transR: trans R
    by (simp add: Strong-dlHPP-Bisimulation-def)
  have transR1: trans R1
  proof –
    {
      fix V V' V''
      assume p1: (V, V') ∈ R1
      assume p2: (V', V'') ∈ R1

      from p1 R1-def obtain ι ι' b b' c1 c1' c2 c2' where passump1:
        V = [c1;(whileι b do c2 od)]
        ∧ V' = [c1';(whileι' b' do c2' od)]
        ∧ ι ∉ set (PPc c) ∧ ι' ∉ set (PPc c)
        ∧ ([c1],[c1']) ∈ R ∧ ([c2],[c2']) ∈ R
        ∧ b ≡d b'
      by force

      with p2 R1-def obtain ι'' b'' c1'' c2'' where passump2:
        V'' = [c1'';(whileι'' b'' do c2'' od)] ∧ ι'' ∉ set (PPc c)
        ∧ ([c1''],[c1'']) ∈ R ∧ ([c2''],[c2'']) ∈ R
        ∧ b' ≡d b''
      by force

      with passump1 transR d-indistinguishable-trans
      have ([c1],[c1'']) ∈ R ∧ ([c2],[c2'']) ∈ R ∧ b ≡d b''
      by (metis trans-def)

      with passump1 passump2 R1-def have (V, V'') ∈ R1
      by auto
    }
}
```

thus *?thesis unfolding trans-def by blast*
qed

have *transR2: trans R2*

proof –

{

fix $V V' V''$

assume $p1: (V, V') \in R2$

assume $p2: (V', V'') \in R2$

from $p1$ *R2-def* **obtain** $\iota \iota' b b' c1 c1'$ **where** *passump1:*

$V = [\text{while}_{\iota} b \text{ do } c1 \text{ od}]$

$\wedge V' = [\text{while}_{\iota'} b' \text{ do } c1' \text{ od}]$

$\wedge \iota \notin \text{set } (PPc \ c) \wedge \iota' \notin \text{set } (PPc \ c)$

$\wedge ([c1], [c1']) \in R \wedge b \equiv_d b'$

by *force*

with $p2$ *R2-def* **obtain** $\iota'' b'' c1''$ **where** *passump2:*

$V'' = [\text{while}_{\iota''} b'' \text{ do } c1'' \text{ od}] \wedge \iota'' \notin \text{set } (PPc \ c)$

$\wedge ([c1'], [c1'']) \in R \wedge b' \equiv_d b''$

by *force*

with *passump1 transR d-indistinguishable-trans*

have $([c1], [c1'']) \in R \wedge b \equiv_d b''$

by (*metis trans-def*)

with *passump1 passump2 R2-def* **have** $(V, V'') \in R2$

by *auto*

}

thus *?thesis unfolding trans-def by blast*

qed

from *SdlHPPR* **have** *eqlenR: $\forall (V, V') \in R. \text{length } V = \text{length } V'$*

by (*simp add: Strong-dlHPP-Bisimulation-def*)

from *R1-def eqlenR* **have** *eqlenR1: $\forall (V, V') \in R1. \text{length } V = \text{length } V'$*

by *auto*

from *R2-def eqlenR* **have** *eqlenR2: $\forall (V, V') \in R2. \text{length } V = \text{length } V'$*

by *auto*

from *R1-def R2-def* **have** $\text{Domain } R1 \cap \text{Domain } R2 = \{\}$

by *auto*

with *commonArefl-subset-commonDomain* **have** *Arefl-a:*

$\text{Arefl } R1 \cap \text{Arefl } R2 = \{\}$

by *force*

with *symR1 symR2 transR1 transR2 eqlenR1 eqlenR2 trans-RuR'*

have $\text{trans } (R1 \cup R2)$

by *force*

```

with R0-def show trans R0 by auto
next
fix V V'
assume inR0:  $(V, V') \in R0$ 
with R0-def R1-def R2-def show  $\text{length } V = \text{length } V'$  by auto
next
fix V V' i
assume inR0:  $(V, V') \in R0$ 
assume irange:  $i < \text{length } V$ 
assume notIDC:  $\neg IDC\ d\ (V!i)$ 
  (htchLoc (pp (V!i)))

from inR0 R0-def R1-def R2-def obtain  $\iota\ \iota'\ b\ b'\ c1\ c1'\ c2\ c2'$ 
  where R0pair:  $((V = [c1;(\text{while}_{\iota}\ b\ \text{do}\ c2\ \text{od})])$ 
 $\wedge V' = [c1';(\text{while}_{\iota'}\ b'\ \text{do}\ c2'\ \text{od})])$ 
 $\vee (V = [\text{while}_{\iota}\ b\ \text{do}\ c1\ \text{od}] \wedge V' = [\text{while}_{\iota'}\ b'\ \text{do}\ c1'\ \text{od}])$ 
 $\wedge \iota \notin \text{set } (PPc\ c) \wedge \iota' \notin \text{set } (PPc\ c)$ 
 $\wedge ([c1],[c1']) \in R \wedge ([c2],[c2']) \in R$ 
 $\wedge b \equiv_d b'$ 
  by force

with irange SdlHPPR strongdlHPPB-NDCIDCaux
[of d PP R [c1] [c1'] i]
have c1NDCIDC:
   $NDC\ d\ c1 \vee IDC\ d\ c1$  (htchLoc (pp c1))
  by auto

— first alternative for V and V'
have case1:  $NDC\ d\ (c1;(\text{while}_{\iota}\ b\ \text{do}\ c2\ \text{od})) \vee$ 
 $IDC\ d\ (c1;(\text{while}_{\iota}\ b\ \text{do}\ c2\ \text{od}))$ 
(htchLoc (pp ( $c1;(\text{while}_{\iota}\ b\ \text{do}\ c2\ \text{od})$ ))))
proof —
  have eqnextmem:  $\bigwedge m. \llbracket c1;(\text{while}_{\iota}\ b\ \text{do}\ c2\ \text{od}) \rrbracket(m) = \llbracket c1 \rrbracket(m)$ 
proof —
  fix m
from nextmem-exists-and-unique obtain m' where c1nextmem:
 $\exists p\ \alpha. \langle c1, m \rangle \rightarrow \langle \alpha \rangle \langle p, m' \rangle$ 
 $\wedge (\forall m''. (\exists p\ \alpha. \langle c1, m \rangle \rightarrow \langle \alpha \rangle \langle p, m'' \rangle) \longrightarrow m'' = m')$ 
  by force

hence eqdir1:  $\llbracket c1 \rrbracket(m) = m'$ 
  by (simp add: NextMem-def, auto)

from c1nextmem obtain p alpha where  $\langle c1, m \rangle \rightarrow \langle \alpha \rangle \langle p, m' \rangle$ 
  by auto

with c1nextmem have
 $\exists p\ \alpha. \langle c1;(\text{while}_{\iota}\ b\ \text{do}\ c2\ \text{od}), m \rangle \rightarrow \langle \alpha \rangle \langle p, m' \rangle$ 
 $\wedge (\forall m''. (\exists p\ \alpha. \langle c1;(\text{while}_{\iota}\ b\ \text{do}\ c2\ \text{od}), m \rangle \rightarrow \langle \alpha \rangle \langle p, m'' \rangle))$ 

```

```

    → m'' = m')
  by (auto, metis MWLsSteps-det.seq1 MWLsSteps-det.seq2
      option.exhaust, metis MWLsSteps-det-cases(3))

  hence eqdir2:  $\llbracket c1;(\text{while}_l b \text{ do } c2 \text{ od}) \rrbracket(m) = m'$ 
    by (simp add: NextMem-def, auto)

  with eqdir1 show  $\llbracket c1;(\text{while}_l b \text{ do } c2 \text{ od}) \rrbracket(m) = \llbracket c1 \rrbracket(m)$ 
    by auto
  qed

  have eqpp:  $pp (c1;(\text{while}_l b \text{ do } c2 \text{ od})) = pp c1$ 
    by simp

  with c1NDCIDC eqnextmem show
    NDC d (c1;(\text{while}_l b \text{ do } c2 \text{ od}))  $\vee$ 
    IDC d (c1;(\text{while}_l b \text{ do } c2 \text{ od}))
    (htchLoc (pp (c1;(\text{while}_l b \text{ do } c2 \text{ od}))))
    by (simp add: IDC-def NDC-def)
  qed

— second alternative for  $\forall V V'$ 
  have case2: NDC d (whilel b do c1 od)
  proof -
    {
      fix m
      from nextmem-exists-and-unique obtain m' p  $\alpha$ 
        where whilenextmem:  $\langle \text{while}_l b \text{ do } c1 \text{ od}, m \rangle \rightarrow \langle \alpha \rangle \langle p, m' \rangle$ 
           $\wedge (\forall m''. (\exists p \alpha. \langle \text{while}_l b \text{ do } c1 \text{ od}, m \rangle \rightarrow \langle \alpha \rangle \langle p, m'' \rangle) \rightarrow m'' = m')$ 
        by blast

      hence m = m'
        by (metis MWLsSteps-det.whilefalse MWLsSteps-det.whiletrue)

      with whilenextmem have eqnextmem:
         $\llbracket \text{while}_l b \text{ do } c1 \text{ od} \rrbracket(m) = m$ 
        by (simp add: NextMem-def, auto)
    }
  thus NDC d (whilel b do c1 od)
    by (simp add: NDC-def)
  qed

  from R0pair case1 case2 irange notIDC
  show NDC d (V!i)
    by force
  next
  fix V V' i m1 m1' m2  $\alpha$  p
  assume inR0: (V, V')  $\in$  R0

```

assume *irange*: $i < \text{length } V$
assume *step*: $\langle V!i, m1 \rangle \rightarrow \langle \alpha \rangle \langle p, m2 \rangle$
assume *dhequal*: $m1 \sim_{d, \text{htchLocSet } PP} m1'$

from *inR0 R0-def R1-def R2-def* **obtain** $\iota \iota' b b' c1 c1' c2 c2'$
where *R0pair*: $((V = [c1; (\text{while}_{\iota} b \text{ do } c2 \text{ od})])$
 $\wedge V' = [c1'; (\text{while}_{\iota'} b' \text{ do } c2' \text{ od})])$
 $\vee (V = [\text{while}_{\iota} b \text{ do } c1 \text{ od}] \wedge V' = [\text{while}_{\iota'} b' \text{ do } c1' \text{ od}])$
 $\wedge \iota \notin \text{set } (PPc \ c) \wedge \iota' \notin \text{set } (PPc \ c)$
 $\wedge ([c1], [c1']) \in R \wedge ([c2], [c2']) \in R$
 $\wedge b \equiv_d b'$
by *force*

— Case 1: V and V' are sequential commands

have *case1*: $\llbracket V = [c1; (\text{while}_{\iota} b \text{ do } c2 \text{ od})];$
 $V' = [c1'; (\text{while}_{\iota'} b' \text{ do } c2' \text{ od})] \rrbracket$
 $\implies \exists p' \alpha' m2'. \langle V!i, m1 \rangle \rightarrow \langle \alpha' \rangle \langle p', m2' \rangle \wedge$
stepResultsinR $p \ p' (R0 \cup R) \wedge ((\alpha, \alpha') \in R0 \vee (\alpha, \alpha') \in R) \wedge$
dhequality-alternative $d \ PP (pp \ (V!i)) \ m2 \ m2'$

proof —

assume *Vassump*: $V = [c1; (\text{while}_{\iota} b \text{ do } c2 \text{ od})]$
assume *V'assump*: $V' = [c1'; (\text{while}_{\iota'} b' \text{ do } c2' \text{ od})]$

have *eqpp*: $pp \ (c1; (\text{while}_{\iota} b \text{ do } c2 \text{ od})) = pp \ c1$
by *simp*

from *Vassump irange step irange* **obtain** $c3$

where *case-distinction*:
 $(p = \text{Some } (\text{while}_{\iota} b \text{ do } c2 \text{ od}) \wedge \langle c1, m1 \rangle \rightarrow \langle \alpha \rangle \langle \text{None}, m2 \rangle)$
 $\vee (p = \text{Some } (c3; (\text{while}_{\iota} b \text{ do } c2 \text{ od}))$
 $\wedge \langle c1, m1 \rangle \rightarrow \langle \alpha \rangle \langle \text{Some } c3, m2 \rangle)$
by (*simp, metis MWLsSteps-det-cases(3)*)

moreover

— Case 1a: first command terminates

$\{$
assume *passump*: $p = \text{Some } (\text{while}_{\iota} b \text{ do } c2 \text{ od})$
assume *stepassump*: $\langle c1, m1 \rangle \rightarrow \langle \alpha \rangle \langle \text{None}, m2 \rangle$

with *R0pair SdlHPPR dhequal*

strongdlHPPB-aux[*of* $d \ PP \ R$

$- [c1] [c1'] \ m1 \ \alpha \ \text{None} \ m2 \ m1']$

obtain $p' \ \alpha' \ m2'$ **where** *c1c1'reason*:

$p' = \text{None} \wedge \langle c1', m1' \rangle \rightarrow \langle \alpha' \rangle \langle p', m2' \rangle \wedge (\alpha, \alpha') \in R \wedge$
dhequality-alternative $d \ PP (pp \ c1) \ m2 \ m2'$

by (*simp add: stepResultsinR-def, fastforce*)

hence *conclpart*:

$\langle c1'; (\text{while}_{\iota'} b' \text{ do } c2' \text{ od}), m1' \rangle$
 $\rightarrow \langle \alpha' \rangle \langle \text{Some } (\text{while}_{\iota'} b' \text{ do } c2' \text{ od}), m2' \rangle$

by (*auto, simp add: MWLsSteps-det.seq1*)

from *R0pair R0-def R2-def* **have**
 ($[while_{\iota} b \text{ do } c2 \text{ od}], [while_{\iota'} b' \text{ do } c2' \text{ od}]$) $\in R0$
by *simp*

with *passump V'assump Vassump eqpp conclpart c1c1'reason irange*
have $\exists p' \alpha' m2'. \langle V!i, m1 \rangle \rightarrow \triangleleft \alpha' \triangleright \langle p', m2 \rangle \wedge$
stepResultsinR $p \ p' (R0 \cup R) \wedge ((\alpha, \alpha') \in R0 \vee$
 $(\alpha, \alpha') \in R) \wedge$
dhequality-alternative d PP (pp (V!i)) m2 m2'
by (*simp add: stepResultsinR-def, auto*)

}

moreover

— Case 1b: first command does not terminate

{

assume *passump*: $p = \text{Some } (c3; (while_{\iota} b \text{ do } c2 \text{ od}))$
assume *steppassump*: $\langle c1, m1 \rangle \rightarrow \triangleleft \alpha \triangleright \langle \text{Some } c3, m2 \rangle$

with *R0pair SdlHPPR dhequal strongdllHPPB-aux[of d PP R*
- [c1] [c1'] m1 α Some c3 m2 m1']
obtain $p' c3' \alpha' m2'$ **where** *c1c1'reason*:
 $p' = \text{Some } c3' \wedge \langle c1', m1 \rangle \rightarrow \triangleleft \alpha' \triangleright \langle p', m2 \rangle \wedge$
 $([c3], [c3']) \in R \wedge (\alpha, \alpha') \in R \wedge$
dhequality-alternative d PP (pp c1) m2 m2'
by (*simp add: stepResultsinR-def, fastforce*)

hence *conclpart*:
 $\langle c1'; (while_{\iota'} b' \text{ do } c2' \text{ od}), m1 \rangle \rightarrow \triangleleft \alpha' \triangleright$
 $\langle \text{Some } (c3'; while_{\iota'} b' \text{ do } c2' \text{ od}), m2 \rangle$
by (*auto, simp add: MWLsSteps-det.seq2*)

from *c1c1'reason R0pair R0-def R1-def* **have**
 ($[c3; while_{\iota} b \text{ do } c2 \text{ od}], [c3'; while_{\iota'} b' \text{ do } c2' \text{ od}]$) $\in R0$
by *simp*

with *passump V'assump Vassump eqpp conclpart c1c1'reason irange*
have $\exists p' \alpha' m2'. \langle V!i, m1 \rangle \rightarrow \triangleleft \alpha' \triangleright \langle p', m2 \rangle \wedge$
stepResultsinR $p \ p' (R0 \cup R) \wedge$
 $((\alpha, \alpha') \in R0 \vee (\alpha, \alpha') \in R) \wedge$
dhequality-alternative d PP (pp (V!i)) m2 m2'
by (*simp add: stepResultsinR-def, auto*)

}

ultimately show $\exists p' \alpha' m2'. \langle V!i, m1 \rangle \rightarrow \triangleleft \alpha' \triangleright \langle p', m2 \rangle \wedge$
stepResultsinR $p \ p' (R0 \cup R) \wedge$
 $((\alpha, \alpha') \in R0 \vee (\alpha, \alpha') \in R) \wedge$
dhequality-alternative d PP (pp (V!i)) m2 m2'

by *auto*
qed

— Case 2: V and V' are while commands

have *case2*: $\llbracket V = [\text{while}_\ell b \text{ do } c1 \text{ od}]; V' = [\text{while}_{\ell'} b' \text{ do } c1' \text{ od}] \rrbracket$
 $\implies \exists p' \alpha' m2'. \langle V!i, m1 \rangle \rightarrow \triangleleft \alpha' \triangleright \langle p', m2 \rangle \wedge$
stepResultsinR $p p' (R0 \cup R) \wedge ((\alpha, \alpha') \in R0 \vee (\alpha, \alpha') \in R) \wedge$
dhequality-alternative d PP (pp (V!i)) m2 m2'

proof –

assume *Vassump*: $V = [\text{while}_\ell b \text{ do } c1 \text{ od}]$
assume *V'assump*: $V' = [\text{while}_{\ell'} b' \text{ do } c1' \text{ od}]$

from *Vassump irange step* **have** *case-distinction*:

$(p = \text{None} \wedge \text{BMap } (E b m1) = \text{False})$
 $\vee p = (\text{Some } (c1; \text{while}_\ell b \text{ do } c1 \text{ od})) \wedge \text{BMap } (E b m1) = \text{True}$
by (*simp, metis MWLsSteps-det-cases(5) option.simps(2)*)

moreover

— Case 2a: b evaluates to False

{

assume *passump*: $p = \text{None}$
assume *eval*: $\text{BMap } (E b m1) = \text{False}$

with *Vassump step irange* **have** *stepconcl*: $\alpha = \square \wedge m2 = m1$
by (*simp, metis (no-types) MWLsSteps-det-cases(5)*)

from *eval R0pair dhequal* **have** *eval'*: $\text{BMap } (E b' m1') = \text{False}$
by (*simp add: d-indistinguishable-def dH-equal-def, auto*)

hence $\langle \text{while}_{\ell'} b' \text{ do } c1' \text{ od}, m1 \rangle \rightarrow \triangleleft \square \triangleright \langle \text{None}, m1 \rangle$
by (*simp add: MWLsSteps-det.whilefalse*)

with *passump R-def Vassump V'assump stepconcl dhequal irange*

have $\exists p' \alpha' m2'. \langle V!i, m1 \rangle \rightarrow \triangleleft \alpha' \triangleright \langle p', m2 \rangle \wedge$
stepResultsinR $p p' (R0 \cup R) \wedge ((\alpha, \alpha') \in R0 \vee (\alpha, \alpha') \in R) \wedge$
dhequality-alternative d PP (pp (V!i)) m2 m2'

by (*simp add: stepResultsinR-def dhequality-alternative-def, auto*)

}

moreover

— Case 2b: b evaluates to True

{

assume *passump*: $p = (\text{Some } (c1; \text{while}_\ell b \text{ do } c1 \text{ od}))$
assume *eval*: $\text{BMap } (E b m1) = \text{True}$

with *Vassump step irange* **have** *stepconcl*: $\alpha = \square \wedge m2 = m1$
by (*simp, metis (no-types) MWLsSteps-det-cases(5)*)

from *eval R0pair dhequal* **have** *eval'*:
 $\text{BMap } (E b' m1') = \text{True}$

by (*simp add: d-indistinguishable-def dH-equal-def, auto*)

hence $\langle \text{step}' : \langle \text{while}_{\iota'} b' \text{ do } c1' \text{ od}, m1 \rangle \rightarrow \langle \square \rangle \rangle$
 $\langle \text{Some } (c1'; \text{while}_{\iota'} b' \text{ do } c1' \text{ od}), m1 \rangle$
by (*simp add: MWLSteps-det.whiletrue*)

from *R1-def R0pair* **have** *inR1*:
 $([c1; \text{while}_{\iota} b \text{ do } c1 \text{ od}], [c1'; \text{while}_{\iota'} b' \text{ do } c1' \text{ od}]) \in R1$
by *auto*

with *step' R0-def passump R-def Vassump V'assump*
stepconcl dhequal irange
have $\exists p' \alpha' m2'. \langle V!i, m1 \rangle \rightarrow \langle \alpha' \rangle \langle p', m2 \rangle \wedge$
stepResultsinR p p' (R0 \cup R) $\wedge ((\alpha, \alpha') \in R0 \vee (\alpha, \alpha') \in R) \wedge$
dhequality-alternative d PP (pp (V!i)) m2 m2'
by (*simp add: stepResultsinR-def dhequality-alternative-def,*
auto)
}
ultimately
show $\exists p' \alpha' m2'. \langle V!i, m1 \rangle \rightarrow \langle \alpha' \rangle \langle p', m2 \rangle \wedge$
stepResultsinR p p' (R0 \cup R) $\wedge ((\alpha, \alpha') \in R0 \vee (\alpha, \alpha') \in R) \wedge$
dhequality-alternative d PP (pp (V!i)) m2 m2'
by *auto*
qed

with *case1 R0pair* **show** $\exists p' \alpha' m2'. \langle V!i, m1 \rangle \rightarrow \langle \alpha' \rangle \langle p', m2 \rangle \wedge$
stepResultsinR p p' (R0 \cup R) $\wedge ((\alpha, \alpha') \in R0 \vee (\alpha, \alpha') \in R) \wedge$
dhequality-alternative d PP (pp (V!i)) m2 m2'
by *auto*
qed

with *Areflassump Up-To-Technique*
have *SdlHPPB d PP (R0 \cup R)*
by *auto*

with *inR2 R0-def* **show** $\exists R. \text{SdlHPPB } d \text{ PP } R \wedge$
 $([\text{while}_{\iota} b \text{ do } c \text{ od}], [\text{while}_{\iota} b \text{ do } c \text{ od}]) \in R$
by *auto*
qed

end

end

4 Security type system

4.1 Abstract security type system with soundness proof

We formalize an abstract version of the type system in [2] using locales [1]. Our formalization of the type system is abstract in the sense that the rules specify abstract semantic side conditions on the expressions within a command that satisfy for proving the soundness of the rules. That is, it can be instantiated with different syntactic approximations for these semantic side conditions in order to achieve a type system for a concrete language for Boolean and arithmetic expressions. Obtaining a soundness proof for such a concrete type system then boils down to proving that the concrete type system interprets the abstract type system.

We prove the soundness of the abstract type system by simply applying the compositionality results proven before.

```

theory Type-System
imports Language-Composition
begin

locale Type-System =
  WWP?: WHATWHERE-Secure-Programs E BMap DA lH
  for E :: ('exp, 'id, 'val) Evalfunction
  and BMap :: 'val  $\Rightarrow$  bool
  and DA :: ('id, 'd::order) DomainAssignment
  and lH :: ('d, 'exp) lHatches
+
fixes
AssignSideCondition :: 'id  $\Rightarrow$  'exp  $\Rightarrow$  nat  $\Rightarrow$  bool
and WhileSideCondition :: 'exp  $\Rightarrow$  bool
and IfSideCondition ::
  'exp  $\Rightarrow$  ('exp, 'id) MWLSCom  $\Rightarrow$  ('exp, 'id) MWLSCom  $\Rightarrow$  bool
assumes semAssignSC: AssignSideCondition x e  $\iota \Longrightarrow$ 
   $e \equiv_{DA} x, (htchLoc \iota) e \wedge (\forall m m' d \iota'. (m \sim_{d, (htchLoc \iota')} m' \wedge$ 
   $\llbracket x :=_{\iota} e \rrbracket(m) =_d \llbracket x :=_{\iota} e \rrbracket(m')$ 
   $\longrightarrow \llbracket x :=_{\iota} e \rrbracket(m) \sim_{d, (htchLoc \iota')} \llbracket x :=_{\iota} e \rrbracket(m'))$ 
and semWhileSC: WhileSideCondition e  $\Longrightarrow \forall d. e \equiv_d e$ 
and semIfSC: IfSideCondition e c1 c2  $\Longrightarrow \forall d. e \equiv_d e$ 
begin

— Security typing rules for the language commands
inductive
ComSecTyping :: ('exp, 'id) MWLSCom  $\Rightarrow$  bool
  ( $\vdash_C$  -)
and ComSecTypingL :: ('exp, 'id) MWLSCom list  $\Rightarrow$  bool
  ( $\vdash_{\mathcal{V}}$  -)
where

```

Skip: $\vdash_{\mathcal{C}} \text{skip}_\iota$ |
Assign: $\llbracket \text{AssignSideCondition } x \ e \ \iota \rrbracket \Longrightarrow \vdash_{\mathcal{C}} x :=_\iota e$ |
Spawn: $\llbracket \vdash_{\mathcal{V}} V \rrbracket \Longrightarrow \vdash_{\mathcal{C}} \text{spawn}_\iota V$ |
Seq: $\llbracket \vdash_{\mathcal{C}} c1; \vdash_{\mathcal{C}} c2 \rrbracket \Longrightarrow \vdash_{\mathcal{C}} c1; c2$ |
While: $\llbracket \vdash_{\mathcal{C}} c; \text{WhileSideCondition } b \rrbracket$
 $\Longrightarrow \vdash_{\mathcal{C}} \text{while}_\iota b \text{ do } c \text{ od}$ |
If: $\llbracket \vdash_{\mathcal{C}} c1; \vdash_{\mathcal{C}} c2; \text{IfSideCondition } b \ c1 \ c2 \rrbracket$
 $\Longrightarrow \vdash_{\mathcal{C}} \text{if}_\iota b \text{ then } c1 \text{ else } c2 \text{ fi}$ |
Parallel: $\llbracket \forall i < \text{length } V. \vdash_{\mathcal{C}} V!i \rrbracket \Longrightarrow \vdash_{\mathcal{V}} V$

inductive-cases *parallel-cases*:

$\vdash_{\mathcal{V}} V$

definition *auxiliary-predicate*

where

auxiliary-predicate $V \equiv \text{unique-PPV } V \longrightarrow \text{WHATWHERE-Secure } V$

— soundness proof of abstract type system

theorem *ComSecTyping-single-is-sound*:

$\llbracket \vdash_{\mathcal{C}} c; \text{unique-PPc } c \rrbracket$
 $\Longrightarrow \text{WHATWHERE-Secure } [c]$

proof (*induct rule: ComSecTyping-ComSecTypingL.inducts(1)*)

[*of - - auxiliary-predicate*], *simp-all add: auxiliary-predicate-def*)

fix ι

show *WHATWHERE-Secure* [*skip* $_\iota$]

by (*metis WHATWHERE-Secure-Skip*)

next

fix $x \ e \ \iota$

assume *AssignSideCondition* $x \ e \ \iota$

thus *WHATWHERE-Secure* [$x :=_\iota e$]

by (*metis WHATWHERE-Secure-Assign semAssignSC*)

next

fix $V \ \iota$

assume *IH*: *unique-PPV* $V \longrightarrow \text{WHATWHERE-Secure } V$

assume *uniPPspawn*: *unique-PPc* (*spawn* $_\iota V$)

hence *unique-PPV* V

by (*simp add: unique-PPV-def unique-PPc-def*)

with *IH* **have** *WHATWHERE-Secure* V

..

with *uniPPspawn* **show** *WHATWHERE-Secure* [*spawn* $_\iota V$]

by (*metis Compositionality-Spawn*)

next

fix $c1 \ c2$

assume *IH1*: *unique-PPc* $c1 \Longrightarrow \text{WHATWHERE-Secure } [c1]$

assume *IH2*: *unique-PPc* $c2 \Longrightarrow \text{WHATWHERE-Secure } [c2]$

assume *uniPPc1c2*: *unique-PPc* ($c1; c2$)

from *uniPPc1c2* **have** *uniPPc1*: *unique-PPc* $c1$

by (*simp add: unique-PPc-def*)

with *IH1* **have** *IS1*: *WHATWHERE-Secure* [$c1$]

```

.
from uniPPc1c2 have uniPPc2: unique-PPc c2
  by (simp add: unique-PPc-def)
with IH2 have IS2: WHATWHERE-Secure [c2]
.

from IS1 IS2 uniPPc1c2 show WHATWHERE-Secure [c1;c2]
  by (metis Compositionality-Seq)
next
fix c b ι
assume SC: WhileSideCondition b
assume IH: unique-PPc c ⇒ WHATWHERE-Secure [c]
assume uniPPwhile: unique-PPc (whileι b do c od)
hence unique-PPc c
  by (simp add: unique-PPc-def)
with IH have WHATWHERE-Secure [c]
.

with uniPPwhile SC show WHATWHERE-Secure [whileι b do c od]
  by (metis Compositionality-While semWhileSC)
next
fix c1 c2 b ι
assume SC: IfSideCondition b c1 c2
assume IH1: unique-PPc c1 ⇒ WHATWHERE-Secure [c1]
assume IH2: unique-PPc c2 ⇒ WHATWHERE-Secure [c2]
assume uniPPif: unique-PPc (ifι b then c1 else c2 fi)
from uniPPif have unique-PPc c1
  by (simp add: unique-PPc-def)
with IH1 have IS1: WHATWHERE-Secure [c1]
.

from uniPPif have unique-PPc c2
  by (simp add: unique-PPc-def)
with IH2 have IS2: WHATWHERE-Secure [c2]
.

from IS1 IS2 SC uniPPif show
  WHATWHERE-Secure [ifι b then c1 else c2 fi]
  by (metis Compositionality-If semIfSC)
next
fix V
assume IH: ∀ i < length V. ⊢C V ! i ∧
  (unique-PPc (V!i) → WHATWHERE-Secure [V!i])
have unique-PPV V → (∀ i < length V. unique-PPc (V!i))
  by (metis uniPPV-uniPPc)
with IH have unique-PPV V → (∀ i < length V. WHATWHERE-Secure [V!i])

  by auto
thus uniPPV: unique-PPV V → WHATWHERE-Secure V
  by (metis parallel-composition)
qed

```

```

theorem ComSecTyping-list-is-sound:
   $\llbracket \vdash_{\mathcal{V}} V; \text{unique-PPV } V \rrbracket \implies \text{WHATWHERE-Secure } V$ 
by (metis ComSecTyping-single-is-sound parallel-cases
      parallel-composition uniPPV-uniPPc)

```

```

end

```

```

end

```

4.2 Example language for Boolean and arithmetic expressions

As an example, we provide a simple example language for instantiating the parameter *'exp* for the language for Boolean and arithmetic expressions (from the entry Strong-Security).

```

theory Expr
imports Types
begin

```

```

— type parameters:
— 'val: numbers, boolean constants....
— 'id: identifier names

```

```

type-synonym ('val) operation = 'val list  $\Rightarrow$  'val

```

```

datatype (dead 'id, dead 'val) Expr =
  Const 'val |
  Var 'id |
  Op 'val operation (( 'id, 'val) Expr) list

```

— defining a simple recursive evaluation function on this datatype

```

primrec ExprEval :: (('id, 'val) Expr, ('id, 'val) Evalfunction
and ExprEvalL :: (('id, 'val) Expr) list  $\Rightarrow$  ('id, 'val) State  $\Rightarrow$  'val list

```

```

where
  ExprEval (Const v) m = v |
  ExprEval (Var x) m = (m x) |
  ExprEval (Op f arglist) m = (f (ExprEvalL arglist m)) |

```

```

  ExprEvalL [] m = [] |
  ExprEvalL (e#V) m = (ExprEval e m)#(ExprEvalL V m)

```

```

end

```

4.3 Example interpretation of abstract security type system

Using the example instantiation of the language for Boolean and arithmetic expressions, we give an example instantiation of our abstract security type system, instantiating the parameter for domains d with a two-level security lattice (from the entry Strong-Security).

```
theory Domain-example
imports Expr
begin
```

— When interpreting, we have to instantiate the type for domains. As an example, we take a type containing 'low' and 'high' as domains.

```
datatype Dom = low | high
```

```
instantiation Dom :: order
begin
```

```
definition
```

```
less-eq-Dom-def:  $d1 \leq d2 = (if\ d1 = d2\ then\ True$   
   $else\ (if\ d1 = low\ then\ True\ else\ False))$ 
```

```
definition
```

```
less-Dom-def:  $d1 < d2 = (if\ d1 = d2\ then\ False$   
   $else\ (if\ d1 = low\ then\ True\ else\ False))$ 
```

```
instance proof
```

```
fix  $x\ y\ z :: Dom$ 
```

```
  show  $(x < y) = (x \leq y \wedge \neg y \leq x)$ 
```

```
    unfolding less-eq-Dom-def less-Dom-def by auto
```

```
  show  $x \leq x$  unfolding less-eq-Dom-def by auto
```

```
  show  $\llbracket x \leq y; y \leq z \rrbracket \implies x \leq z$ 
```

```
    unfolding less-eq-Dom-def by  $((split\ if-split-asm)+, auto)$ 
```

```
  show  $\llbracket x \leq y; y \leq x \rrbracket \implies x = y$ 
```

```
    unfolding less-eq-Dom-def by  $((split\ if-split-asm)+,$   
   $auto, (split\ if-split-asm)+, auto)$ 
```

```
qed
```

```
end
```

```
end
```

```
theory Type-System-example
```

```
imports Type-System Strong-Security.Expr Strong-Security.Domain-example
```

```
begin
```

— When interpreting, we have to instantiate the type for domains. As an example, we take a type containing 'low' and 'high' as domains.

consts $DA :: ('id, Dom) DomainAssignment$
consts $BMap :: 'val \Rightarrow bool$
consts $lH :: (Dom, ('id, 'val) Expr) lHatches$

— redefine all the abbreviations necessary for auxiliary lemmas with the correct parameter instantiation

abbreviation $MWLSStepsdet' ::$
 $(('id, 'val) Expr, 'id, 'val, (('id, 'val) Expr, 'id) MWLsCom) TLSteps-curry$
 $((1 \langle -, - \rangle) \rightarrow \langle \cdot \rangle / (1 \langle -, - \rangle) [0, 0, 0, 0, 0] 81)$
where
 $\langle c1, m1 \rangle \rightarrow \langle \alpha \rangle \langle c2, m2 \rangle \equiv$
 $((c1, m1), \alpha, (c2, m2)) \in MWLs-semantic.MWLSSteps-det ExprEval BMap$

abbreviation $d-equal' :: ('id, 'val) State$
 $\Rightarrow Dom \Rightarrow ('id, 'val) State \Rightarrow bool$
 $((- = -))$
where
 $m \equiv_d m' \equiv WHATWHERE.d-equal DA d m m'$

abbreviation $dH-equal' :: ('id, 'val) State \Rightarrow Dom$
 $\Rightarrow (Dom, ('id, 'val) Expr) Hatches$
 $\Rightarrow ('id, 'val) State \Rightarrow bool$
 $((- \sim_{d,H} -))$
where
 $m \sim_{d,H} m' \equiv WHATWHERE.dH-equal ExprEval DA d H m m'$

abbreviation $NextMem' :: (('id, 'val) Expr, 'id) MWLsCom$
 $\Rightarrow ('id, 'val) State \Rightarrow ('id, 'val) State$
 $([\cdot])(-)$
where
 $[[c]](m)$
 $\equiv WHATWHERE.NextMem (MWLs-semantic.MWLSSteps-det ExprEval BMap)$
 $c m$

abbreviation $dH-indistinguishable' :: ('id, 'val) Expr \Rightarrow Dom$
 $\Rightarrow (Dom, ('id, 'val) Expr) Hatches \Rightarrow ('id, 'val) Expr \Rightarrow bool$
 $((- \equiv_{d,H} -))$
where
 $e1 \equiv_{d,H} e2$
 $\equiv WHATWHERE-Secure-Programs.dH-indistinguishable ExprEval DA d H e1 e2$

abbreviation $htchLoc :: nat \Rightarrow (Dom, ('id, 'val) Expr) Hatches$
where
 $htchLoc \iota \equiv WHATWHERE.htchLoc lH \iota$

— Security typing rules for expressions

inductive

ExprSecTyping :: (*Dom*, ('*id*', '*val*') *Expr*) *Hatches*

⇒ ('*id*', '*val*') *Expr* ⇒ *Dom* ⇒ *bool*

(- ⊢_ℰ - : -)

for *H* :: (*Dom*, ('*id*', '*val*') *Expr*) *Hatches*

where

Consts: $H \vdash_{\mathcal{E}} (\text{Const } v) : d \mid$

Vars: $DA \ x = d \implies H \vdash_{\mathcal{E}} (\text{Var } x) : d \mid$

Hatch: $(d, e) \in H \implies H \vdash_{\mathcal{E}} e : d \mid$

Ops: $\llbracket \forall i < \text{length } \text{arglist}. H \vdash_{\mathcal{E}} (\text{arglist}!i) : (d!!i) \wedge (d!!i) \leq d \rrbracket$
 $\implies H \vdash_{\mathcal{E}} (\text{Op } f \ \text{arglist}) : d$

— function substituting a certain expression with another expression in expressions

primrec *Subst* :: ('*id*', '*val*') *Expr* ⇒ ('*id*', '*val*') *Expr*

⇒ ('*id*', '*val*') *Expr* ⇒ ('*id*', '*val*') *Expr*

(-<->)

and *SubstL* :: ('*id*', '*val*') *Expr list* ⇒ ('*id*', '*val*') *Expr*

⇒ ('*id*', '*val*') *Expr* ⇒ ('*id*', '*val*') *Expr list*

where

$(\text{Const } v) \langle e1 \setminus e2 \rangle = (\text{if } e1 = (\text{Const } v) \text{ then } e2 \text{ else } (\text{Const } v)) \mid$

$(\text{Var } x) \langle e1 \setminus e2 \rangle = (\text{if } e1 = (\text{Var } x) \text{ then } e2 \text{ else } (\text{Var } x)) \mid$

$(\text{Op } f \ \text{arglist}) \langle e1 \setminus e2 \rangle = (\text{if } e1 = (\text{Op } f \ \text{arglist}) \text{ then } e2 \text{ else}$

$(\text{Op } f \ (\text{SubstL } \text{arglist } e1 \ e2))) \mid$

$\text{SubstL } [] \ e1 \ e2 = [] \mid$

$\text{SubstL } (e \# V) \ e1 \ e2 = (e \langle e1 \setminus e2 \rangle) \# (\text{SubstL } V \ e1 \ e2)$

definition *SubstClosure* :: '*id*' ⇒ ('*id*', '*val*') *Expr* ⇒ *bool*

where

$\text{SubstClosure } x \ e \equiv \forall (d', e', \iota') \in \text{lh}. (d', e' \langle (\text{Var } x) \setminus e \rangle, \iota') \in \text{lh}$

definition *synAssignSC* :: '*id*' ⇒ ('*id*', '*val*') *Expr* ⇒ *nat* ⇒ *bool*

where

$\text{synAssignSC } x \ e \ \iota \equiv \exists d. ((\text{htchLoc } \iota) \vdash_{\mathcal{E}} e : d \wedge d \leq DA \ x)$

$\wedge (\text{SubstClosure } x \ e)$

definition *synWhileSC* :: ('*id*', '*val*') *Expr* ⇒ *bool*

where

$\text{synWhileSC } e \equiv (\exists d. (\{\} \vdash_{\mathcal{E}} e : d) \wedge (\forall d'. d \leq d'))$

definition *synIfSC* :: ('*id*', '*val*') *Expr*

⇒ ('*id*', '*val*') *Expr*, '*id*') *MWLSCom*

⇒ ('*id*', '*val*') *Expr*, '*id*') *MWLSCom* ⇒ *bool*

where

$\text{synIfSC } e \ c1 \ c2 \equiv \exists d. (\{\} \vdash_{\mathcal{E}} e : d \wedge (\forall d'. d \leq d'))$

— auxiliary lemma for locale interpretation (theorem 7 in original paper)

lemma *ExprTypable-with-smallerd-implies-dH-indistinguishable*:

$\llbracket H \vdash_{\mathcal{E}} e : d'; d' \leq d \rrbracket \implies e \equiv_{d,H} e$

proof (*induct rule*: *ExprSecTyping.induct*,

simp-all add: *WHATWHERE-Secure-Programs.dH-indistinguishable-def*

WHATWHERE.dH-equal-def *WHATWHERE.d-equal-def*, *auto*)

fix *dl arglist f m1 m2 d' d*

assume *main*: $\forall i < \text{length } \text{arglist}.$

$(H \vdash_{\mathcal{E}} (\text{arglist}!i) : (dl!i)) \wedge (dl!i \leq d \longrightarrow$

$(\forall m1\ m2. (\forall x. DA\ x \leq d \longrightarrow m1\ x = m2\ x) \wedge$

$(\forall (d',e) \in H. d' \leq d \longrightarrow \text{ExprEval } e\ m1 = \text{ExprEval } e\ m2) \longrightarrow$

$\text{ExprEval } (\text{arglist}!i)\ m1 = \text{ExprEval } (\text{arglist}!i)\ m2)) \wedge dl!i \leq d'$

assume *smaller*: $d' \leq d$

assume *eqeval*: $\forall (d',e) \in H. d' \leq d \longrightarrow \text{ExprEval } e\ m1 = \text{ExprEval } e\ m2$

assume *eqstate*: $\forall x. DA\ x \leq d \longrightarrow m1\ x = m2\ x$

from *main* *smaller* **have** *irangesubst*:

$\forall i < \text{length } \text{arglist}. dl!i \leq d$

by (*metis order-trans*)

with *eqstate* *eqeval* *main* **have**

$\forall i < \text{length } \text{arglist}. \text{ExprEval } (\text{arglist}!i)\ m1$

$= \text{ExprEval } (\text{arglist}!i)\ m2$

by *force*

hence *substmap*: $(\text{ExprEvalL } \text{arglist } m1) = (\text{ExprEvalL } \text{arglist } m2)$

by (*induct arglist*, *auto*, *force*)

show $f (\text{ExprEvalL } \text{arglist } m1) = f (\text{ExprEvalL } \text{arglist } m2)$

by (*subst substmap*, *auto*)

qed

— auxiliary lemma about substitutions in expressions and in memories

lemma *substexp-substmem*:

$\text{ExprEval } e' \langle \text{Var } x \setminus e \rangle m = \text{ExprEval } e' (m(x := \text{ExprEval } e\ m))$

$\wedge \text{ExprEvalL } (\text{SubstL } \text{elist } (\text{Var } x) e) m$

$= \text{ExprEvalL } \text{elist } (m(x := \text{ExprEval } e\ m))$

by (*induct-tac e'* **and** *elist rule*: *ExprEval.induct* *ExprEvalL.induct*, *simp-all*)

— another auxiliary lemma for locale interpretation (lemma 8 in original paper)

lemma *SubstClosure-implications*:

$\llbracket \text{SubstClosure } x\ e; m \sim_{d,(\text{htchLoc } \iota')} m' \rrbracket;$

$\llbracket x :=_{\iota} e \rrbracket(m) =_d \llbracket x :=_{\iota} e \rrbracket(m') \rrbracket$

$\implies \llbracket x :=_{\iota} e \rrbracket(m) \sim_{d,(\text{htchLoc } \iota')} \llbracket x :=_{\iota} e \rrbracket(m') \rrbracket$

proof —

fix *m1 m1'*

assume *substclosure*: *SubstClosure* *x e*

assume *dequalm2*: $\llbracket x :=_{\iota} e \rrbracket(m1) =_d \llbracket x :=_{\iota} e \rrbracket(m1') \rrbracket$

assume $dhequalm1: m1 \sim_{d, (htchLoc \iota')} m1'$

from $MWLS\text{-}semantics.nextmem\text{-}exists\text{-}and\text{-}unique$ **obtain** $m2$ **where** $m1step$:

$(\exists p \alpha. \langle x :=_{\iota} e, m1 \rangle \rightarrow \triangleleft \alpha \triangleright \langle p, m2 \rangle)$
 $\wedge (\forall m''. (\exists p \alpha. \langle x :=_{\iota} e, m1 \rangle \rightarrow \triangleleft \alpha \triangleright \langle p, m'' \rangle) \longrightarrow m'' = m2)$
by force

hence $m2\text{-}is\text{-}next: \llbracket x :=_{\iota} e \rrbracket(m1) = m2$

by ($simp$ $add: WHATWHERE.NextMem\text{-}def, auto$)

from $m1step$ $MWLS\text{-}semantics.MWLSSteps\text{-}det.assign$

[$of ExprEval e m1 - x \iota BMap$]

have $m2eq: m2 = m1(x := (ExprEval e m1))$

by auto

from $MWLS\text{-}semantics.nextmem\text{-}exists\text{-}and\text{-}unique$ **obtain** $m2'$ **where** $m1'\text{step}$:

$(\exists p \alpha. \langle x :=_{\iota} e, m1' \rangle \rightarrow \triangleleft \alpha \triangleright \langle p, m2' \rangle)$
 $\wedge (\forall m''. (\exists p \alpha. \langle x :=_{\iota} e, m1' \rangle \rightarrow \triangleleft \alpha \triangleright \langle p, m'' \rangle) \longrightarrow m'' = m2')$
by force

hence $m2'\text{-}is\text{-}next: \llbracket x :=_{\iota} e \rrbracket(m1') = m2'$

by ($simp$ $add: WHATWHERE.NextMem\text{-}def, auto$)

from $m1'\text{step}$ $MWLS\text{-}semantics.MWLSSteps\text{-}det.assign$

[$of ExprEval e m1' - x \iota BMap$]

have $m2'\text{eq}: m2' = m1'(x := (ExprEval e m1'))$

by auto

from $m2eq$ $substexp\text{-}substmem$

have $substeval1: \forall e'. ExprEval (e' < Var x \setminus e >) m1 = ExprEval e' m2$

by force

from $m2'\text{eq}$ $substexp\text{-}substmem$

have $substeval2: \forall e'. ExprEval e' < Var x \setminus e > m1' = ExprEval e' m2'$

by force

from $substclosure$ **have**

$\forall (d', e') \in htchLoc \iota'. (d', e' < Var x \setminus e >) \in (htchLoc \iota')$

by ($simp$ $add: SubstClosure\text{-}def$ $WHATWHERE.htchLoc\text{-}def, auto$)

with $dhequalm1$ **have**

$\forall (d', e') \in htchLoc \iota'.$

$d' \leq d \longrightarrow ExprEval e' < Var x \setminus e > m1 = ExprEval e' < Var x \setminus e > m1'$

by ($simp$ $add: WHATWHERE.dH\text{-}equal\text{-}def, auto$)

with $substeval1$ $substeval2$ **have**

$\forall (d', e') \in htchLoc \iota'.$

$d' \leq d \longrightarrow ExprEval e' m2 = ExprEval e' m2'$

by auto

with $dequalm2$ $m2\text{-}is\text{-}next$ $m2'\text{-}is\text{-}next$

show $\llbracket x :=_{\iota} e \rrbracket(m1) \sim_{d, htchLoc \iota'} \llbracket x :=_{\iota} e \rrbracket(m1')$

by ($simp$ $add: WHATWHERE.dH\text{-}equal\text{-}def$)

qed

— interpretation of the abstract type system using the above definitions for the side conditions

interpretation *Type-System-example: Type-System ExprEval BMap DA lH*
synAssignSC synWhileSC synIfSC

by (*unfold-locales, auto,*
metis ExprTypable-with-smallerd-implies-dH-indistinguishable
synAssignSC-def,
metis SubstClosure-implications synAssignSC-def,
simp add: synWhileSC-def,
metis ExprTypable-with-smallerd-implies-dH-indistinguishable
WHATWHERE-Secure-Programs.empH-implies-dHindistinguishable-eq-dindistinguishable,

simp add: synIfSC-def,
metis ExprTypable-with-smallerd-implies-dH-indistinguishable
WHATWHERE-Secure-Programs.empH-implies-dHindistinguishable-eq-dindistinguishable)

end

References

- [1] C. Ballarin. Locales and Locale Expressions in Isabelle/Isar. In S. Berardi, M. Coppo, and F. Damiani, editors, *TYPES*, volume 3085 of *Lecture Notes in Computer Science*, pages 34–50. Springer, 2003.
- [2] A. Lux, H. Mantel, and M. Perner. Scheduler-independent declassification. In *Proceedings of the 11th International Conference on Mathematics of Program Construction (MPC)*, LNCS 7342, pages 25–47. Springer, 2012.