An Isabelle/HOL formalization of Strong Security

Sylvia Grewe, Alexander Lux, Heiko Mantel, Jens Sauer

March 17, 2025

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. Strong security from [2] formalizes noninterference for concurrent systems.

We present an Isabelle/HOL formalization of strong security for arbitrary security lattices ([2] uses a two-element security lattice). The formalization includes compositionality proofs for strong security and a soundness proof for a security type system that checks strong security 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.

Contents

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:

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* $=$
$('com\ list) \times ('id, 'val)$ *State*

— type for program states (including the set of commands and a symbol for terminating - None)
type-synonym $'com$ *ProgramState* $= 'com\ option$

— type for configurations with program states
type-synonym $('id, 'val, 'com)$ *PSConfig* =
$'com$ *ProgramState* $\times$ $('id, 'val)$ *State*

— type for labels with a list of spawned threads
type-synonym $'com$ *Label* = $'com$ *list*

— type for step relations from single commands to a program state, with a label
type-synonym $('exp, 'id, 'val, 'com)$ *TLSteps* =
$(('id, 'val, 'com)$ *TConfig* $\times$ $'com$ *Label*
$\times$ $('id, 'val, 'com)$ *PSConfig*$)$ *set*

— curried version of previously defined type
type-synonym $('exp, 'id, 'val, 'com)$ *TLSteps-curry* =
$'com \Rightarrow ('id, 'val)$ *State* $\Rightarrow$ $'com$ *Label* $\Rightarrow$ $'com$ *ProgramState*
$\Rightarrow ('id, 'val)$ *State* $\Rightarrow$ *bool*

— type for step relations from thread pools to thread pools
type-synonym $('exp, 'id, 'val, 'com)$ *TPSteps* =
$(('id, 'val, 'com)$ *TPConfig* $\times$ $('id, 'val, 'com)$ *TPConfig*$)$ *set*

— curried version of previously defined type
type-synonym $('exp, 'id, 'val, 'com)$ *TPSteps-curry* =
$'com$ *list* $\Rightarrow ('id, 'val)$ *State* $\Rightarrow$ $'com$ *list* $\Rightarrow ('id, 'val)$ *State* $\Rightarrow$ *bool*

— define type of step relations for single threads to thread pools
type-synonym $('exp, 'id, 'val, 'com)$ *TSteps* =
$(('id, 'val, 'com)$ *TConfig* $\times$ $('id, 'val, 'com)$ *TPConfig*$)$ *set*

— define the same type as TSteps, but in a curried version (allowing syntax abbreviations)
type-synonym $('exp, 'id, 'val, 'com)$ *TSteps-curry* =
$'com \Rightarrow ('id, 'val)$ *State* $\Rightarrow$ $'com$ *list* $\Rightarrow ('id, 'val)$ *State* $\Rightarrow$ *bool*

— type for simple domain assignments; 'd has to be an instance of order (partial order
type-synonym $('id, 'd)$ *DomainAssignment* = $'id \Rightarrow 'd$::*order*

type-synonym $'com$ *Bisimulation-type* = $(('com$ *list*$)$ $\times$ $('com$ *list*$))$ *set*

— type for escape hatches
type-synonym $('d, 'exp)$ *Hatch* = $'d \times 'exp$

— 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* = $'d \times 'exp \times$ *nat*

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

end

2 Strong security

2.1 Definition of strong security

We define strong security such that it is parametric in a security lattice (*'d*). The definition of strong security by itself is language-independent, therefore the definition is parametric in a programming language (*'com*) in addition.

theory *Strong-Security*
imports *Types*
begin

locale *Strong-Security* =
fixes *SR* :: $('exp, 'id, 'val, 'com)$ *TSteps*
and *DA* :: $('id, 'd::order)$ *DomainAssignment*

begin

— define when two states are indistinguishable for an observer on domain d
definition *d-equal* :: $'d::order \Rightarrow ('id, 'val)$ *State*
$\Rightarrow ('id, 'val)$ *State* $\Rightarrow$ *bool*
where
d-equal d m m' $\equiv \forall x. ((DA\ x) \leq d \longrightarrow (m\ x) = (m'\ x))$

abbreviation *d-equal'* :: $('id, 'val)$ *State*
$\Rightarrow 'd::order \Rightarrow ('id, 'val)$ *State* $\Rightarrow$ *bool*
(‹(- =- -)›)
where
$m =_d m' \equiv$ *d-equal d m m'*

— transitivity of d-equality
lemma *d-equal-trans*:
$[\![m =_d m'; m' =_d m'']\!] \Longrightarrow m =_d m''$
by (*simp add*: *d-equal-def*)

abbreviation *SRabbr* :: $('exp, 'id, 'val, 'com)$ *TSteps-curry*
(‹(1⟨-,/-⟩) →/ (1⟨-,/-⟩)› [0,0,0,0] 81)
where
$\langle c,m \rangle \rightarrow \langle c',m' \rangle \equiv ((c,m),(c',m')) \in SR$

— predicate for strong d-bisimulation

definition *Strong-d-Bisimulation* :: $'d \Rightarrow 'com\ Bisimulation\text{-}type \Rightarrow bool$
where
Strong-d-Bisimulation d R $\equiv$
$(sym\ R) \wedge$
$(\forall (V,V') \in R.\ length\ V = length\ V') \wedge$
$(\forall (V,V') \in R.\ \forall i < length\ V.\ \forall m1\ m1'\ m2\ W.$
$\langle V!i,m1\rangle \rightarrow \langle W,m2\rangle \wedge m1 =_d m1'$
$\longrightarrow (\exists W'\ m2'.\ \langle V'!i,m1'\rangle \rightarrow \langle W',m2'\rangle \wedge (W,W') \in R \wedge m2 =_d m2'))$

— union of all strong d-bisimulations
definition *USdB* :: $'d \Rightarrow 'com\ Bisimulation\text{-}type$
(‹≈_› *65*)
where
$\approx_d \equiv \bigcup\{r.\ (Strong\text{-}d\text{-}Bisimulation\ d\ r)\}$

abbreviation *relatedbyUSdB* :: $'com\ list \Rightarrow 'd \Rightarrow 'com\ list \Rightarrow bool$
(‹(- ≈_ -)› [*66*,*66*] *65*)
where $V \approx_d V' \equiv (V,V') \in USdB\ d$

— predicate to define when a program is strongly secure
definition *Strongly-Secure* :: $'com\ list \Rightarrow bool$
where
Strongly-Secure $V \equiv (\forall d.\ V \approx_d V)$

— auxiliary lemma to obtain central strong d-Bisimulation property as Lemma in meta logic (allows instantiating all the variables manually if necessary)
lemma *strongdB-aux*: $\bigwedge V\ V'\ m1\ m1'\ m2\ W\ i.\ [\![$ *Strong-d-Bisimulation d R*;
$i < length\ V$; $(V,V') \in R$; $\langle V!i,m1\rangle \rightarrow \langle W,m2\rangle$; $m1 =_d m1'$ $]\!]$
$\Longrightarrow (\exists W'\ m2'.\ \langle V'!i,m1'\rangle \rightarrow \langle W',m2'\rangle \wedge (W,W') \in R \wedge m2 =_d m2')$
by (*simp add*: *Strong-d-Bisimulation-def*, *fastforce*)

lemma *trivialpair-in-USdB*:
$[] \approx_d []$
by (*simp add*: *USdB-def Strong-d-Bisimulation-def*,
rule-tac x={([],[])} **in** *exI*, *simp add*: *sym-def*)

lemma *USdBsym*: *sym* $(\approx_d)$
by (*simp add*: *USdB-def Strong-d-Bisimulation-def sym-def*, *auto*)

lemma *USdBeqlen*:
$V \approx_d V' \Longrightarrow length\ V = length\ V'$
by (*simp add*: *USdB-def Strong-d-Bisimulation-def*, *auto*)

lemma *USdB-Strong-d-Bisimulation*:
Strong-d-Bisimulation d $(\approx_d)$
proof (*simp add*: *Strong-d-Bisimulation-def*, *auto*)
show *sym* $(\approx_d)$ **by** (*rule USdBsym*)
next

fix $V\ V'$
show $V \approx_d V' \Longrightarrow$ *length* $V =$ *length* V' **by** (*rule USdBeqlen*, *auto*)
next
fix $V\ V'$ *m1 m1' m2* $W\ i$
assume *inUSdB*: $V \approx_d V'$
assume *stepV*: $\langle V!i, m1\rangle \rightarrow \langle W, m2\rangle$
assume *irange*: $i <$ *length* V
assume *dequal*: $m1 =_d m1'$

from *inUSdB* **obtain** R **where** *someR*:
Strong-d-Bisimulation $d\ R \wedge (V, V') \in R$
by (*simp add*: *USdB-def*, *auto*)

with *strongdB-aux stepV irange dequal* **show**
$\exists\, W'\ m2'.\ \langle V'!i, m1'\rangle \rightarrow \langle W', m2'\rangle \wedge W \approx_d W' \wedge m2 =_d m2'$
by (*simp add*: *USdB-def*, *fastforce*)

qed

lemma *USdBtrans*: *trans* $(\approx_d)$
proof (*simp add*: *trans-def*, *auto*)
fix $V\ V'\ V''$
assume *p1*: $V \approx_d V'$
assume *p2*: $V' \approx_d V''$

let $?R = \{(V, V'').\ \exists\, V'.\ V \approx_d V' \wedge V' \approx_d V''\}$

from *p1 p2* **have** *inRest*: $(V, V'') \in\ ?R$ **by** *auto*

have *SdB-rest*: *Strong-d-Bisimulation* $d\ ?R$
proof (*simp add*: *Strong-d-Bisimulation-def sym-def*, *auto*)
fix $V\ V'\ V''$
assume *p1*: $V \approx_d V'$
moreover
assume *p2*: $V' \approx_d V''$
moreover
from *p1 USdBsym* **have** $V' \approx_d V$
by (*simp add*: *sym-def*)
moreover
from *p2 USdBsym* **have** $V'' \approx_d V'$
by (*simp add*: *sym-def*)
ultimately show
$\exists\, V'.\ V'' \approx_d V' \wedge V' \approx_d V$
by (*rule-tac* $x=V'$ **in** *exI*, *auto*)
next
fix $V\ V'\ V''$
assume *p1*: $V \approx_d V'$
moreover

assume *p2*: $V' \approx_d V''$
moreover
from *p1 USdBeqlen*[*of V V'*] **have** *length V = length V'*
  by *auto*
moreover
from *p2 USdBeqlen*[*of V' V''*] **have** *length V' = length V''*
  by *auto*
ultimately show *eqlen*: *length V = length V''* **by** *auto*
next
fix *V V' V'' i m1 m1' W m2*
assume *step*: $\langle V!i,m1\rangle \rightarrow \langle W,m2\rangle$
assume *dequal*: $m1 =_d m1'$
assume *p1*: $V \approx_d V'$
assume *p2*: $V' \approx_d V''$
assume *irange*: $i <$ *length V*
from *p1 USdBeqlen*[*of V V'*]
have *leq*: *length V = length V'*
  by *force*

have *deq-same*: $m1' =_d m1'$ **by** (*simp add*: *d-equal-def*)

from *irange step dequal p1 USdB-Strong-d-Bisimulation*
  strongdB-aux[*of d* $\approx_d$ *i V V' m1 W m2 m1'*]
obtain *W' m2'* **where** *p1concl*:
  $\langle V'!i,m1'\rangle \rightarrow \langle W',m2'\rangle \wedge W \approx_d W' \wedge m2 =_d m2'$
  by *auto*

with *deq-same leq USdB-Strong-d-Bisimulation*
  strongdB-aux[*of d* $\approx_d$ *i V' V'' m1' W' m2' m1'*]
  irange p2 dequal **obtain** *W'' m2''* **where** *p2concl*:
  $W' \approx_d W'' \wedge \langle V''!i,m1'\rangle \rightarrow \langle W'',m2''\rangle \wedge m2' =_d m2''$
  by *auto*

from *p1concl p2concl d-equal-trans* **have** *tt''*: $m2 =_d m2''$
  by *blast*

from *p1concl p2concl* **have** $(W,W'') \in$ *?R*
  by *auto*

with *p2concl tt''* **show** $\exists W''\ m2''.\ \langle V''!i,m1'\rangle \rightarrow \langle W'',m2''\rangle \wedge$
  $(\exists V'.\ W \approx_d V' \wedge V' \approx_d W'') \wedge m2 =_d m2''$
  by *auto*
qed

hence *liftup*: *?R* $\subseteq (\approx_d)$
  by (*simp add*: *USdB-def*, *auto*)

with *inRest* **show** $V \approx_d V''$
  by *auto*

qed

end

end

2.2 Proof technique for compositionality results

For proving compositionality results for strong security, we formalize the following "up-to technique" and prove it sound:

theory *Up-To-Technique*
imports *Strong-Security*
begin

context *Strong-Security*
begin

— define d-bisimulation 'up to' union of strong d-Bisimulations
definition *d-Bisimulation-Up-To-USdB* ::
$'d \Rightarrow 'com$ *Bisimulation-type* $\Rightarrow$ *bool*
where
d-Bisimulation-Up-To-USdB d R $\equiv$
$(sym\ R) \wedge (\forall (V,V') \in R.\ length\ V = length\ V') \wedge$
$(\forall (V,V') \in R.\ \forall i < length\ V.\ \forall m1\ m1'\ W\ m2.$
$\langle V!i,m1\rangle \rightarrow \langle W,m2\rangle \wedge (m1 =_d m1')$
$\longrightarrow (\exists W'\ m2'.\ \langle V'!i,m1'\rangle \rightarrow \langle W',m2'\rangle$
$\wedge (W,W') \in (R \cup (\approx_d)) \wedge (m2 =_d m2')))$

lemma *UpTo-aux*: $\bigwedge V\ V'\ m1\ m1'\ m2\ W\ i.\ [\![$ *d-Bisimulation-Up-To-USdB d R*;
$i < length\ V;\ (V,V') \in R;\ \langle V!i,m1\rangle \rightarrow \langle W,m2\rangle;\ m1 =_d m1'\ ]\!]$
$\Longrightarrow (\exists W'\ m2'.\ \langle V'!i,m1'\rangle \rightarrow \langle W',m2'\rangle$
$\wedge (W,W') \in (R \cup (\approx_d)) \wedge (m2 =_d m2'))$
by (*simp add*: *d-Bisimulation-Up-To-USdB-def*, *fastforce*)

lemma *RuUSdBeqlen*:
$[\![$ *d-Bisimulation-Up-To-USdB d R*;
$(V,V') \in (R \cup (\approx_d))\ ]\!]$
$\Longrightarrow length\ V = length\ V'$
by (*auto*, *simp add*: *d-Bisimulation-Up-To-USdB-def*, *auto*,
rule USdBeqlen, *auto*)

lemma *Up-To-Technique*:
assumes *upToR*: *d-Bisimulation-Up-To-USdB d R*
shows $R \subseteq \approx_d$
proof –
define S **where** $S = R \cup (\approx_d)$

```
from S-def have R ⊆ S
  by auto
moreover
have S ⊆ (≈d)
proof (simp add: USdB-def, auto, rule-tac x=S in exI, auto,
    simp add: Strong-d-Bisimulation-def, auto)
    — show symmetry
  show symS: sym S
  proof −
    from upToR
    have Rsym: sym R
      by (simp add: d-Bisimulation-Up-To-USdB-def)
    with USdBsym have Usym:
      sym (R ∪ (≈d))
      by (metis sym-Un)
    with S-def show ?thesis
      by simp
  qed
next
  fix V V'
  assume inS: (V,V') ∈ S
    — show equal length (by definition)
  from inS S-def upToR RuUSdBeqlen
  show eqlen: length V = length V'
    by simp
next
    — show general bisimulation property
  fix V V' W m1 m1' m2 i
  assume inS: (V,V') ∈ S
  assume irange: i < length V
  assume stepV: ⟨V!i,m1⟩ → ⟨W,m2⟩
  assume dequal: m1 =d m1'

  from inS show ∃ W' m2'. ⟨V'!i,m1'⟩ → ⟨W',m2'⟩ ∧
    (W,W') ∈ S ∧ m2 =d m2'
  proof (simp add: S-def, auto)
    assume firstcase: (V,V') ∈ R

    with upToR dequal irange stepV
      UpTo-aux[of d R i V V' m1 W m2 m1']
    show ∃ W' m2'. ⟨V'!i,m1'⟩ → ⟨W',m2'⟩ ∧
      ((W,W') ∈ R ∨ W ≈d W') ∧ m2 =d m2'
      by (auto simp add: S-def)
  next
    assume secondcase: V ≈d V'

    from USdB-Strong-d-Bisimulation upToR
      secondcase dequal irange stepV
      strongdB-aux[of d ≈d i V V' m1 W m2 m1']
```

show $\exists$ *W' m2'*. ⟨*V'*!*i*,*m1'*⟩ → ⟨*W'*,*m2'*⟩ ∧
((*W*,*W'*) ∈ *R* ∨ *W* $\approx_d$ *W'*) ∧ *m2* $=_d$ *m2'*
by *auto*
qed
qed

ultimately show *?thesis* **by** *auto*
qed

end

end

2.3 Proof of parallel compositionality

We prove that strong security is preserved under composition of strongly secure threads.

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

context *Strong-Security*
begin

theorem *parallel-composition*:
assumes *eqlen*: *length V* = *length V'*
assumes *partsrelated*: ∀ *i* < *length V*. [*V*!*i*] $\approx_d$ [*V'*!*i*]
shows *V* $\approx_d$ *V'*
proof −
define *R* **where** *R* = {(*V*,*V'*). *length V* = *length V'*
∧ (∀ *i* < *length V*. [*V*!*i*] $\approx_d$ [*V'*!*i*])}
from *eqlen partsrelated* **have** *inR*: (*V*,*V'*) ∈ *R*
by (*simp add*: *R-def*)

have *d-Bisimulation-Up-To-USdB d R*
proof (*simp add*: *d-Bisimulation-Up-To-USdB-def*, *auto*)
from *USdBsym* **show** *sym R*
by (*simp add*: *R-def sym-def*)
next
fix *V V'*
assume (*V*,*V'*) ∈ *R*
with *USdBeqlen* **show** *length V* = *length V'*
by (*simp add*: *R-def*)
next
fix *V V' i m1 m1' RS m2*
assume *inR*: (*V*,*V'*) ∈ *R*
assume *irange*: *i* < *length V*
assume *step*: ⟨*V*!*i*,*m1*⟩ → ⟨*RS*,*m2*⟩

assume *dequal*: $m1 =_d m1'$

from *inR* **have** *Vassump*:
$length\ V = length\ V' \wedge (\forall i < length\ V.\ [V!i] \approx_d [V'!i])$
by (*simp add*: *R-def*)

with *step dequal USdB-Strong-d-Bisimulation irange*
strongdB-aux[*of* $d \approx_d 0\ [V!i]\ [V'!i]\ m1\ RS\ m2\ m1'$]
show $\exists RS'\ m2'.\ \langle V'!i,m1'\rangle \rightarrow \langle RS',m2'\rangle \wedge$
$((RS,RS') \in R \vee RS \approx_d RS') \wedge m2 =_d m2'$
by (*simp*, *fastforce*)
qed

hence $R \subseteq (\approx_d)$
by (*rule Up-To-Technique*)

with *inR* **show** *?thesis* **by** *auto*
qed

lemma *parallel-decomposition*:
assumes *related*: $V \approx_d V'$
shows $\forall i < length\ V.\ [V!i] \approx_d [V'!i]$
proof −
define *R* **where** $R = \{(C,C').\ \exists i\ W\ W'.\ W \approx_d W' \wedge i < length\ W$
$\wedge C = [W!i] \wedge C' = [W'!i]\}$

with *related* **have** *inR*: $\forall i < length\ V.\ ([V!i],[V'!i]) \in R$
by *auto*
have *d-Bisimulation-Up-To-USdB d R*
proof (*simp add*: *d-Bisimulation-Up-To-USdB-def*, *auto*)
from *USdBsym USdBeqlen* **show** *sym R*
by (*simp add*: *sym-def R-def*, *metis*)
next
fix $C\ C'$
assume $(C,C') \in R$
with *USdBeqlen* **show** $length\ C = length\ C'$
by (*simp add*: *R-def*, *auto*)
next
fix $C\ C'\ i\ m1\ m1'\ RS\ m2$
assume *inR*: $(C,C') \in R$
assume *irange*: $i < length\ C$
assume *step*: $\langle C!i,m1\rangle \rightarrow \langle RS,m2\rangle$
assume *dequal*: $m1 =_d m1'$

from *inR* **obtain** $j\ W\ W'$ **where** *Rassump*:
$W \approx_d W' \wedge j < length\ W \wedge C = [W!j] \wedge C' = [W'!j]$
by (*simp add*: *R-def*, *auto*)

with *irange* **have** *i0*: $i = 0$ **by** *auto*

from *Rassump i0 strongdB-aux*[*of d* $\approx_d$ *j W W'* *m1 RS m2 m1'*] *USdB-Strong-d-Bisimulation step dequal*
show $\exists\, RS'\ m2'.\ \langle C'!i,m1' \rangle \rightarrow \langle RS',m2' \rangle \land ((RS,RS') \in R \lor RS \approx_d RS') \land m2 =_d m2'$
by *auto*
qed

hence $R \subseteq (\approx_d)$
by (*rule Up-To-Technique*)

with *inR* **show** *?thesis*
by *auto*

qed

lemma *USdB-comp-head-tail*:
assumes *relatedhead*: $[c] \approx_d [c']$
assumes *relatedtail*: $V \approx_d V'$
shows $(c\#V) \approx_d (c'\#V')$
proof −
from *relatedtail USdBeqlen* **have** *eqlen*: $length\ (c\#V) = length\ (c'\#V')$
by *force*

from *relatedtail parallel-decomposition* **have** *singleV*:
$\forall\, i < length\ V.\ [V!i] \approx_d [V'!i]$
by *force*

with *relatedhead* **have** *intermediate*:
$\forall\, i < length\ (c\#V).\ [(c\#V)!i] \approx_d [(c'\#V')!i]$
by (*auto*, *case-tac i*, *auto*)

with *eqlen parallel-composition*
show *?thesis*
by *blast*
qed

lemma *USdB-decomp-head-tail*:
assumes *relatedlist*: $(c\#V) \approx_d (c'\#V')$
shows $[c] \approx_d [c'] \land V \approx_d V'$
proof *auto*
from *relatedlist USdBeqlen*[*of* $c\#V\ c'\#V'$]
have *eqlen*: $length\ V = length\ V'$
by *auto*

from *relatedlist parallel-decomposition*[*of* $c\#V\ c'\#V'\ d$]

```
  have intermediate:
    ∀ i < length (c#V). [(c#V)!i] ≈d [(c'#V')!i]
    by auto
  thus [c] ≈d [c']
    by force

  from intermediate eqlen show V ≈d V'
  proof (case-tac V)
    assume Vcase1: V = []
    with eqlen have V' = [] by auto
    with Vcase1 trivialpair-in-USdB show V ≈d V'
      by auto
  next
    fix c1 W
    assume Vcase2: V = c1#W
    hence Vlen: length V > 0 by auto

    from intermediate have intermediate-aux:
      ⋀i. i < length V
      ⟹ [V!i] ≈d [V'!i]
      by force

    with parallel-composition[of V V'] eqlen
    show V ≈d V'
      by blast

  qed
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 fork command (Multi-threaded While Language with fork, MWLf). Note that the language is still parametric in the language used for Boolean and arithmetic expressions (*'exp*).

```
theory MWLf
imports Types
begin
```

— SYNTAX

— Commands for the multi-threaded while language with fork (to instantiate 'com)
datatype (*'exp*, *'id*) *MWLfCom*
= *Skip* (‹*skip*›)
| *Assign 'id 'exp*
(‹-:=-› [*70*,*70*] *70*)

| *Seq* (*'exp*, *'id*) *MWLfCom* (*'exp*, *'id*) *MWLfCom*
(‹-;-› [*61*,*60*] *60*)

| *If-Else 'exp* (*'exp*, *'id*) *MWLfCom* (*'exp*, *'id*) *MWLfCom*
(‹*if - then - else - fi*› [*80*,*79*,*79*] *70*)

| *While-Do 'exp* (*'exp*, *'id*) *MWLfCom*
(‹*while - do - od*› [*80*,*79*] *70*)

| *Fork* (*'exp*, *'id*) *MWLfCom* ((*'exp*, *'id*) *MWLfCom*) *list*
(‹*fork - -*› [*70*,*70*] *70*)

— SEMANTICS

locale *MWLf-semantics* =
fixes *E* :: (*'exp*, *'id*, *'val*) *Evalfunction*
and *BMap* :: *'val* ⇒ *bool*
begin

— steps semantics, set of deterministic steps from single threads to either single threads or thread pools
inductive-set
MWLfSteps-det :: (*'exp*, *'id*, *'val*, (*'exp*, *'id*) *MWLfCom*) *TSteps*
and *MWLfSteps-det'* :: (*'exp*, *'id*, *'val*, (*'exp*, *'id*) *MWLfCom*) *TSteps-curry*
(‹(*1*⟨-,/-⟩) →/ (*1*⟨-,/-⟩)› [*0*,*0*,*0*,*0*] *81*)
where
⟨*c1*,*m1*⟩ → ⟨*c2*,*m2*⟩ ≡ ((*c1*,*m1*),(*c2*,*m2*)) ∈ *MWLfSteps-det* |
skip: ⟨*skip*,*m*⟩ → ⟨[],*m*⟩ |
assign: (*E e m*) = *v* ⟹ ⟨*x* := *e*,*m*⟩ → ⟨[],*m*(*x* := *v*)⟩ |
seq1: ⟨*c1*,*m*⟩ → ⟨[],*m'*⟩ ⟹ ⟨*c1*;*c2*,*m*⟩ → ⟨[*c2*],*m'*⟩ |
seq2: ⟨*c1*,*m*⟩ → ⟨*c1'*#*V*,*m'*⟩ ⟹ ⟨*c1*;*c2*,*m*⟩ → ⟨(*c1'*;*c2*)#*V*,*m'*⟩ |
iftrue: *BMap* (*E b m*) = *True* ⟹
⟨*if b then c1 else c2 fi*,*m*⟩ → ⟨[*c1*],*m*⟩ |
iffalse: *BMap* (*E b m*) = *False* ⟹
⟨*if b then c1 else c2 fi*,*m*⟩ → ⟨[*c2*],*m*⟩ |
whiletrue: *BMap* (*E b m*) = *True* ⟹
⟨*while b do c od*,*m*⟩ → ⟨[*c*;(*while b do c od*)],*m*⟩ |
whilefalse: *BMap* (*E b m*) = *False* ⟹
⟨*while b do c od*,*m*⟩ → ⟨[],*m*⟩ |
fork: ⟨*fork c V*,*m*⟩ → ⟨*c*#*V*,*m*⟩

inductive-cases *MWLfSteps-det-cases*:
⟨*skip*,*m*⟩ → ⟨*W*,*m'*⟩
⟨*x* := *e*,*m*⟩ → ⟨*W*,*m'*⟩
⟨*c1*;*c2*,*m*⟩ → ⟨*W*,*m'*⟩
⟨*if b then c1 else c2 fi*,*m*⟩ → ⟨*W*,*m'*⟩
⟨*while b do c od*,*m*⟩ → ⟨*W*,*m'*⟩
⟨*fork c V*,*m*⟩ → ⟨*W*,*m'*⟩

— non-deterministic, possibilistic system step (added for intuition, not used in the proofs)
inductive-set
MWLfSteps-ndet :: (*'exp*, *'id*, *'val*, (*'exp*,*'id*) *MWLfCom*) *TPSteps*
and *MWLfSteps-ndet'* :: (*'exp*, *'id*, *'val*, (*'exp*,*'id*) *MWLfCom*) *TPSteps-curry*
(‹(*1*⟨-,/-⟩) ⇒/ (*1*⟨-,/-⟩)› [*0*,*0*,*0*,*0*] *81*)
where
⟨*V1*,*m1*⟩ ⇒ ⟨*V2*,*m2*⟩ ≡ ((*V1*,*m1*),(*V2*,*m2*)) ∈ *MWLfSteps-ndet* |
⟨*ci*,*m*⟩ → ⟨*c*,*m'*⟩ ⟹ ⟨*Vf* @ [*ci*] @ *Va*,*m*⟩ ⇒ ⟨*Vf* @ *c* @ *Va*,*m'*⟩

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 *Strongly-Secure-Skip-Assign*
imports *MWLf Parallel-Composition*
begin

locale *Strongly-Secure-Programs* =
L? : *MWLf-semantics E BMap*
+ *SS?*: *Strong-Security MWLfSteps-det DA*
for *E* :: (*'exp*, *'id*, *'val*) *Evalfunction*
and *BMap* :: *'val* ⇒ *bool*
and *DA* :: (*'id*, *'d*::*order*) *DomainAssignment*
begin

abbreviation *USdBname* ::*'d* ⇒ (*'exp*, *'id*) *MWLfCom Bisimulation-type*
(‹≈₋›)
where $\approx_d$ ≡ *USdB d*

abbreviation *relatedbyUSdB* :: (*'exp*,*'id*) *MWLfCom list* ⇒ *'d*
⇒ (*'exp*,*'id*) *MWLfCom list* ⇒ *bool* (**infixr** ‹≈₋› *65*)

where $V \approx_d V' \equiv (V,V') \in USdB\ d$

— define when two expressions are indistinguishable with respect to a domain d
definition *d-indistinguishable* :: *'d::order* ⇒ *'exp* ⇒ *'exp* ⇒ *bool*
where
d-indistinguishable d e1 e2 ≡
  $\forall m\ m'.\ ((m =_d m') \longrightarrow ((E\ e1\ m) = (E\ e2\ m')))$

abbreviation *d-indistinguishable'* :: *'exp* ⇒ *'d::order* ⇒ *'exp* ⇒ *bool*
(‹(- ≡- -)›)
where
$e1 \equiv_d e2 \equiv$ *d-indistinguishable d e1 e2*

— symmetry of d-indistinguishable
lemma *d-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-indistinguishable-trans*:
$[\![\ e \equiv_d e';\ e' \equiv_d e''\ ]\!] \Longrightarrow e \equiv_d e''$
by (*simp add*: *d-indistinguishable-def d-equal-def*, *metis*)

theorem *Strongly-Secure-Skip*:
[*skip*] $\approx_d$ [*skip*]
proof −
 define *R0* **where** *R0* = {(*V*::(*'exp*,*'id*) *MWLfCom list*,*V'*::(*'exp*,*'id*) *MWLfCom list*).
   V = [*skip*] ∧ *V'* = [*skip*]}

 have *uptoR0*: *d-Bisimulation-Up-To-USdB d R0*
  proof (*simp add*: *d-Bisimulation-Up-To-USdB-def*, *auto*)
   show *sym R0* **by** (*simp add*: *R0-def sym-def*)
  next
   fix *V V'*
   assume (*V*,*V'*) ∈ *R0*
   thus *length V* = *length V'*
    by (*simp add*: *R0-def*)
  next
   fix *V V' i m1 m1' W m2*
   assume *inR0*: (*V*,*V'*) ∈ *R0*
   assume *irange*: *i* < *length V*
   assume *step*: ⟨*V*!*i*,*m1*⟩ → ⟨*W*,*m2*⟩
   assume *dequal*: $m1 =_d m1'$

   from *inR0* **have** *Vassump*:
    V = [*skip*] ∧ *V'* = [*skip*]
    by (*simp add*: *R0-def*)

with *step irange* **have** *step1*:
$W = [] \wedge m2 = m1$
by (*simp*, *metis MWLf-semantics.MWLfSteps-det-cases*(*1*))

from *Vassump irange* **obtain** *m2'* **where** *step2*:
$\langle V'!i,m1' \rangle \rightarrow \langle [],m2' \rangle \wedge m2' = m1'$
by (*simp*, *metis MWLfSteps-det.skip*)

with *step1 dequal trivialpair-in-USdB* **show** $\exists W'\ m2'.$
$\langle V'!i,m1' \rangle \rightarrow \langle W',m2' \rangle \wedge$
$((W,W') \in R0 \vee W \approx_d W') \wedge m2 =_d m2'$
by *auto*
qed

hence $R0 \subseteq \approx_d$
by (*rule Up-To-Technique*)

thus *?thesis*
by (*simp add*: *R0-def*)

qed

theorem *Strongly-Secure-Assign*:
assumes *d-indistinguishable-exp*: $e \equiv_{DA\ x} e'$
shows $[x := e] \approx_d [x := e']$
proof −
define *R0* **where** $R0 = \{(V,V').\ \exists x\ e\ e'.\ V = [x := e] \wedge V' = [x := e'] \wedge$
$e \equiv_{DA\ x} e'\}$

from *d-indistinguishable-exp* **have** *inR0*: $([x{:=}e],[x{:=}e']) \in R0$
by (*simp add*: *R0-def*)

have *d-Bisimulation-Up-To-USdB d R0*
proof (*simp add*: *d-Bisimulation-Up-To-USdB-def*, *auto*)
from *d-indistinguishable-sym* **show** *sym R0*
by (*simp add*: *R0-def sym-def*, *fastforce*)
next
fix $V\ V'$
assume $(V,V') \in R0$
thus $length\ V = length\ V'$
by (*simp add*: *R0-def*, *auto*)
next
fix $V\ V'\ i\ m1\ m1'\ W\ m2$
assume *inR0*: $(V,V') \in R0$
assume *irange*: $i < length\ V$
assume *step*: $\langle V!i,m1 \rangle \rightarrow \langle W,m2 \rangle$
assume *dequal*: $m1 =_d m1'$

from *inR0* **obtain** $x\ e\ e'$ **where** *Vassump*:

$V = [x := e] \wedge V' = [x := e'] \wedge$
$e \equiv_{DA\ x} e'$
by (*simp add*: *R0-def*, *auto*)

with *step irange* **obtain** *v* **where** *step1*:
$E\ e\ m1 = v \wedge W = [] \wedge m2 = m1(x := v)$
by (*auto*, *metis MWLf-semantics.MWLfSteps-det-cases*(*2*))

from *Vassump irange* **obtain** *m2' v'* **where** *step2*:
$E\ e'\ m1' = v' \wedge \langle V'!i,m1' \rangle \rightarrow \langle [],m2' \rangle \wedge m2' = m1'(x := v')$
by (*auto*, *metis MWLfSteps-det.assign*)

with *Vassump dequal step step1*
have *dequalnext*: $m1(x := v) =_d m1'(x := v')$
by (*simp add*: *d-equal-def d-indistinguishable-def*, *auto*)

with *step1 step2 trivialpair-in-USdB* **show** $\exists\ W'\ m2'.$
$\langle V'!i,m1' \rangle \rightarrow \langle W',m2' \rangle \wedge ((W,W') \in R0 \vee W \approx_d W')$
$\wedge\ m2 =_d m2'$
by *auto*
qed

hence $R0 \subseteq \approx_d$
by (*rule Up-To-Technique*)

with *inR0* **show** *?thesis*
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 *Strongly-Secure-Skip-Assign*
begin

context *Strongly-Secure-Programs*
begin

theorem *Compositionality-Seq*:
  assumes *relatedpart1*: $[c1] \approx_d [c1']$
  assumes *relatedpart2*: $[c2] \approx_d [c2']$
  shows $[c1;c2] \approx_d [c1';c2']$
proof −
  define *R0* **where** $R0 = \{(S1,S2).\ \exists\, c1\ c1'\ c2\ c2'\ W\ W'.$
    $S1 = (c1;c2)\#W \wedge S2 = (c1';c2')\#W' \wedge$
    $[c1] \approx_d [c1'] \wedge [c2] \approx_d [c2'] \wedge W \approx_d W'\}$

  from *relatedpart1 relatedpart2 trivialpair-in-USdB*
  have *inR0*: $([c1;c2],[c1';c2']) \in R0$
    by (*simp add*: *R0-def*)

  have *uptoR0*: *d-Bisimulation-Up-To-USdB d R0*
    proof (*simp add*: *d-Bisimulation-Up-To-USdB-def*, *auto*)
        from *USdBsym*
        show *sym R0*
          by (*simp add*: *sym-def R0-def*, *fastforce*)
      next
        fix *S1 S2*
        assume *inR0*: $(S1,S2) \in R0$
        with *USdBeqlen* **show** *length S1 = length S2*
          by (*auto simp add*: *R0-def*)
      next
        fix *S1 S2 RS m1 m2 m1' i*
        assume *inR0*: $(S1,S2) \in R0$
        assume *irange*: $i <$ *length S1*
        assume *S1step*: $\langle S1!i,m1\rangle \rightarrow \langle RS,m2\rangle$
        assume *dequal*: $m1 =_d m1'$
        from *inR0* **obtain** *c1 c1' c2 c2' V V'*
          where *R0def'*: $S1 = (c1;c2)\#V \wedge S2 = (c1';c2')\#V' \wedge$
          $[c1] \approx_d [c1'] \wedge [c2] \approx_d [c2'] \wedge V \approx_d V'$
          by (*simp add*: *R0-def*, *force*)

        with *irange* **have** *case-distinction1*:
          $i = 0 \vee (V \neq [] \wedge i \neq 0)$
          by *auto*
        moreover
        have *case1*: $i = 0 \Longrightarrow$
          $\exists\, RS'\ m2'.\ \langle S2!i,m1'\rangle \rightarrow \langle RS',m2'\rangle \wedge$
            $((RS,RS') \in R0 \vee RS \approx_d RS') \wedge m2 =_d m2'$
          proof −
            assume *i0*: $i = 0$

              — get the two different sub-cases:
            with *R0def' S1step* **obtain** *c3 W* **where** *case-distinction*:
              $RS = [c2] \wedge \langle c1,m1\rangle \rightarrow \langle [],m2\rangle$
              $\vee\ RS = (c3;c2)\#W \wedge \langle c1,m1\rangle \rightarrow \langle c3\#W,m2\rangle$

by (*simp*, *metis MWLfSteps-det-cases*(*3*))
moreover
— Case 1: first command terminates
{
assume *RSassump*: $RS = [c2]$
assume *StepAssump*: $\langle c1,m1\rangle \rightarrow \langle[],m2\rangle$

from *USdBeqlen*[*of* []] *StepAssump R0def'*
USdB-Strong-d-Bisimulation dequal
strongdB-aux[*of* $d \approx_d i$
$[c1]$ $[c1']$ $m1$ [] $m2$ $m1'$] $i0$
obtain W' $m2'$ **where** *c1c1'reason*:
$\langle c1',m1'\rangle \rightarrow \langle W',m2'\rangle \wedge W' = []$
$\wedge [] \approx_d W' \wedge m2 =_d m2'$
by *fastforce*

with *c1c1'reason* **have** *conclpart*:
$\langle c1';c2',m1'\rangle \rightarrow \langle[c2'],m2'\rangle \wedge m2 =_d m2'$
by (*simp add*: *MWLfSteps-det.seq1*)

with *RSassump R0def' i0* **have** *case1-concl*:
$\exists RS'\ m2'.\ \langle S2!i,m1'\rangle \rightarrow \langle RS',m2'\rangle \wedge$
$((RS,RS') \in R0 \vee RS \approx_d RS') \wedge m2 =_d m2'$
by (*simp*, *rule-tac x*=$[c2']$ **in** *exI*, *auto*)
}
moreover
— Case 2: first command does not terminate
{
assume *RSassump*: $RS = (c3;c2)\#W$
assume *StepAssump*: $\langle c1,m1\rangle \rightarrow \langle c3\#W,m2\rangle$

from *StepAssump R0def' USdB-Strong-d-Bisimulation dequal*
strongdB-aux[*of* $d \approx_d i$ $[c1]$ $[c1']$ $m1$
$c3\#W$ $m2$ $m1'$] $i0$
obtain V'' $m2'$ **where** *c1c1'reason*:
$\langle c1',m1'\rangle \rightarrow \langle V'',m2'\rangle$
$\wedge (c3\#W) \approx_d V'' \wedge m2 =_d m2'$
by *fastforce*

with *USdBeqlen*[*of* $c3\#W$ V''] **obtain** $c3'$ W'
where *V''reason*:
$V'' = c3'\#W' \wedge length\ W = length\ W'$
by (*cases* V'', *force*, *force*)

with *c1c1'reason* **have** *conclpart1*:
$\langle c1';c2',m1'\rangle \rightarrow \langle(c3';c2')\#W',m2'\rangle \wedge m2 =_d m2'$
by (*simp add*: *MWLfSteps-det.seq2*)

from *V''reason c1c1'reason*

USdB-decomp-head-tail[*of c3 W*]
USdB-Strong-d-Bisimulation
have *c3aWinUSDB*:
$[c3] \approx_d [c3'] \wedge W \approx_d W'$
by *blast*

with *R0def'* **have** *conclpart2*:
$((c3;c2)\#W,(c3';c2')\#W') \in R0$
by (*auto simp add*: *R0-def*)

with *i0 RSassump R0def' V''reason conclpart1*
have *case2-concl*:
$\exists RS'\ m2'.\ \langle S2!i,m1' \rangle \rightarrow \langle RS',m2' \rangle \wedge$
$((RS,RS') \in R0 \vee RS \approx_d RS') \wedge m2 =_d m2'$
by (*rule-tac x*=$(c3';c2')\#W'$ **in** *exI*, *auto*)
}
ultimately
show $\exists RS'\ m2'.\ \langle S2!i,m1' \rangle \rightarrow \langle RS',m2' \rangle \wedge$
$((RS,RS') \in R0 \vee RS \approx_d RS') \wedge m2 =_d m2'$
by *blast*
qed
moreover
have *case2*: $[\![\ V \neq []; i \neq 0\ ]\!]$
$\Longrightarrow \exists RS'\ m2'.\ \langle S2!i,m1' \rangle \rightarrow \langle RS',m2' \rangle \wedge$
$((RS,RS') \in R0 \vee RS \approx_d RS') \wedge m2 =_d m2'$
proof −
assume *Vnonempt*: $V \neq []$
assume *inot0*: $i \neq 0$

with *Vnonempt irange R0def'* **have** *i1range*:
$(i - Suc\ 0) < length\ V$
by *simp*

from *inot0 R0def'* **have** *S1ieq*: $S1!i = V!(i - Suc\ 0)$
by *auto*

from *inot0 R0def'* **have** $S2!i = V'!(i - Suc\ 0)$
by *auto*

with *S1ieq R0def' S1step i1range dequal*
USdB-Strong-d-Bisimulation
strongdB-aux[*of d USdB d*
i−Suc 0 V V' m1 RS m2 m1']
show $\exists RS'\ m2'.\ \langle S2!i,m1' \rangle \rightarrow \langle RS',m2' \rangle \wedge$
$((RS,RS') \in R0 \vee RS \approx_d RS') \wedge m2 =_d m2'$
by *force*
qed
ultimately show $\exists RS'\ m2'.\ \langle S2!i,m1' \rangle \rightarrow \langle RS',m2' \rangle \wedge$
$((RS,RS') \in R0 \vee RS \approx_d RS') \wedge m2 =_d m2'$

```
        by auto
      qed

  hence R0 ⊆ ≈d
    by (rule Up-To-Technique)

  with inR0 show ?thesis
    by auto

qed

theorem Compositionality-Fork:
  fixes V::('exp,'id) MWLfCom list
  assumes relatedmain: [c] ≈d [c']
  assumes relatedthreads: V ≈d V'
  shows [fork c V] ≈d [fork c' V']
proof −
  define R0 where R0 = {(F1,F2). ∃ c1 c1' W W'.
    F1 = [fork c1 W] ∧ F2 = [fork c1' W']
    ∧ [c1] ≈d [c1'] ∧ W ≈d W'}
  from relatedmain relatedthreads
  have inR0: ([fork c V],[fork c' V']) ∈ R0
    by (simp add: R0-def)

  have uptoR0: d-Bisimulation-Up-To-USdB d R0
    proof (simp add: d-Bisimulation-Up-To-USdB-def, auto)
      from USdBsym show sym R0
        by (simp add: R0-def sym-def, auto)
    next
      fix F1 F2
      assume inR0: (F1,F2) ∈ R0
      with R0-def USdBeqlen show length F1 = length F2
        by auto
    next
      fix F1 F2 c1V m1 m2 m1' i
      assume inR0: (F1,F2) ∈ R0
      assume irange: i < length F1
      assume F1step: ⟨F1!i,m1⟩ → ⟨c1V,m2⟩
      assume dequal: m1 =d m1'

      from inR0 obtain c1 c1' V V'
        where R0def': F1 = [fork c1 V] ∧ F2 = [fork c1' V'] ∧
        [c1] ≈d [c1'] ∧ V ≈d V'
        by (simp add: R0-def, force)

      from irange R0def' F1step
      have rew: c1V = c1#V ∧ m2 = m1
        by (simp, metis MWLf-semantics.MWLfSteps-det-cases(6))
```

from *irange R0def' MWLfSteps-det.fork* **have** *F2step*:
$\langle F2!i,m1' \rangle \rightarrow \langle c1'\#V',m1' \rangle$
by *force*

from *R0def' USdB-comp-head-tail* **have** *conclpart*:
$((c1\#V,c1'\#V') \in R0 \vee (c1\#V) \approx_d (c1'\#V'))$
by *auto*

with *irange rew inR0 F1step dequal R0def' F2step*
show $\exists\, c1V'\ m2'.\ \langle F2!i,m1' \rangle \rightarrow \langle c1V',m2' \rangle \wedge$
$((c1V,c1V') \in R0 \vee c1V \approx_d c1V') \wedge m2 =_d m2'$
by *fastforce*
qed

hence $R0 \subseteq \approx_d$
by (*rule Up-To-Technique*)

with *inR0* **show** *?thesis*
by *auto*

qed

theorem *Compositionality-If*:
assumes *dind-or-branchesrelated*:
$b \equiv_d b' \vee [c1] \approx_d [c2] \vee [c1'] \approx_d [c2']$
assumes *branch1related*: $[c1] \approx_d [c1']$
assumes *branch2related*: $[c2] \approx_d [c2']$
shows $[\mathit{if}\ b\ \mathit{then}\ c1\ \mathit{else}\ c2\ \mathit{fi}] \approx_d [\mathit{if}\ b'\ \mathit{then}\ c1'\ \mathit{else}\ c2'\ \mathit{fi}]$
proof −
define *R1* **where** $R1 = \{(I1,I2).\ \exists\, c1\ c1'\ c2\ c2'\ b\ b'.$
$I1 = [\mathit{if}\ b\ \mathit{then}\ c1\ \mathit{else}\ c2\ \mathit{fi}] \wedge I2 = [\mathit{if}\ b'\ \mathit{then}\ c1'\ \mathit{else}\ c2'\ \mathit{fi}] \wedge$
$[c1] \approx_d [c1'] \wedge [c2] \approx_d [c2'] \wedge b \equiv_d b'\}$

define *R2* **where** $R2 = \{(I1,I2).\ \exists\, c1\ c1'\ c2\ c2'\ b\ b'.$
$I1 = [\mathit{if}\ b\ \mathit{then}\ c1\ \mathit{else}\ c2\ \mathit{fi}] \wedge I2 = [\mathit{if}\ b'\ \mathit{then}\ c1'\ \mathit{else}\ c2'\ \mathit{fi}] \wedge$
$[c1] \approx_d [c1'] \wedge [c2] \approx_d [c2'] \wedge$
$([c1] \approx_d [c2] \vee [c1'] \approx_d [c2'])\}$

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

from *dind-or-branchesrelated branch1related branch2related*
have *inR0*: $([\mathit{if}\ b\ \mathit{then}\ c1\ \mathit{else}\ c2\ \mathit{fi}],[\mathit{if}\ b'\ \mathit{then}\ c1'\ \mathit{else}\ c2'\ \mathit{fi}]) \in R0$
by (*simp add*: *R0-def R1-def R2-def*)

have *uptoR0*: *d-Bisimulation-Up-To-USdB d R0*
proof (*simp add*: *d-Bisimulation-Up-To-USdB-def*, *auto*)
from *USdBsym d-indistinguishable-sym*
have *symR1*: *sym R1*
by (*simp add*: *sym-def R1-def*, *fastforce*)

from *USdBsym*
have *symR2*: *sym R2*
  by (*simp add*: *sym-def R2-def*, *fastforce*)

from *symR1 symR2* **show** *sym R0*
  by (*simp add*: *sym-def R0-def*)

next

fix *I1 I2*
assume *inR0*: $(I1,I2) \in R0$
thus *length I1* = *length I2*
  by (*simp add*: *R0-def R1-def R2-def*, *auto*)

next

fix *I1 I2 RS m1 m1′ m2 i*
assume *inR0*: $(I1,I2) \in R0$
assume *irange*: $i < length\ I1$
assume *I1step*: $\langle I1!i,m1\rangle \rightarrow \langle RS,m2\rangle$
assume *dequal*: $m1 =_d m1'$

have *inR1case*: $(I1,I2) \in R1$
  $\Longrightarrow \exists RS'\ m2'.\ \langle I2!i,m1'\rangle \rightarrow \langle RS',m2'\rangle \wedge$
    $((RS,RS') \in R0 \vee RS \approx_d RS') \wedge m2 =_d m2'$
  proof −
    assume *inR1*: $(I1,I2) \in R1$

    then obtain *c1 c1′ c2 c2′ b b′* **where** *R1def′*:
      I1 = [*if b then c1 else c2 fi*]
      ∧ *I2* = [*if b′ then c1′ else c2′ fi*] ∧
      $[c1] \approx_d [c1'] \wedge [c2] \approx_d [c2'] \wedge b \equiv_d b'$
      by (*simp add*: *R1-def*, *force*)
    moreover
    — get the two different cases True and False from semantics:
    from *irange R1def′ I1step* **have** *case-distinction*:
      RS = [*c1*] ∧ *BMap* (*E b m1*) = *True* ∨
      RS = [*c2*] ∧ *BMap* (*E b m1*) = *False*
      by (*simp*, *metis MWLf-semantics.MWLfSteps-det-cases*(*4*))
    moreover
      — Case 1: b evaluates to True
    {
      assume *bevalT*: *BMap* (*E b m1*) = *True*
      assume *RSassump*: *RS* = [*c1*]
      from *irange bevalT I1step R1def′ RSassump* **have** *memeq*:
        m2 = *m1*
        by (*simp*, *metis MWLf-semantics.MWLfSteps-det-cases*(*4*))

      from *bevalT R1def′ dequal* **have** *b′evalT*:
        BMap (*E b′ m1′*) = *True*
        by (*simp add*: *d-indistinguishable-def*)

      hence *I2step-case1*:

$\langle$*if b′ then c1′ else c2′ fi*,*m1′*$\rangle \rightarrow \langle$[*c1′*],*m1′*$\rangle$
by (*simp add*: *MWLfSteps-det.iftrue*)

with *irange dequal RSassump memeq R1def′*
have *case1-concl*:
$\exists$ *RS′ m2′*. $\langle$*I2*!*i*,*m1′*$\rangle \rightarrow \langle$*RS′*,*m2′*$\rangle \wedge$
$((RS,RS') \in R0 \vee RS \approx_d RS') \wedge m2 =_d m2'$
by *auto*
}
moreover
— Case 2: b evaluates to False
{
assume *bevalF*: *BMap* (*E b m1*) = *False*
assume *RSassump*: *RS* = [*c2*]
from *irange bevalF I1step R1def′ RSassump* **have** *memeq*:
m1 = *m2*
by (*simp*, *metis MWLf-semantics.MWLfSteps-det-cases*(*4*))

from *bevalF R1def′ dequal* **have** *b′evalF*:
BMap (*E b′ m1′*) = *False*
by (*simp add*: *d-indistinguishable-def*)

hence *I2step-case1*:
$\langle$*if b′ then c1′ else c2′ fi*,*m1′*$\rangle \rightarrow \langle$[*c2′*],*m1′*$\rangle$
by (*simp add*: *MWLfSteps-det.iffalse*)

with *irange dequal RSassump memeq R1def′*
have *case1-concl*:
$\exists$ *RS′ m2′*. $\langle$*I2*!*i*,*m1′*$\rangle \rightarrow \langle$*RS′*,*m2′*$\rangle \wedge$
$((RS,RS') \in R0 \vee RS \approx_d RS') \wedge m2 =_d m2'$
by *auto*
}
ultimately show
$\exists$ *RS′ m2′*. $\langle$*I2*!*i*,*m1′*$\rangle \rightarrow \langle$*RS′*,*m2′*$\rangle \wedge$
$((RS,RS') \in R0 \vee RS \approx_d RS') \wedge m2 =_d m2'$
by *auto*
qed

have *inR2case*: (*I1*,*I2*) $\in$ *R2*
$\Longrightarrow \exists$ *RS′ m2′*. $\langle$*I2*!*i*,*m1′*$\rangle \rightarrow \langle$*RS′*,*m2′*$\rangle \wedge$
$((RS,RS') \in R0 \vee RS \approx_d RS') \wedge m2 =_d m2'$
proof −
assume *inR2*: (*I1*,*I2*) $\in$ *R2*
then obtain *c1 c1′ c2 c2′ b b′* **where** *R2def′*:
I1 = [*if b then c1 else c2 fi*]
$\wedge$ *I2* = [*if b′ then c1′ else c2′ fi*] $\wedge$
$[c1] \approx_d [c1'] \wedge [c2] \approx_d [c2'] \wedge$
$([c1] \approx_d [c2] \vee [c1'] \approx_d [c2'])$

by (*simp add*: *R2-def*, *force*)
moreover
— get the two different cases for the result from semantics:
from *irange R2def′ I1step* **have** *case-distinction-left*:
$(RS = [c1] \vee RS = [c2]) \wedge m2 = m1$
by (*simp*, *metis MWLf-semantics.MWLfSteps-det-cases*(*4*))
moreover
from *irange R2def′ dequal* **obtain** *RS′* **where** *I2step*:
$\langle I2!i, m1' \rangle \rightarrow \langle RS', m1' \rangle$
$\wedge (RS' = [c1'] \vee RS' = [c2']) \wedge m1 =_d m1'$
by (*simp*, *metis MWLfSteps-det.iffalse MWLfSteps-det.iftrue*)
moreover
from *USdBtrans* **have** $[\![[c1] \approx_d [c2]; [c2] \approx_d [c2']]\!]$
$\Longrightarrow [c1] \approx_d [c2']$
by (*unfold trans-def*, *blast*)
moreover
from *USdBtrans* **have** $[\![[c1] \approx_d [c1']; [c1'] \approx_d [c2']]\!]$
$\Longrightarrow [c1] \approx_d [c2']$
by (*unfold trans-def*, *blast*)
moreover
from *USdBsym* **have** $[c1] \approx_d [c2] \Longrightarrow [c2] \approx_d [c1]$
by (*simp add*: *sym-def*)
moreover
from *USdBtrans* **have** $[\![[c2] \approx_d [c1]; [c1] \approx_d [c1']]\!]$
$\Longrightarrow [c2] \approx_d [c1']$
by (*unfold trans-def*, *blast*)
moreover
from *USdBsym* **have** $[c1'] \approx_d [c2'] \Longrightarrow [c2'] \approx_d [c1']$
by (*simp add*: *sym-def*)
moreover
from *USdBtrans* **have** $[\![[c2] \approx_d [c2']; [c2'] \approx_d [c1']]\!]$
$\Longrightarrow [c2] \approx_d [c1']$
by (*unfold trans-def*, *blast*)
ultimately show
$\exists RS'\ m2'.\ \langle I2!i, m1' \rangle \rightarrow \langle RS', m2' \rangle \wedge$
$((RS, RS') \in R0 \vee RS \approx_d RS') \wedge m2 =_d m2'$
by *auto*
qed

from *inR0 inR1case inR2case* **show**
$\exists RS'\ m2'.\ \langle I2!i, m1' \rangle \rightarrow \langle RS', m2' \rangle \wedge$
$((RS, RS') \in R0 \vee RS \approx_d RS') \wedge m2 =_d m2'$
by (*auto simp add*: *R0-def*)
qed

hence $R0 \subseteq \approx_d$
by (*rule Up-To-Technique*)

with *inR0* **show** *?thesis*

by *auto*

qed

theorem *Compositionality-While*:
assumes *dind*: $b \equiv_d b'$
assumes *bodyrelated*: $[c] \approx_d [c']$
shows $[while\ b\ do\ c\ od] \approx_d [while\ b'\ do\ c'\ od]$
proof –
define *R1* **where** $R1 = \{(S1,S2).\ \exists\, c1\ c1'\ c2\ c2'\ b\ b'\ W\ W'.$
$S1 = (c1;(while\ b\ do\ c2\ od))\#W \wedge$
$S2 = (c1';(while\ b'\ do\ c2'\ od))\#W' \wedge$
$[c1] \approx_d [c1'] \wedge [c2] \approx_d [c2'] \wedge W \approx_d W' \wedge b \equiv_d b'\}$

define *R2* **where** $R2 = \{(W1,W2).\ \exists\, c1\ c1'\ b\ b'.$
$W1 = [while\ b\ do\ c1\ od] \wedge W2 = [while\ b'\ do\ c1'\ od] \wedge$
$[c1] \approx_d [c1'] \wedge b \equiv_d b'\}$

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

from *dind bodyrelated*
have *inR0*: $([while\ b\ do\ c\ od],[while\ b'\ do\ c'\ od]) \in R0$
by (*simp add*: *R0-def R1-def R2-def*)

have *uptoR0*: *d-Bisimulation-Up-To-USdB d R0*
proof (*simp add*: *d-Bisimulation-Up-To-USdB-def*, *auto*)
from *USdBsym d-indistinguishable-sym* **have** *symR1*: *sym R1*
by (*simp add*: *sym-def R1-def*, *fastforce*)
from *USdBsym d-indistinguishable-sym* **have** *symR2*: *sym R2*
by (*simp add*: *sym-def R2-def*, *fastforce*)
from *symR1 symR2* **show** *sym R0*
by (*simp add*: *sym-def R0-def*)
next
fix *W1 W2*
assume *inR0*: $(W1,W2) \in R0$
with *USdBeqlen* **show** $length\ W1 = length\ W2$
by (*simp add*: *R0-def R1-def R2-def*, *force*)
next
fix *W1 W2 i m1 m1' RS m2*
assume *inR0*: $(W1,W2) \in R0$
assume *irange*: $i < length\ W1$
assume *W1step*: $\langle W1!i,m1\rangle \rightarrow \langle RS,m2\rangle$
assume *dequal*: $m1 =_d m1'$

from *inR0* **show** $\exists\, RS'\ m2'.\ \langle W2!i,m1'\rangle \rightarrow \langle RS',m2'\rangle \wedge$
$((RS,RS') \in R0 \vee RS \approx_d RS') \wedge m2 =_d m2'$
proof (*simp add*: *R0-def*, *auto*)
assume *inR1*: $(W1,W2) \in R1$

then obtain *c1 c1' c2 c2' b b' V V'*
  where *R1def'*: $W1 = (c1;(while\ b\ do\ c2\ od))\#V$
  $\wedge\ W2 = (c1';(while\ b'\ do\ c2'\ od))\#V' \wedge$
  $[c1] \approx_d [c1'] \wedge [c2] \approx_d [c2'] \wedge V \approx_d V' \wedge b \equiv_d b'$
  by (*simp add*: *R1-def*, *force*)

with *irange* **have** *case-distinction1*: $i = 0 \vee$
  $(V \neq [] \wedge i \neq 0)$
  by *auto*
moreover
have *case1*: $i = 0 \Longrightarrow$
  $\exists RS'\ m2'.\ \langle W2!i,m1' \rangle \rightarrow \langle RS',m2' \rangle \wedge$
  $((RS,RS') \in R1 \vee (RS,RS') \in R2 \vee RS \approx_d RS')$
  $\wedge\ m2 =_d m2'$
  proof –
    assume *i0*: $i = 0$
      — get the two different sub-cases:
    with *R1def' W1step* **obtain** *c3 W* **where** *case-distinction*:
      $RS = [while\ b\ do\ c2\ od] \wedge \langle c1,m1 \rangle \rightarrow \langle [],m2 \rangle$
      $\vee\ RS = (c3;(while\ b\ do\ c2\ od))\#W \wedge \langle c1,m1 \rangle \rightarrow \langle c3\#W,m2 \rangle$
      by (*simp*, *metis MWLfSteps-det-cases*(*3*))
    moreover
      — Case 1: first command terminates
    {
      assume *RSassump*: $RS = [while\ b\ do\ c2\ od]$
      assume *StepAssump*: $\langle c1,m1 \rangle \rightarrow \langle [],m2 \rangle$

      from *USdBeqlen*[*of* []] *StepAssump R1def'*
        USdB-Strong-d-Bisimulation dequal
        strongdB-aux[*of* $d \approx_d i$
        [*c1*] [*c1'*] *m1* [] *m2 m1'*] *i0*
      obtain *W' m2'* **where** *c1c1'reason*:
        $\langle c1',m1' \rangle \rightarrow \langle W',m2' \rangle \wedge W' = []$
        $\wedge\ [] \approx_d W' \wedge m2 =_d m2'$
        by *fastforce*

      with *c1c1'reason* **have** *conclpart1*:
        $\langle c1';(while\ b'\ do\ c2'\ od),m1' \rangle$
        $\rightarrow \langle [while\ b'\ do\ c2'\ od],m2' \rangle \wedge m2 =_d m2'$
        by (*simp add*: *MWLfSteps-det.seq1*)

      from *R1def'* **have** *conclpart2*:
        $([while\ b\ do\ c2\ od],[while\ b'\ do\ c2'\ od]) \in R2$
        by (*simp add*: *R2-def*)

      with *conclpart1 RSassump i0 R1def'*
      have *case1-concl*:
        $\exists RS'\ m2'.\ \langle W2!i,m1' \rangle \rightarrow \langle RS',m2' \rangle \wedge$
        $((RS,RS') \in R1 \vee (RS,RS') \in R2 \vee RS \approx_d RS')$

∧ $m2 =_d m2'$
  by *auto*
}
moreover
— Case 2: first command does not terminate
{
assume *RSassump*: $RS = (c3;(while\ b\ do\ c2\ od))\#W$
assume *StepAssump*: $\langle c1,m1\rangle \rightarrow \langle c3\#W,m2\rangle$

from *StepAssump R1def' USdB-Strong-d-Bisimulation dequal*
  strongdB-aux[*of* $d \approx_d i$
  [*c1*] [*c1'*] *m1 c3#W m2 m1'*] *i0*
obtain *V'' m2'* **where** *c1c1'reason*:
  $\langle c1',m1'\rangle \rightarrow \langle V'',m2'\rangle$
  $\wedge (c3\#W) \approx_d V'' \wedge m2 =_d m2'$
  by *fastforce*

with *USdBeqlen*[*of c3#W V''*] **obtain** *c3' W'*
  where *V''reason*: $V'' = c3'\#W'$
  by (*cases V''*, *force*, *force*)

with *c1c1'reason* **have** *conclpart1*:
  $\langle c1';(while\ b'\ do\ c2'\ od),m1'\rangle \rightarrow$
  $\langle (c3';(while\ b'\ do\ c2'\ od))\#W',m2'\rangle$
  $\wedge\ m2 =_d m2'$
  by (*simp add*: *MWLfSteps-det.seq2*)

from *V''reason*
  c1c1'reason USdB-decomp-head-tail[*of c3 W*]
  USdB-Strong-d-Bisimulation
have *c3aWinUSDB*:
  $[c3] \approx_d [c3'] \wedge W \approx_d W'$
  by *blast*

with *R1def'* **have** *conclpart2*:
  $((c3;(while\ b\ do\ c2\ od))\#W,$
    $(c3';(while\ b'\ do\ c2'\ od))\#W') \in R1$
  by (*simp add*: *R1-def*)

with *i0 RSassump R1def' V''reason conclpart1*
have *case2-concl*:
  $\exists RS'\ m2'.\ \langle W2!i,m1'\rangle \rightarrow \langle RS',m2'\rangle \wedge$
  $((RS,RS') \in R1 \vee (RS,RS') \in R2 \vee RS \approx_d RS')$
  $\wedge\ m2 =_d m2'$
  by *auto*
}
ultimately
show $\exists RS'\ m2'.\ \langle W2!i,m1'\rangle \rightarrow \langle RS',m2'\rangle \wedge$
  $((RS,RS') \in R1 \vee (RS,RS') \in R2 \vee RS \approx_d RS')$

$\wedge$ *m2* $=_d$ *m2'*
by *blast*
qed
moreover
have *case2*: ⟦ $V \neq []$; $i \neq 0$ ⟧
$\Longrightarrow \exists RS'\ m2'.\ \langle W2!i,m1' \rangle \rightarrow \langle RS',m2' \rangle \wedge$
$((RS,RS') \in R1 \vee (RS,RS') \in R2 \vee RS \approx_d RS')$
$\wedge\ m2 =_d m2'$
proof –
assume *Vnonempt*: $V \neq []$
assume *inot0*: $i \neq 0$

with *Vnonempt irange R1def'* **have** *i1range*:
$(i - Suc\ 0) < length\ V$
by *simp*

from *inot0 R1def'* **have** *W1ieq*: $W1!i = V!(i - Suc\ 0)$
by *auto*

from *inot0 R1def'* **have** $W2!i = V'!(i - Suc\ 0)$
by *auto*

with *W1ieq R1def' W1step i1range dequal*
USdB-Strong-d-Bisimulation
strongdB-aux[*of d USdB d*
i–Suc 0 V V' m1 RS m2 m1']
show $\exists RS'\ m2'.\ \langle W2!i,m1' \rangle \rightarrow \langle RS',m2' \rangle \wedge$
$((RS,RS') \in R1 \vee (RS,RS') \in R2 \vee RS \approx_d RS')$
$\wedge\ m2 =_d m2'$
by *force*
qed
ultimately show $\exists RS'\ m2'.\ \langle W2!i,m1' \rangle \rightarrow \langle RS',m2' \rangle \wedge$
$((RS,RS') \in R1 \vee (RS,RS') \in R2 \vee RS \approx_d RS')$
$\wedge\ m2 =_d m2'$
by *auto*
next
assume *inR2*: $(W1,W2) \in R2$

then obtain *c1 c1' b b'* **where** *R2def'*:
$W1 = [while\ b\ do\ c1\ od] \wedge W2 = [while\ b'\ do\ c1'\ od] \wedge$
$[c1] \approx_d [c1'] \wedge b \equiv_d b'$
by (*auto simp add*: *R2-def*)
— get the two different cases:
moreover
from *irange R2def' W1step* **have** *case-distinction*:
$RS = [c1;(while\ b\ do\ c1\ od)] \wedge BMap\ (E\ b\ m1) = True \vee$
$RS = [] \wedge BMap\ (E\ b\ m1) = False$
by (*simp,metis MWLf-semantics.MWLfSteps-det-cases*(*5*))
moreover

— Case 1: b evaluates to True

{

assume *bevalT*: *BMap* (*E b m1*)
assume *RSassump*: *RS* = [*c1*;(*while b do c1 od*)]
from *irange bevalT W1step R2def′ RSassump* **have** *memeq*:
$m2 = m1$
by (*simp*,*metis MWLf-semantics.MWLfSteps-det-cases*(*5*))

from *bevalT R2def′ dequal* **have** *b′evalT*: *BMap* (*E b′ m1′*)
by (*simp add*: *d-indistinguishable-def*)

hence *W2step-case1*:
⟨*while b′ do c1′ od*,*m1′*⟩
→ ⟨[*c1′*;(*while b′ do c1′ od*)],*m1′*⟩
by (*simp add*: *MWLfSteps-det.whiletrue*)

from *trivialpair-in-USdB R2def′* **have** *inWR2*:
([*c1*;(*while b do c1 od*)],
[*c1′*;(*while b′ do c1′ od*)]) ∈ *R1*
by (*auto simp add*: *R1-def*)

with *irange dequal RSassump memeq W2step-case1 R2def′*
have *case1-concl*:
∃ *RS′ m2′*. ⟨*W2*!*i*,*m1′*⟩ → ⟨*RS′*,*m2′*⟩ ∧
((*RS*,*RS′*) ∈ *R1* ∨ (*RS*,*RS′*) ∈ *R2* ∨ $RS \approx_d RS'$)
∧ $m2 =_d m2'$
by *auto*

}

moreover

— Case 2: b evaluates to False

{

assume *bevalF*: *BMap* (*E b m1*) = *False*
assume *RSassump*: *RS* = []
from *irange bevalF W1step R2def′ RSassump* **have** *memeq*:
$m2 = m1$
by (*simp*,*metis MWLf-semantics.MWLfSteps-det-cases*(*5*))

from *bevalF R2def′ dequal* **have** *b′equalF*:
BMap (*E b′ m1′*) = *False*
by (*simp add*: *d-indistinguishable-def*)

hence *W2step-case2*:
⟨*while b′ do c1′ od*,*m1′*⟩ → ⟨[],*m1′*⟩
by (*simp add*: *MWLfSteps-det.whilefalse*)

with *trivialpair-in-USdB irange dequal RSassump*
memeq R2def′
have *case1-concl*:
∃ *RS′ m2′*. ⟨*W2*!*i*,*m1′*⟩ → ⟨*RS′*,*m2′*⟩ ∧

$((RS,RS') \in R1 \vee (RS,RS') \in R2 \vee RS \approx_d RS')$
$\wedge\ m2 =_d m2'$
by *force*
}
ultimately
show $\exists RS'\ m2'.\ \langle W2!i,m1'\rangle \rightarrow \langle RS',m2'\rangle \wedge$
$((RS,RS') \in R1 \vee (RS,RS') \in R2 \vee RS \approx_d RS')$
$\wedge\ m2 =_d m2'$
by *auto*
qed
qed

hence $R0 \subseteq \approx_d$
by (*rule Up-To-Technique*)

with *inR0* **show** *?thesis*
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* =
SSP? : *Strongly-Secure-Programs E BMap DA*
for *E* :: (*'exp*, *'id*, *'val*) *Evalfunction*

and *BMap* :: *'val* ⇒ *bool*
and *DA* :: (*'id*, *'d*::*order*) *DomainAssignment*
+
fixes
AssignSideCondition :: *'id* ⇒ *'exp* ⇒ *bool*
and *WhileSideCondition* :: *'exp* ⇒ *bool*
and *IfSideCondition* ::
'exp ⇒ (*'exp*,*'id*) *MWLfCom* ⇒ (*'exp*,*'id*) *MWLfCom* ⇒ *bool*
assumes *semAssignSC*: *AssignSideCondition x e* $\Longrightarrow$ $e \equiv_{DA\ x}$ e
and *semWhileSC*: *WhileSideCondition e* $\Longrightarrow \forall\, d.\ e \equiv_d e$
and *semIfSC*: *IfSideCondition e c1 c2* $\Longrightarrow \forall\, d.\ e \equiv_d e \vee [c1] \approx_d [c2]$
begin

— Security typing rules for the language commands
inductive
ComSecTyping :: (*'exp*, *'id*) *MWLfCom* ⇒ *bool*
(‹$\vdash_{\mathcal{C}}$ -›)
and *ComSecTypingL* :: (*'exp*,*'id*) *MWLfCom list* ⇒ *bool*
(‹$\vdash_{\mathcal{V}}$ -›)
where
skip: $\vdash_{\mathcal{C}}$ *skip* |
Assign: ⟦ *AssignSideCondition x e* ⟧ $\Longrightarrow \vdash_{\mathcal{C}}$ *x* := *e* |
Fork: ⟦ $\vdash_{\mathcal{C}}$ *c*; $\vdash_{\mathcal{V}}$ *V* ⟧ $\Longrightarrow \vdash_{\mathcal{C}}$ *fork c V* |
Seq: ⟦ $\vdash_{\mathcal{C}}$ *c1*; $\vdash_{\mathcal{C}}$ *c2* ⟧ $\Longrightarrow \vdash_{\mathcal{C}}$ *c1*;*c2* |
While: ⟦ $\vdash_{\mathcal{C}}$ *c*; *WhileSideCondition b* ⟧
$\Longrightarrow \vdash_{\mathcal{C}}$ *while b do c od* |
If: ⟦ $\vdash_{\mathcal{C}}$ *c1*; $\vdash_{\mathcal{C}}$ *c2*; *IfSideCondition b c1 c2* ⟧
$\Longrightarrow \vdash_{\mathcal{C}}$ *if b then c1 else c2 fi* |
Parallel: ⟦ ∀ *i* < *length V*. $\vdash_{\mathcal{C}}$ *V*!*i* ⟧ $\Longrightarrow \vdash_{\mathcal{V}}$ *V*

inductive-cases *parallel-cases*:
$\vdash_{\mathcal{V}}$ *V*

— soundness proof of abstract type system
theorem *ComSecTyping-single-is-sound*:
$\vdash_{\mathcal{C}}$ *c* $\Longrightarrow$ *Strongly-Secure* [*c*]
by (*induct rule*: *ComSecTyping-ComSecTypingL.inducts*(*1*)
[*of - - Strongly-Secure*],
auto simp add: *Strongly-Secure-def*,
metis Strongly-Secure-Skip,
metis Strongly-Secure-Assign semAssignSC,
metis Compositionality-Fork,
metis Compositionality-Seq,
metis Compositionality-While semWhileSC,
metis Compositionality-If semIfSC,
metis parallel-composition)

theorem *ComSecTyping-list-is-sound*:
$\vdash_{\mathcal{V}}$ *V* $\Longrightarrow$ *Strongly-Secure V*

by (*metis ComSecTyping-single-is-sound Strongly-Secure-def parallel-composition parallel-cases*)

end

end

4.2 Example language for Boolean and arithmetic expressions

As and example, we provide a simple example language for instantiating the parameter *'exp* for the language for Boolean and arithmetic expressions.

theory *Expr*
imports *Types*
begin

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

type-synonym (*'val*) *operation* = *'val list* ⇒ *'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* ⇒ (*'id*, *'val*) *State* ⇒ *'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.

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: $d1 \leq d2 = (\textit{if } d1 = d2 \textit{ then True}$
$\quad \textit{else } (\textit{if } d1 = \textit{low then True else False}))$

definition
less-Dom-def: $d1 < d2 = (\textit{if } d1 = d2 \textit{ then False}$
$\quad \textit{else } (\textit{if } d1 = \textit{low then True else False}))$

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 $[\![x \leq y; y \leq z]\!] \Longrightarrow x \leq z$
    unfolding *less-eq-Dom-def* **by** ((*split if-split-asm*)+, *auto*)
  show $[\![x \leq y; y \leq x]\!] \Longrightarrow x = y$
    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 Expr 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* ⇒ *bool*

abbreviation *d-indistinguishable'* :: ('*id*,'*val*) *Expr* ⇒ *Dom*

$\Rightarrow$ ('id,'val) Expr $\Rightarrow$ bool
(‹(- $\equiv_{-}$ -)›)
where
$e1 \equiv_d e2$
$\equiv$ *Strongly-Secure-Programs.d-indistinguishable ExprEval DA d e1 e2*

abbreviation *relatedbyUSdB'* :: (('id,'val) Expr, 'id) MWLfCom list
$\Rightarrow$ Dom $\Rightarrow$ (('id,'val) Expr, 'id) MWLfCom list $\Rightarrow$ bool (**infixr** ‹$\approx_{-}$› 65)
where $V \approx_d V' \equiv (V, V') \in$ *Strong-Security.USdB*
(*MWLf-semantics.MWLfSteps-det ExprEval BMap*) *DA d*

— Security typing rules for expressions - will be part of a side condition
inductive
ExprSecTyping :: ('id, 'val) Expr $\Rightarrow$ Dom set $\Rightarrow$ bool
(‹$\vdash_{\mathcal{E}}$ - : -›)
where
Consts: $\vdash_{\mathcal{E}}$ (Const v) : {d} |
Vars: $\vdash_{\mathcal{E}}$ (Var x) : {DA x} |
Ops: $\forall i <$ length arglist. $\vdash_{\mathcal{E}}$ (arglist!i) : (dl!i)
$\Longrightarrow \vdash_{\mathcal{E}}$ (Op f arglist) : ($\bigcup$ {d. ($\exists i <$ length arglist. d = (dl!i))})

definition *synAssignSC* :: 'id $\Rightarrow$ ('id, 'val) Expr $\Rightarrow$ bool
where
synAssignSC x e $\equiv \exists D$. ($\vdash_{\mathcal{E}}$ e : D $\wedge$ ($\forall d \in D$. (d $\leq$ DA x)))

definition *synWhileSC* :: ('id, 'val) Expr $\Rightarrow$ bool
where
synWhileSC e $\equiv \exists D$. ($\vdash_{\mathcal{E}}$ e : D $\wedge$ ($\forall d{\in}D$. $\forall d'$. $d \leq d'$))

definition *synIfSC* :: ('id, 'val) Expr $\Rightarrow$ (('id, 'val) Expr, 'id) MWLfCom
$\Rightarrow$ (('id, 'val) Expr, 'id) MWLfCom $\Rightarrow$ bool
where
synIfSC e c1 c2 $\equiv$
$\forall d$. ($\neg$ ($e \equiv_d e$) $\longrightarrow$ [c1] $\approx_d$ [c2])

lemma *ExprTypable-with-smallerD-implies-d-indistinguishable*:
$\llbracket \vdash_{\mathcal{E}} e : D'; \forall d' \in D'.\ d' \leq d \rrbracket \Longrightarrow e \equiv_d e$
proof (*induct rule*: *ExprSecTyping.induct*,
simp-all add: *Strongly-Secure-Programs.d-indistinguishable-def*
Strong-Security.d-equal-def, *auto*)
fix *dl* **and** *arglist*::(('id, 'val) Expr) list **and** *f*::'val list $\Rightarrow$ 'val
and *m1*::('id,'val) State **and** *m2*::('id,'val) State
assume *main*: $\forall i <$ length arglist. $\vdash_{\mathcal{E}}$ arglist!i : dl!i $\wedge$
(($\forall d' \in$ (dl!i). $d' \leq d$) $\longrightarrow$
($\forall m\ m'$. ($\forall x$. DA x $\leq$ d $\longrightarrow$ m x = m' x)
$\longrightarrow$ ExprEval (arglist!i) m = ExprEval (arglist!i) m'))
assume *smaller*: $\forall D$. ($\exists i <$ length arglist. D = (dl!i))
$\longrightarrow$ ($\forall d'{\in}D$. $d' \leq d$)
assume *eqstate*: $\forall x$. DA x $\leq$ d $\longrightarrow$ m1 x = m2 x

from *smaller* **have** *irangesubst*:
$\forall\, i < \mathit{length}\ \mathit{arglist}.\ \forall\, d' \in (\mathit{dl}!i).\ d' \leq d$
by *auto*

with *eqstate main* **have**
$\forall\, i < \mathit{length}\ \mathit{arglist}.\ \mathit{ExprEval}\ (\mathit{arglist}!i)\ \mathit{m1}$
$= \mathit{ExprEval}\ (\mathit{arglist}!i)\ \mathit{m2}$
by *force*

hence *substmap*: $(\mathit{ExprEvalL}\ \mathit{arglist}\ \mathit{m1}) = (\mathit{ExprEvalL}\ \mathit{arglist}\ \mathit{m2})$
by (*induct arglist*, *auto*, *force*)

show $f\ (\mathit{ExprEvalL}\ \mathit{arglist}\ \mathit{m1}) = f\ (\mathit{ExprEvalL}\ \mathit{arglist}\ \mathit{m2})$
by (*subst substmap*, *auto*)
qed

interpretation *Type-System-example*: *Type-System ExprEval BMap DA synAssignSC synWhileSC synIfSC*
by (*unfold-locales*, *simp add*: *synAssignSC-def*,
metis ExprTypable-with-smallerD-implies-d-indistinguishable,
simp add: *synWhileSC-def*,
metis ExprTypable-with-smallerD-implies-d-indistinguishable,
simp add: *synIfSC-def*, *metis*)

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. Sabelfeld and D. Sands. Probabilistic noninterference for multithreaded programs. In *Computer Security Foundations Workshop, 2000. CSFW-13. Proceedings. 13th IEEE*, pages 200–214. IEEE, 2000.