**D31.1**

**Formal specification of a generic MILS separation kernel**

<table>
<thead>
<tr>
<th>Project number:</th>
<th>318353</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project acronym:</td>
<td>EURO-MILS</td>
</tr>
<tr>
<td>Project title:</td>
<td>EURO-MILS: Secure European Virtualisation for Trustworthy Applications in Critical Domains</td>
</tr>
<tr>
<td>Start date of the project:</td>
<td>1st October, 2012</td>
</tr>
<tr>
<td>Duration:</td>
<td>36 months</td>
</tr>
<tr>
<td>Programme:</td>
<td>FP7/2007-2013</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Deliverable type:</th>
<th>Report</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deliverable reference number:</td>
<td>ICT-318353 / D31.1 / 1.0</td>
</tr>
<tr>
<td>Activity and Work package contributing to deliverable:</td>
<td>Activity 3 / WP31</td>
</tr>
<tr>
<td>Due date:</td>
<td>September 2013 – M12</td>
</tr>
<tr>
<td>Actual submission date:</td>
<td>31st October, 2013</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Responsible organisation:</th>
<th>OUNL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Editors:</td>
<td>OUNL (Freek Verbeek, Julien Schmaltz)</td>
</tr>
<tr>
<td>Dissemination level:</td>
<td>PU</td>
</tr>
<tr>
<td>Revision:</td>
<td>1.0 (r1.0)</td>
</tr>
</tbody>
</table>

**Abstract:**

We introduce a theory of intransitive non-interference for separation kernels with control. We show that it can be instantiated for a simple API consisting of IPC and events.

**Keywords:**

separation kernel with control, formal model, instantiation, IPC, events, Isabelle/HOL
Editors
OUNL (Freek Verbeek, Julien Schmaltz) (OUNL)

Contributors (ordered according to beneficiary numbers)
Sergey Tverdyshev, Oto Havle, Holger Blasum (SYSGO AG)
Bruno Langenstein, Werner Stephan (Deutsches Forschungszentrum für künstliche Intelligenz / DFKI GmbH)
Abderrahmane Feliachi, Yakoub Nemouchi, Burkhart Wolff (Université Paris Sud)
Freek Verbeek, Julien Schmaltz (Open University of The Netherlands)

Acknowledgment
The research leading to these results has received funding from the European Union’s Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 318353.
Executive Summary

We introduce a theory of intransitive non-interference for separation kernels with control. We show that it can be instantiated for a simple API consisting of IPC and events.
Contents

1 Introduction
   1.1 Binders for the option type .................................................. 3
   1.2 Theorems on lists .................................................................. 3

2 A generic model for separation kernels ................................. 6
   2.1 K (Kernel) ....................................................................... 6
       2.1.1 Execution semantics ...................................................... 7
   2.2 SK (Separation Kernel) .......................................................... 8
       2.2.1 Security for non-interfering domains .............................. 10
       2.2.2 Security for indirectly interfering domains .................... 21
   2.3 ISK (Interruptible Separation Kernel) .................................... 34
   2.4 CISK (Controlled Interruptible Separation Kernel) ............... 48
       2.4.1 Execution semantics ...................................................... 49
       2.4.2 Formulations of security ................................................. 50
       2.4.3 Proofs ..................................................................... 50

3 Instantiation by a separation kernel with concrete actions ...... 57
   3.1 Model of a separation kernel configuration ............................ 57
       3.1.1 Type definitions .......................................................... 57
       3.1.2 Configuration .............................................................. 58
   3.2 Formulation of a subject-subject communication policy and an information flow policy, and showing both can be derived from subject-object configuration data ........................................ 58
       3.2.1 Specification .............................................................. 58
       3.2.2 Derivation ................................................................. 59
   3.3 Separation kernel state and atomic step function ................. 60
       3.3.1 Interrupt points .......................................................... 60
       3.3.2 System state .............................................................. 60
       3.3.3 Atomic step .............................................................. 61
   3.4 Preconditions and invariants for the atomic step ................ 63
       3.4.1 Atomic steps of SK_IPC preserve invariants ..................... 63
       3.4.2 Summary theorems on atomic step invariants ................. 65
   3.5 The view-partitioning equivalence relation ............................ 66
       3.5.1 Elementary properties .................................................. 67
   3.6 Atomic step locally respects the information flow policy ......... 67
       3.6.1 Locally respects of atomic step functions ....................... 67
       3.6.2 Summary theorems on view-partitioning locally respects ... 70
   3.7 Weak step consistency ......................................................... 70
       3.7.1 Weak step consistency of auxiliary functions .................. 70
       3.7.2 Weak step consistency of atomic step functions ............... 72
       3.7.3 Summary theorems on view-partitioning weak step consistency .................. 74
   3.8 Separation kernel model ....................................................... 75
       3.8.1 Initial state of separation kernel model ......................... 75
       3.8.2 Types for instantiation of the generic model ................... 76
       3.8.3 Possible action sequences ............................................ 77
       3.8.4 Control ................................................................. 77
       3.8.5 Discharging the proof obligations .................................. 78
   3.9 Link implementation to CISK: the specific separation kernel is an interpretation of the generic model ................................................................. 87
4 Conclusion
1 Introduction

In Section 2 we introduce a theory of intransitive non-interference for separation kernels with control, based on [3]. We show that it can be instantiated for a simple API consisting of IPC and events (Section 3). The rest of this section gives some auxiliary theories used for Section 2.

1.1 Binders for the option type

theory Option-Binders
imports Option
begin

The following functions are used as binders in the theorems that are proven. At all times, when a result is None, the theorem becomes vacuously true. The expression “\( m \to \alpha \)” means “First compute \( m \), if it is None then return True, otherwise pass the result to \( \alpha \)”. B2 is a short hand for sequentially doing two independent computations. The following syntax is associated to B2: “\( m_1 \parallel m_2 \to \alpha \)” represents “First compute \( m_1 \) and \( m_2 \), if one of them is None then return True, otherwise pass the result to \( \alpha \)”.

definition B :: \('a option \Rightarrow ('a \Rightarrow bool) \Rightarrow bool\) (infixl \(\Rightarrow\) 65)
where
B m \alpha \equiv case m of None \Rightarrow True / \alpha

definition B2 :: \('a option \Rightarrow 'a option \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool\)
where
B2 m1 m2 \alpha \equiv m1 \Rightarrow (\lambda a. m2 \Rightarrow (\lambda b. \alpha a b))

syntax B2 :: ['a option, 'a option, ('a \Rightarrow 'a \Rightarrow bool)] \Rightarrow bool (; ; ; 0, 0, 10) 10

Some rewriting rules for the binders

lemma rewrite-B2-to-cases[simp]:
shows B2 s t f = (case s of None \Rightarrow True \parallel (Some s1) \Rightarrow (case t of None \Rightarrow True \parallel (Some t1) \Rightarrow f s1 t1))
using assms unfolding B2-def B-def by(cases s,cases t,simp+)

lemma rewrite-B-None[simp]:
shows None \Rightarrow \alpha = True
unfolding B-def by(auto)

lemma rewrite-B-m-True[simp]:
shows m \Rightarrow (\lambda a . True) = True
unfolding B-def by(cases m,simp+)

lemma rewrite-B2-cases:
shows (case a of None \Rightarrow \alpha \parallel (case b of None \Rightarrow True \parallel (Some t) \Rightarrow f s t))
= (\forall s t . a = (Some s) \land b = (Some t) \to f s t)
by(cases a,simp,cases b,simp+)

definition strict-equal :: 'a option \Rightarrow 'a \Rightarrow bool
where strict-equal m a \equiv case m of None \Rightarrow False \parallel (Some a') \Rightarrow a' = a

end

1.2 Theorems on lists

theory List-Theorems
imports List
begin

definition lastn :: nat \Rightarrow 'a list \Rightarrow 'a list
where lastn n x = drop ((length x) - n) x

EURO-MILS D31.1
**definition** is-sub-seq :: 'a ⇒ 'a ⇒ 'a list ⇒ bool
**where** is-sub-seq a b x ≡ \exists n . Suc n < length x ∧ x!n = a ∧ x'(Suc n) = b

**definition** prefixes :: 'a list set ⇒ 'a list set
**where** prefixes s ≡ \{ x . \exists n y . n > 0 ∧ y ∈ s ∧ take n y = x\}

**lemma** drop-one [simp]:
  shows x ≠ [] → length x ≥ 1 by (induct x, auto)

**lemma** length-ge-one:
  shows x ≠ [] → length x ≥ 1 by (induct x, auto)

**lemma** take-but-one [simp]:
  shows x ≠ [] → lastn ((length x) - I) x = tl x unfolding lastn-def
  using length-ge-one [where x=x] by auto

**lemma** Suc-m-minus-n [simp]:
  shows m ≥ n → Suc m - n = Suc (m - n) by auto

**lemma** lastn-one-less:
  shows n > 0 ∧ n ≤ length x ∧ lastn n x = (a#y) → lastn (n - I) x = y unfolding lastn-def
  using drop-Suc [where n=length x - n and xs=x] drop-tl [where n=length x - n and xs=x] by (auto)

**lemma** list-sub-implies-member:
  shows ∀ a x . set (a#x) ⊆ Z → a ∈ Z by auto

**lemma** subset-smaller-list:
  shows ∀ a x . set (a#x) ⊆ Z → set x ⊆ Z by auto

**lemma** second-elt-is-hd-tl:
  shows tl x = (a # x') → a = x ! I
  by (cases x, auto)

**lemma** length-ge-2-implies-tl-not-empty:
  shows length x ≥ 2 → tl x ≠ []
  by (cases x, auto)

**lemma** length-It-2-implies-tl-empty:
  shows length x < 2 → tl x = []
  by (cases x, auto)

**lemma** first-second-is-sub-seq:
  shows length x ≥ 2 → is-sub-seq (hd x) (x!l) x
  proof
  assume length x ≥ 2
  hence I: (Suc 0) < length x by auto
  hence x!0 = hd x by (cases x, auto)
  from this I show is-sub-seq (hd x) (x!l) x unfolding is-sub-seq-def by auto
  qed

**lemma** hd-drop-is-nth:
  shows n < length x → hd (drop n x) = x!n
  proof (induct x arbitrary: n)
  case Nil
  thus ?case by simp
  next
  case (Cons a x)
  { have hd (drop n (a # x)) = (a # x) ! n
  proof (cases n)
  case 0
  thus ?thesis by simp
  next
  case (Suc m)
  from Suc Cons show ?thesis by auto
  qed
  }

**proof**
thus \(?\text{case by auto}\)
\text{qed}

\textbf{lemma} def-of-hd:
\textbf{shows} \(y = a \# x \rightarrow \text{hd} y = a\) \textbf{by simp}
\textbf{lemma} def-of-tl:
\textbf{shows} \(y = a \# x \rightarrow \text{tl} y = x\) \textbf{by simp}
\textbf{lemma} drop-yields-results-implies-nbound:
\textbf{shows} \(\text{drop} n x \notin [\ ] \rightarrow n < \text{length} x\)
\textbf{by} (\text{induct} x,\text{auto})
\textbf{lemma} hd-take[\textit{simp}]:
\textbf{shows} \(n > 0 \implies \text{hd} (\text{take} n x) = \text{hd} x\)
\textbf{by} (\text{cases} x,\text{simp},\text{cases} n,\text{auto})
\textbf{lemma} consecutive-is-sub-seq:
\textbf{shows} \(a \# (b \# x) = \text{lastn} n y \implies \text{is-sub-seq} a b y\)
\textbf{proof—}
\textbf{assume} \(1: a \# (b \# x) = \text{lastn} n y\)
\textbf{from} \(1\) \textbf{drop-Suc[\textit{where} n=(\text{length} y) - n \text{ and} xs=y]}\)
\textbf{drop-tl[\textit{where} n=(\text{length} y) - n \text{ and} xs=y]}\)
\textbf{def-of-tl[\textit{where} y=\text{lastn} n y \text{ and} a=a \text{ and} x=b\#x]}\)
\textbf{drop-yields-results-implies-nbound[\textit{where} n=\text{Suc} (\text{length} y - n) \text{ and} x=y]}\)
\textbf{have} \(3: \text{Suc} (\text{length} y - n) < \text{length} y\) \textbf{unfolding} \(\text{lastn-def}\) \textbf{by} \text{auto}
\textbf{from} \(3\) \textbf{hd-drop-is-nth[\textit{where} n=(\text{length} y) - n \text{ and} x=y]}\)
\textbf{def-of-hd[\textit{where} y=\text{drop} (\text{Suc} (\text{length} y - n)) y \text{ and} x=b\#x and a=a]}\)
\textbf{have} \(4: y!((\text{length} y - n) = a\) \textbf{unfolding} \(\text{lastn-def}\) \textbf{by} \text{auto}
\textbf{from} \(3\) \textbf{hd-drop-is-nth[\textit{where} n=\text{Suc} ((\text{length} y) - n) \text{ and} x=y]}\)
\textbf{def-of-hd[\textit{where} y=\text{drop} (\text{Suc} (\text{length} y - n)) y and x=x and a=b]}\)
\textbf{drop-Suc[\textit{where} n=(\text{length} y) - n \text{ and} xs=y]}\)
\textbf{drop-tl[\textit{where} n=(\text{length} y) - n \text{ and} xs=y]}\)
\textbf{def-of-tl[\textit{where} y=\text{lastn} n y and a=a and x=b\#x]}\)
\textbf{have} \(5: y!\text{Suc} (\text{length} y - n) = b\) \textbf{unfolding} \(\text{lastn-def}\) \textbf{by} \text{auto}
\textbf{from} \(3\) \textbf{4} \textbf{5} \textbf{show} \textit{thesis}
\textbf{unfolding} \(\text{is-sub-seq-def}\) \textbf{by} \text{auto}
\textbf{qed}

\textbf{lemma} sub-seq-in-prefixed:
\textbf{assumes} \(\exists y \in\text{prefixes} X. \text{is-sub-seq} a a' y\)
\textbf{shows} \(\exists y \in X. \text{is-sub-seq} a a' y\)
\textbf{proof—}
\textbf{from} \(\text{assms obtain} y \text{where} y: y \in\text{prefixes} X \land \text{is-sub-seq} a a' y\) \textbf{by} \text{auto}
\textbf{then obtain} \(n x \text{ where} x: n > 0 \land x \in X \land \text{take} n x = y\)
\textbf{unfolding} \(\text{prefixes-def}\) \textbf{by} \text{auto}
\textbf{from} \(y\) \textbf{obtain} \(i \text{ where} \text{sub-seq-index} \text{ Suc} i < \text{length} y \land y! i = a \land y! \text{Suc} i = a'\)
\textbf{unfolding} \(\text{is-sub-seq-def}\) \textbf{by} \text{auto}
\textbf{from} \(\text{sub-seq-index} x\) \textbf{have} \(\text{is-sub-seq} a a' x\)
\textbf{unfolding} \(\text{is-sub-seq-def}\) \textbf{using} \(\text{nth-take}\) \textbf{by} \text{auto}
\textbf{from} \(\text{this} x\) \textbf{show} \textit{thesis} \textbf{using} \(\text{metis}\)
\textbf{qed}

\textbf{lemma} set-tl-is-subset:
\textbf{shows} \(\text{set} (\text{tl} x) \subseteq \text{set} x\) \textbf{by} (\text{induct} x,\text{auto})
\textbf{lemma} x-is-hd-snd-tl:
\textbf{shows} \(\text{length} x \geq 2 \rightarrow x = (\text{hd} x) \# x!1 \# \text{tl}(\text{tl} x)\)
\textbf{proof}(\text{induct} x)
\textbf{case} Nil
\textbf{show} \textit{?case by auto}
case (Cons a xs)
  show ?case by (induct xs, auto)
qed

lemma tl-x-not-x:
  shows x ≠ [] → tl x ≠ x by (induct x, auto)

lemma tl-hd-x-not-tl-x:
  shows x ≠ [] ∧ hd x ≠ [] → tl (hd x) ≠ tl x ≠ x using tl-x-not-x by (induct x, simp, auto)
end

2 A generic model for separation kernels

This section defines a detailed generic model of separation kernels called CISK (Controlled Interruptible Separation Kernel). It contains a generic functional model of the behaviour of a separation kernel as a transition system, definitions of the security property and proofs that the functional model satisfies security properties. It is based on Rushby’s approach [2] for noninterference. For an explanation of the model, its structure and an overview of the proofs, we refer to the document entitled “A New Theory of Intransitive Noninterference for Separation Kernels with Control” [3].

The structure of the model is based on locales and refinement:

- locale “Kernel” defines a highly generic model for a kernel, with execution semantics. It defines a state transition system with some extensions to the one used in [2]. The transition system defined here stores the currently active domain in the state, and has transitions for explicit context switches and interrupts and provides a notion of control. As each operation of the system will be split into atomic actions in our model, only certain sequences of actions will correspond to a run on a real system. Therefore, the function run, which applies an execution on a state and computes the resulting new state, is partial and defined for realistic traces only. Later, but not in this locale, we will define a predicate to distinguish realistic traces from other traces. Security properties are also not part of this locale, but will be introduced in the locales to be described next.

- locale “Separation_Kernel” extends “Kernel” with constraints concerning non-interference. The theorem is only sensible for realistic traces; for unrealistic trace it will hold vacuously.

- locale “Interruptible_Separation_Kernel” refines “Separation_Kernel” with interruptible action sequences. It defines function “realistic_trace” based on these action sequences. Therefore, we can formulate a total run function.

- locale “Controlled_Interruptible_Separation_Kernel” refines “Interruptible_Separation_Kernel” with abortable action sequences. It refines function “control” which now uses a generic predicate “aborting” and a generic function “set_error_code” to manage aborting of action sequences.

2.1 K (Kernel)

theory K
imports Main List Set Transitive-Closure List-Theorems Option-Binders
begin

The model makes use of the following types:

'state A state contains information about the resources of the system, as well as which domain is currently active. We decided that a state does not need to include a program stack, as in this model the actions that are executed are modelled separately.

'dom A domain is an entity executing actions and making calls to the kernel. This type represents the names of all domains. Later on, we define security policies in terms of domains.
'action_t Actions of type 'action_t represent atomic instructions that are executed by the kernel. As kernel actions are assumed to be atomic, we assume that after each kernel action an interrupt point can occur.

'execution_t An execution of some domain is the code or the program that is executed by the domain. One call from a domain to the kernel will typically trigger a succession of one or more kernel actions. Therefore, an execution is represented as a list of sequences of kernel actions. Non-kernel actions are not taken into account.

'output_t Given the current state and an action an output can be computed deterministically.

time_t Time is modelled using natural numbers. Each atomic kernel action can be executed within one time unit.

type-synonym ('action-t) execution = 'action-t list list
type-synonym time-t = nat

Function kstep (for kernel step) computes the next state based on the current state s and a given action a. It may assume that it makes sense to perform this action, i.e., that any precondition that is necessary for execution of action a in state s is met. If not, it may return any result. This precondition is represented by generic predicate kprecondition (for kernel precondition). Only realistic traces are considered. Predicate realistic_execution decides whether a given execution is realistic.

Function current returns given the state the domain that is currently executing actions. The model assumes a single-core setting, i.e., at all times only one domain is active. Interrupt behavior is modelled using functions interrupt and cswitch (for context switch) that dictate respectively when interrupts occur and how interrupts occur. Interrupts are solely time-based, meaning that there is an at beforehand fixed schedule dictating which domain is active at which time.

Finally, we add function control. This function represents control of the kernel over the execution as performed by the domains. Given the current state s, the currently active domain d and the execution α of that domain, it returns three objects. First, it returns the next action that domain d will perform. Commonly, this is the next action in execution α. It may also return None, indicating that no action is done. Secondly, it returns the updated execution. When executing action α, typically, this action will be removed from the current execution (i.e., updating the program stack). Thirdly, it can update the state to set, e.g., error codes.

locale Kernel =
    fixes kstep :: 'state-t ⇒ 'action-t ⇒ 'state-t
    and output-f :: 'state-t ⇒ 'action-t ⇒ 'output-t
    and s0 :: 'state-t
    and current :: 'state-t ⇒ 'dom-t
    and cswitch :: time-t ⇒ 'state-t ⇒ 'state-t
    and interrupt :: time-t ⇒ bool
    and kprecondition :: 'state-t ⇒ 'action-t ⇒ bool
    and realistic-execution :: 'action-t execution ⇒ bool
    and control :: 'state-t ⇒ 'dom-t ⇒ 'action-t execution ⇒
        (('action-t option) × 'action-t execution × 'state-t)
    and kinvolved :: 'action-t ⇒ 'dom-t set
begin

2.1.1 Execution semantics

Short hand notations for using function control.

definition next-action :: 'state-t ⇒ ('dom-t ⇒ 'action-t execution) ⇒ 'action-t option
where next-action s execs = fst (control s (current s) (execs (current s)))
definition next-execs :: 'state-t ⇒ ('dom-t ⇒ 'action-t execution) ⇒ ('dom-t ⇒ 'action-t execution)
where next-exec s execs = (fun-upd execs (current s) (fst (snd (control s (current s) (execs (current s))))))
definition next-state: 'state-t ⇒ ('dom-t ⇒ 'action-t execution) ⇒ 'state-t
where next-state s execs = snd (snd (control s (current s) (execs (current s))))

A thread is empty iff either it has no further action sequences to execute, or when the current action sequence is finished and there are no further action sequences to execute.

abbreviation thread-empty: 'action-t execution ⇒ bool
where thread-empty exec ≡ exec = [] ∨ exec = [[]]

Wrappers for function kstep and kprecondition that deal with the case where the given action is None.

definition step where step s oa ≡ case oa of None ⇒ s | (Some a) ⇒ kstep s a
definition precondition: 'state-t ⇒ 'action-t option ⇒ bool
where precondition s a ≡ a → kprecondition s
definition involved
where involved oa ≡ case oa of None ⇒ {} | (Some a) ⇒ kinvolved a

Execution semantics are defined as follows: a run consists of consecutively running sequences of actions. These sequences are interruptable. Run first checks whether an interrupt occurs. When this happens, function cswitch may switch the context. Otherwise, function control is used to determine the next action a, which also yields a new state s'. Action a is executed by executing (step s' a). The current execution of the current domain is updated.

Note that run is a partial function, i.e., it computes results only when at all times the preconditions hold. Such runs are the realistic ones. For other runs, we do not need to – and cannot – prove security. All the theorems are formulated in such a way that they hold vacuously for unrealistic runs.

function run z time-t ⇒ 'state-t option ⇒ ('dom-t ⇒ 'action-t execution) ⇒ 'state-t option
where run 0 s execs = s
| run (Suc n) None execs = None
| interrupt (Suc n) ⇒ run (Suc n) (Some s) execs = run n (Some (cswitch (Suc n) s)) execs
| ¬interrupt (Suc n) ⇒ thread-empty(execs (current s)) ⇒ run (Suc n) (Some s) execs = run n (Some s) execs
| ¬interrupt (Suc n) ⇒ ¬thread-empty(execs (current s)) ⇒ ¬precondition (next-state s execs) (next-action s execs) ⇒ run (Suc n) (Some s) execs = None
| ¬interrupt (Suc n) ⇒ ¬thread-empty(execs (current s)) ⇒ precondition (next-state s execs) (next-action s execs) ⇒ run (Suc n) (Some s) execs = run n (Some (step (next-state s execs) (next-action s execs))) (next-exec s execs)
using not0-implies-Suc by (metis option.exhaust prod-cases3,auto)
termination by lexicographic-order
end
end

2.2 SK (Separation Kernel)

theory SK
imports K
begin

Locale Kernel is now refined to a generic model of a separation kernel. The security policy is represented using function ia. Function vpeq is adopted from Rushby and is an equivalence relation representing whether two states are equivalent from the point of view of the given domain.

We assume constraints similar to Rushby, i.e., weak step consistency, locally respects, and output consistency. Additional assumptions are:

Step Atomicity Each atomic kernel step can be executed within one time slot. Therefore, the domain that is currently active does not change by executing one action.
Time-based Interrupts As interrupts occur according to a prefixed time-based schedule, the domain that is active after a call of switch depends on the currently active domain only (cswitch\_consistency). Also, cswitch can only change which domain is currently active (cswitch\_consistency).

Control Consistency States that are equivalent yield the same control. That is, the next action and the updated execution depend on the currently active domain only (next\_action\_consistent, next\_execs\_consistent), the state as updated by the control function remains in vpeq (next\_state\_consistent, locally\_respects\_next\_state). Finally, function control cannot change which domain is active (current\_next\_state).

### Definition actions-in-execution: 'action-t execution ⇒ 'action-t set where actions-in-execution exec ≡ \{ a . ∃ aseq ∈ set exec . a ∈ set aseq \}

locale Separation-Kernel = Kernel kstep output-f s0 current cswitch interrupt kprecondition realistic-execution control kinvolved

for kstep : 'state-t ⇒ 'action-t ⇒ 'state-t and output-f : 'state-t ⇒ 'action-t ⇒ 'output-t and s0 : 'state-t and current : 'state-t ⇒ 'dom-t — Returns the currently active domain and cswitch : 'time-t ⇒ 'state-t ⇒ 'state-t — Switches the current domain and interrupt : 'time-t ⇒ bool — Returns true if an interrupt occurs in the given state at the given time and kprecondition : 'state-t ⇒ 'action-t ⇒ bool — Returns true if an precondition holds that relates the current action to the state and realistic-execution : 'action-t execution ⇒ bool — In this locale, this function is completely unconstrained. and control : 'state-t ⇒ 'dom-t ⇒ 'action-t execution ⇒ (('action-t option) × 'action-t execution × 'state-t) and kinvolved : 'action-t ⇒ 'dom-t set +

fixes ifp : 'dom-t ⇒ 'dom-t ⇒ bool and vpeq : 'dom-t ⇒ 'state-t ⇒ 'state-t ⇒ bool

assumes vpeq\_transitive : ∀ a b c u . (vpeq u a b ∧ vpeq u b c) ⇒ vpeq u a c and vpeq\_symmetric : ∀ a b u . vpeq u a b ⇒ vpeq u b a and vpeq\_reflexive : ∀ a u . vpeq u a a and ifp\_reflexive : ∀ a u . ifp u a u and weakly\_step\_consistent : ∀ s a t u a . vpeq u s t ∧ vpeq (current s) s t ∧ kprecondition s a ∧ kprecondition t a ∧ current s = current t ⇒ vpeq u (kstep s a) (kstep t a)

and locally\_respects : ∀ a s r . ¬ifp (current s) u ∧ kprecondition s a ⇒ vpeq u s (kstep s a)

and output\_consistent : ∀ a s t . vpeq (current s) s t ∧ current s = current t ⇒ (output-f s a) = (output-f t a)

and step\_atomicity : ∀ a s t . current s = current t ⇒ (kstep s a) = current s

and cswitch\_independent\_of\_state : ∀ n s t . current s = current t ⇒ current (cswitch n s t) = current (cswitch n t)

and cswitch\_consistency : ∀ u s t n . vpeq u s t ⇒ vpeq u (cswitch n s) (cswitch n t)

and next\_action\_consistent : ∀ s t execs . vpeq (current s) s t ∧ (∀ d ∈ involved (next\_action s execs) . vpeq d s t) ∧ current s = current t ⇒ next\_action s execs = next\_action t execs

and next\_execs\_consistent : ∀ s t execs . vpeq (current s) s t ∧ (∀ d ∈ involved (next\_execs s execs) . vpeq d s t) ∧ current s = current t ⇒ fst (snd (control s (current s) (execs (current s))))) = fst (snd (control t (current s) (execs (current s)))))

and next\_state\_consistent : ∀ s t u execs . vpeq (current s) s t ∧ vpeq u s t ∧ current s = current t ⇒ vpeq u (next\_state s execs) (next\_state t execs)

and current\_next\_state : ∀ s execs . current (next\_state s execs) = current s

and locally\_respects\_next\_state : ∀ s u execs . ¬ifp (current s) u ⇒ vpeq u s (next\_state s execs)

and involved\_ifp : ∀ s a t . ∀ d ∈ involved a . kprecondition s (the a) ⇒ ifp d (current s)

and next\_action\_from\_execs : ∀ s execs . next\_action s execs → (∀ a . a ∈ actions\_in\_execution (execs (current s)))

and next\_execs\_subset : ∀ s execs u . actions\_in\_execution (next\_execs s execs u) ⊆ actions\_in\_execution (execs u)

begin

Note that there are no proof obligations on function “interrupt”. Its typing enforces the assumptions
that switching is based on time and not on state. This assumption is sufficient for these proofs, i.e., no further assumptions are required.

2.2.1 Security for non-interfering domains

We define security for domains that are completely non-interfering. That is, for all domains \( u \) and \( v \) such that \( v \) may not interfere in any way with domain \( u \), we prove that the behavior of domain \( u \) is independent of the actions performed by \( v \). In other words, the output of domain \( u \) in some run is at all times equivalent to the output of domain \( u \) when the actions of domain \( v \) are replaced by some other set actions.

A domain is unrelated to \( u \) if and only if the security policy dictates that there is no path from the domain to \( u \).

**abbreviation** unrelated \(:= \ (\text{dom-t} \Rightarrow \text{dom-t} \Rightarrow \text{bool})\)

**where** unrelated \( d \ u \equiv \neg \text{ifp}^{**} d \ u \)

To formulate the new theorem to prove, we redefine purging: all domains that may not influence domain \( u \) are replaced by arbitrary action sequences.

**definition** purge \(:= \ (\text{dom-t} \Rightarrow \text{action-t execution}) \Rightarrow \text{dom-t} \Rightarrow (\text{dom-t} \Rightarrow \text{action-t execution})\)

**where** purge execs \( u \equiv \lambda d . \) (if unrelated \( d \ u \) then

\[
\begin{align*}
&\quad (\text{SOME alpha . realistic-execution alpha}) \\
&\quad \text{else execs d})
\end{align*}
\]

A normal run from initial state \( s_0 \) ending in state \( s_f \) is equivalent to a run purged for domain \((currents_f)\).

**definition** NI-unrelated where NI-unrelated

\[
\equiv \forall \ \text{execs a n . run n (Some s0) execs} \rightarrow \\
(\lambda s_f . \ \text{run n (Some s0) (purge execs (current s-f))} \rightarrow \\
(\lambda s_f2 . \ \text{output-f s-f a = output-f s-f2 a \land current s-f = current s-f2}))
\]

The following properties are proven inductive over states \( s \) and \( t \):

1. Invariably, states \( s \) and \( t \) are equivalent for any domain \( v \) that may influence the purged domain \( u \). This is more general than proving that “vpeq u s t” is inductive. The reason we need to prove equivalence over all domains \( v \) is so that we can use weak step consistency.

2. Invariably, states \( s \) and \( t \) have the same active domain.

**abbreviation** equivalent-states :: \( '\text{state-t option} \Rightarrow '\text{state-t option} \Rightarrow '\text{dom-t} \Rightarrow \text{bool} \)

**where** equivalent-states \( s t u \equiv s \parallel t \rightarrow (\lambda s t . (\forall v . \text{ifp}^{**} v u \rightarrow vpeq v s t) \land \text{current s = current t})\)

Rushby’s view partitioning is redefined. Two states that are initially \( u \)-equivalent are \( u \)-equivalent after performing respectively a realistic run and a realistic purged run.

**definition** view-partitioned::bool where view-partitioned

\[
\equiv \forall \ \text{execs ms mt n u . equivalent-states ms mt u} \rightarrow \\
(\text{run n ms execs} \parallel \\
\text{run n mt (purge execs u)} \rightarrow \\
(\lambda rs rt . vpeq u rs rt \land \text{current rs = current rt}))
\]

We formulate a version of predicate view-partitioned that is on one hand more general, but on the other hand easier to prove inductive over function run. Instead of reasoning over execs and (purge execs u), we reason over any two executions execs1 and execs2 for which the following relation holds:

**definition** purged-relation :: \( '\text{dom-t} \Rightarrow ('\text{dom-t} \Rightarrow '\text{action-t execution}) \Rightarrow ('\text{dom-t} \Rightarrow '\text{action-t execution}) \Rightarrow \text{bool} \)

**where** purged-relation \( u \) execs1 execs2 \( \equiv \forall d . \ \text{ifp}^{**} d u \rightarrow \text{execs1 d = execs2 d} \)
The inductive version of view partitioning says that runs on two states that are \( u \)-equivalent and on
two executions that are purged-related yield \( u \)-equivalent states.

**definition** view-partitioned-ind:bool where view-partitioned-ind

\[ \forall \text{ execs1 execs2 s t n u. equivalent-states s t u \land purged-rel u execs1 execs2} \rightarrow \text{equivalent-states (run n s execs1) (run n t execs2)} \]

A proof that when state \( t \) performs a step but state \( s \) not, the states remain equivalent for any domain
\( v \) that may interfere with \( u \).

**lemma** vpeq-s-t:  
assumes \( \text{prec-t: precondition (next-state t execs2) (next-action t execs2)} \)
assumes \( \text{not-ifp-curr-u: \sim ifp^{**} (current t) u} \)
assumes \( \text{vpeq-s-t: \forall \ v. ifp^{**} v u \rightarrow vpeq v s t} \)
shows \( \forall \ v. ifp^{**} v u \rightarrow vpeq v s (\text{step (next-state t execs2) (next-action t execs2)}) \)

**proof**

\[
\begin{align*}
& \text{fix } v \\
& \text{assume ifp-v-u: ifp^{**} v u}
\end{align*}
\]

\[
\begin{align*}
& \text{from ifp-v-u not-ifp-curr-u have unrelated: \sim ifp^{**} (current t) v using rtranclp-trans by metis} \\
& \text{from this current-next-state[THEN spec,THEN spec,where x1=t]} \\
& \text{ locally-respects[THEN spec,THEN spec,THEN spec,where x1=next-state t execs2] vpeq-reflexive} \\
& \text{ prec-s have vpeq v (next-state t execs2) (step (next-state t execs2) (next-action t execs2))} \\
& \text{ unfolding step-def precondition-def B-def} \\
& \text{ by (cases next-action t execs2,auto)} \\
& \text{ from unrelated this locally-respects-next-state vpeq-transitive have vpeq v t (step (next-state t execs2) (next-action t execs2)) by blast} \\
& \text{ from this and ifp-v-u and vpeq-s-t and vpeq-symmetric and vpeq-transitive have vpeq v s (step (next-state t execs2) (next-action t execs2)) by metis}
\end{align*}
\]

**thus** \(?\)thesis by auto

**qed**

A proof that when state \( s \) performs a step but state \( t \) not, the states remain equivalent for any domain
\( v \) that may interfere with \( u \).

**lemma** vpeq-n-s:  
assumes \( \text{prec-s: precondition (next-state s execs) (next-action s execs)} \)
assumes \( \text{not-ifp-curr-u: \sim ifp^{**} (current s) u} \)
assumes \( \text{vpeq-s-t: \forall \ v. ifp^{**} v u \rightarrow vpeq v s t} \)
shows \( \forall \ v. ifp^{**} v u \rightarrow vpeq v s (\text{step (next-state s execs) (next-action s execs)}) \)

**proof**

\[
\begin{align*}
& \text{fix } v \\
& \text{assume ifp-v-u: ifp^{**} v u}
\end{align*}
\]

\[
\begin{align*}
& \text{from ifp-v-u and not-ifp-curr-u have unrelated: \sim ifp^{**} (current s) v using rtranclp-trans by metis} \\
& \text{from this current-next-state[THEN spec,THEN spec,where x1=s] vpeq-reflexive} \\
& \text{ unrelated locally-respects[THEN spec,THEN spec,THEN spec,where x1=next-state s execs and x2=the (next-action s execs)] prec-s} \\
& \text{ have vpeq v (next-state s execs) (step (next-state s execs) (next-action s execs))} \\
& \text{ unfolding step-def precondition-def B-def} \\
& \text{ by (cases next-action s execs,auto)} \\
& \text{ from unrelated this locally-respects-next-state vpeq-transitive have vpeq v s (step (next-state s execs) (next-action s execs)) by blast} \\
& \text{ from this and ifp-v-u and vpeq-s-t and vpeq-symmetric and vpeq-transitive have vpeq v s (step (next-state s execs) (next-action s execs)) by metis}
\end{align*}
\]

**thus** \(?\)thesis by auto
A proof that when both states $s$ and $t$ perform a step, the states remain equivalent for any domain $v$ that may interfere with $u$. It assumes that the current domain can interact with $u$ (the domain for which is purged).

**Lemma vpeq-ns-nt-ifp-u**

**Assumes**

\[ \text{vpeq-s-t: } \forall v. \text{ifp}^\ast \ast v u \implies v s t \]

and 

\[ \text{current-s-t: } \text{current s = current t} \]

**Shows**

\[ \text{precondition (next-state s execs) } a \land \text{precondition (next-state t execs) } a \implies \text{ifp}^\ast \ast \text{ (current s) } u \implies \forall v. \text{ifp}^\ast \ast v u \implies v s t \]

\[ \text{vpeq (step (next-state s execs) a) (step (next-state t execs) a))} \]

**Proof**

\[
\begin{align*}
\text{assume } & \text{precs: precondition (next-state s execs) } a \land \text{precondition (next-state t execs) } a \\
\text{assume } & \text{ifp-curr: ifp}^\ast \ast \text{ (current s) } u \\
\text{from } & \text{vpeq-s-t have } \text{vpeq-curr-s-t: ifp}^\ast \ast \text{ (current s) } u \implies v s t \text{ by auto} \\
\text{from } & \text{ifp-curr precs next-state-consistent[THEN spec,THEN spec,\textbf{where} } x1=s \text{ and } x=t'] vpeq-curr-s-t vpeq-s-t \\
& \text{current-next-state current-s-t weakly-step-consistent[THEN spec,THEN spec,THEN spec,THEN spec,THEN spec,\textbf{where} } x3=\text{next-state execs and } x2=\text{next-state t execs and } x=\text{the a]} \\
\text{show } & \forall v. \text{ifp}^\ast \ast v u \implies v s t \text{ by (cases a,auto)} \\
\text{unfolding } & \text{step-def precondition-def B-def} \\
& \text{by (cases a,auto)} \\
\end{align*}
\]

**QED**

A proof that when both states $s$ and $t$ perform a step, the states remain equivalent for any domain $v$ that may interfere with $u$. It assumes that the current domain cannot interact with $u$ (the domain for which is purged).

**Lemma vpeq-ns-nt-not-ifp-u**

**Assumes**

\[ \text{purged-a-a2: purged-relation } u \text{ execs execs2} \]

and 

\[ \text{prec-s: precondition (next-state s execs) (next-action s execs)} \]

and 

\[ \text{current-s-t: current s = current t} \]

and 

\[ \text{vpeq-s-t: } \forall v. \text{ifp}^\ast \ast v u \implies v s t \]

**Shows**

\[ \text{ifp}^\ast \ast \text{ (current s) } u \land \text{precondition (next-state t execs2)} \text{ (next-action t execs2)} \implies \forall v. \text{ifp}^\ast \ast v u \implies v s t \]

\[ \text{vpeq (step (next-state s execs) (next-action s execs)) (step (next-state t execs2) (next-action t execs2))} \]

**Proof**

\[
\begin{align*}
\text{assume } & \text{not-ifp: } \neg \text{ifp}^\ast \ast \text{ (current s) } u \\
\text{assume } & \text{prec-t: precondition (next-state t execs2) (next-action t execs2)} \\
\text{fix } & a a' v \\
\text{assume } & \text{ifp-v-uc ifp}^\ast \ast v u \\
\text{from } & \text{not-ifp and purged-a-a2 have } \neg \text{ifp}^\ast \ast \text{ (current s) } u \text{ unfolding purged-relation-def by auto} \\
\text{from } & \text{this and } \text{ifp-v-u have } \neg \text{ifp-curr-v: } \neg \text{ifp}^\ast \ast \text{ (current s) } v \text{ using } \text{transclp-trans by metis} \\
\text{from } & \text{this current-next-state[THEN spec,THEN spec,\textbf{where} } x1=s \text{ and } x=\text{execs}] \text{ prec-s vpeq-reflexive} \\
& \text{locally-respects[THEN spec,THEN spec,THEN spec,\textbf{where} } x=\text{next-state s execs and } x2=\text{the (next-action s execs) and } x=v] \\
\text{have } & \text{vpeq v (next-state s execs) (step (next-state s execs) (next-action s execs))} \\
\text{unfolding } & \text{step-def precondition-def B-def} \\
& \text{by (cases next-action s execs,auto)} \\
\text{from } & \text{not-ifp-curr-v this locally-respects-next-state vpeq-transitive} \\
& \text{have vpeq-s-sns: vpeq v s (step (next-state s execs) (next-action s execs))} \\
& \text{by blast} \\
\text{from } & \text{not-ifp-curr-v current-s-t current-next-state[THEN spec,THEN spec,\textbf{where} } x1=t' \text{ and } x=\text{execs2}] \text{ prec-t} \\
& \text{locally-respects[THEN spec,THEN spec,\textbf{where} } x=\text{next-state t execs2}] \text{ vpeq-reflexive} \\
& \text{have } 0: \text{vpeq v (next-state t execs2) (step (next-state t execs2) (next-action t execs2))} \\
\text{unfolding } & \text{step-def precondition-def B-def} \\
& \text{by (cases next-action t execs2,auto)} \\
\text{from } & \text{not-ifp-curr-v current-s-t current-next-state have } I: \neg \text{ifp}^\ast \ast \text{ (current t) } v \\
\end{align*}
\]
using rtranclp-trans by auto
from 0 ! locally-respects-next-state vpeq-transitive
have vpeq-t-nt: vpeq v t' (step (next-state t' execs2) (next-action t' execs2)) by blast
from vpeq-s-ns and vpeq-t-nt and vpeq-s-t and ifp-v-u and vpeq-symmetric and vpeq-transitive
have vpeq-nb-ns: vpeq v (step (next-state s execs) (next-action s execs)) (step (next-state t' execs2) (next-action t' execs2)) by blast
}
thus ?thesis by auto qed

A run with a purged list of actions appears identical to a run without purging, when starting from two
states that appear identical.

lemma unwinding-implies-view-partitioned-ind:
shows view-partitioned-ind
proof
−
{ fix execs execs2 s t n u
  have equivalent-states s t u ∧ purged-relation u execs execs2 —equivalent-states (run n s execs) (run n t execs2) u
  proof (induct n s execs arbitrary: t u execs2 rule: run.induct)
  case (1 s execs t u execs2)
  show ?case by auto
  next
  case (2 n execs t u execs2)
  show ?case by simp
  next
  case (3 n s execs t u execs2)
  assume interrupt-s: interrupt (Suc n)
  assume IH: (∃t u execs2. equivalent-states (Some (cswitch (Suc n) s)) t u ∧ purged-relation u execs execs2 —equivalent-states (run n (Some (cswitch (Suc n) s)) execs) (run n t execs2) u)
  { fix t'
  assume t = Some t'
  fix rs
  assume rs: run (Suc n) (Some s) execs = Some rs
  fix rt
  assume rt: run (Suc n) (Some t') execs2 = Some rt

  assume vpeq-s-t: ∀ v. ifp'' v u — vpeq v s t'
  assume current-s-t: current s = current t'
  assume purged-a-a2: purged-relation u execs execs2

  — The following terminology is used: states rs and rt (for: run-s and run-t) are the states after a run. States ns
  and nt (for: next-s and next-t) are the states after one step.
  — We prove two properties: the states rs and rt have equal active domains (current-rs-rt) and are vpeq for all
domains v that may influence u (vpeq-rs-rt). Both are proven using the IH. To use the IH, we have to prove that the
properties hold for the next step (in this case, a context switch). Statement current-ns-nt states that after one step
states ns and nt have the same active domain. Statement vpeq-ns-nt states that after one step states ns and nt are
vpeq for all domains v that may influence u (vpeq-rs-rt).

  from current-s-t cswitch-independent-of-state
  have current-ns-nt: current (cswitch (Suc n) s) = current (cswitch (Suc n) t') by blast
  from cswitch-consistency vpeq-s-t
  have vpeq-ns-nt: ∀ v. ifp'' v u — vpeq v (cswitch (Suc n) s) (cswitch (Suc n) t') by auto
  from current-ns-nt vpeq-ns-nt interrupt-s vpeq-reflexive purged-a-a2 current-s-t IH[where u=u and t=Some (cswitch (Suc n) t') and ?execs2.0=execs2]
have current-rs-rt: current rs = current rt using rs rt by(auto)
{
  fix v
  assume ia: ifp"** v u
  from current-rs-nt vpeq-nv-nt ia interrupt-s vpeq-reflexive purged-a-a2 IH[where u=u and t=Some (cswitch (Suc n) t') and ?execs2.0=execs2]
  have vpeq-rs-rt: vpeq v rs rt using rs rt by(auto)
}
from current-rs-rt and this have equivalent-states (Some rs) (Some rt) u by auto

thus ?case by(simp add-option.splits.cases t,simp+)
next
case (4 n execs s t u execs2)
assume not-interrupt: ~interrupt (Suc n)
assume thread-empty-s: thread-empty(execs (current s))
assume IH: (\forall u execs2. equivalent-states (Some s) t u \land purged-relation u execs execs2 \implies equivalent-states
(run n (Some s) execs) (run n t execs2) u)
{
  fix t'
  assume t: t = Some t'
  fix rs
  assume rs: run (Suc n) (Some s) execs = Some rs
  fix rt
  assume rt: run (Suc n) (Some t') execs2 = Some rt

  assume vpeq-s-t: \forall v. ifp"** v u \implies vpeq v s t'
  assume current-s-t: current s = current t'
  assume purged-a-a2: purged-relation u execs execs2

  — The following terminology is used: states rs and rt (for: run-s and run-t) are the states after a run. States ns
  and nt (for: next-s and next-t) are the states after one step.

  — We prove two properties: the states rs and rt have equal active domains (current-rs-rt) and are vpeq for all
  domains v that may influence u (vpeq-rs-rt). Both are proven using the IH. To use the IH, we have to prove that
  the properties hold for the next step (in this case, nothing happens in s as the thread is empty). Statement current-rs-nt
  states that after one step states ns and nt have the same active domain. Statement vpeq-nv-nt states that after one
  step states ns and nt are vpeq for all domains v that may influence u (vpeq-ns-nt).

  from ifp-reflexive and vpeq-s-t have vpeq-s-t-u vpeq u s t' by auto
  from thread-empty-s and purged-a-a2 and current-s-t have purged-a-na2: ~ifp"** (current t') u \implies
  purged-relation u execs (next-exec t' execs2)
  by(unfold next-exec-def,unfold purged-relation-def,auto)
  from step-atomicity current-next-state current-s-t have current-s-nt: current s = current (step (next-state t' execs2))
  (next-action t' execs2))
  unfolding step-def
  by (cases next-action t' execs2,auto)

  — The proof is by case distinction. If the current thread is empty in state t as well (case t-empty), then nothing
  happens and the proof is trivial. Otherwise (case t-not-empty), since the current thread has different executions in
  states s and t, we now show that it cannot influence u (statement not-ifp-curr-t). If in state t the precondition holds
  (case t-prec), locally respects shows that the states remain vpeq. Otherwise, (case t-not-prec), everything holds
  vacuously.

  have current-rs-rt: current rs = current rt
  proof (cases thread-empty(execs2 (current t')) rule:case-split[case-names t-empty t-not-empty])
  case t-empty
  from purged-a-a2 and vpeq-s-t and current-s-t IH[where t=Some t' and u=u and ?execs2.0=execs2]
  have equivalent-states (run n (Some s) execs) (run n (Some t') execs2) u using rs rt by(auto)
  from this not-interrupt t-empty thread-empty-s
show ?thesis using rs rt by(auto)
next
case t-not-empty
  from t-not-empty current-next-state and vpeq-s-t-u and thread-empty-s and purged-a-a2 and current-s-t
  have not-ifp-curr-t \not\in\{ (current (next-state t' execs2)) \} u unfolding purged-relation-def by auto
show ?thesis
proof (cases precondition (next-state t' execs2) (next-action t' execs2) rule :case-split[case-names t-prec t-not-prec])
case t-prec
  from locally-respects-next-state current-next-state t-prec not-ifp-curr-t vpeq-s-t locally-respects vpeq-s-nt
  have vpeq-s-nt: (∀ v . ifp^∗∗ v u → vpeq v s (step (next-state t' execs2) (next-action t' execs2))) by auto
  from vpeq-s-nt purged-a-na2 this current-s-nt not-ifp-curr-t current-next-state
  IH[where t=Some (step (next-state t' execs2) (next-action t' execs2)) and u=u and ?execs2.0=next-execs t' execs2]
  have equivalent-states (run n (Some s) execs) (run n (Some (step (next-state t' execs2) (next-action t' execs2))) (next-execs t' execs2)) u using rs rt by(auto)
next
case t-not-prec
  thus ?thesis using rt t-not-empty not-interrupt by(auto)
qed
qed

{ fix v
  assume ia : ifp^∗∗ v u
  have vpeq v rs rt
  proof (cases thread-empty(execs2 (current t')) rule :case-split[case-names t-empty t-not-empty])
case empty
  from purged-a-a2 and vpeq-s-t and current-s-t IH[where t=Some t' and u=u and ?execs2.0=execs2]
  have equivalent-states (run n (Some s) execs) (run n (Some t' execs2)) u using rs rt by(auto)
next
case t-not-empty
  show ?thesis
  proof (cases precondition (next-state t' execs2) (next-action t' execs2) rule :case-split[case-names t-prec t-not-prec])
case t-prec
  from t-not-empty current-next-state and vpeq-s-t-u and thread-empty-s and purged-a-a2 and current-s-t
  have not-ifp-curr-t : ∼ifp^∗∗ (current (next-state t' execs2)) u unfolding purged-relation-def by auto
  from t-prec current-next-state locally-respects-next-state this and vpeq-s-t and locally-respects and
  vpeq-s-nt
  have vpeq-s-nt: (∃ t . ifp^∗∗ v u → vpeq v s (step (next-state t' execs2) (next-action t' execs2))) by auto
  from vpeq-s-nt purged-a-na2 this current-s-nt not-ifp-curr-t current-next-state
  IH[where t=Some (step (next-state t' execs2) (next-action t' execs2)) and u=u and ?execs2.0=next-execs t' execs2]
  have equivalent-states (run n (Some s) execs) (run n (Some (step (next-state t' execs2) (next-action t' execs2))) (next-execs t' execs2)) u using rs rt by(auto)
next
case t-not-prec
thus \(\text{thesis using}\) rt t-not-empty not-interrupt \(\text{by(\text{auto})}\)
\qquad\qquad\quad \text{qed}
\quad\qquad\quad \text{qed}
\}
\end{quote}

\begin{quote}
\textbf{from current-rs-rt and this have} equivalent-states \((\text{Some } rs) (\text{Some } rt)\) \(u\) \text{ by auto}
\end{quote}
\end{quote}

\begin{quote}
\begin{change}
\textbf{thus} \(\text{case by(\text{simp add:option.splits,cases } t,\text{simp+})}\)
\end{change}
\end{quote}

\begin{quote}
\begin{change}
\begin{itemize}
\item next
\item case \((5n\ \text{execs } s\ t\ u\ \text{execs2})\)
\item assume \(\text{not-interrupt}\): \(~\text{interrupt} (\text{Suc } n)\)
\item assume \(\text{thread-not-empty-s}\): \(\text{thread-empty} (\text{execs } (\text{current } s))\)
\item assume \(\text{not-prec-s}\): \(\sim\ \text{precondition} (\text{next-state } s\ \text{execs}) (\text{next-action } s\ \text{execs})\)
\end{itemize}

\text{— Whenever the precondition does not hold, the entire theorem flattens to True and everything holds vacuously.}
\end{quote}

\begin{quote}
\begin{change}
\begin{itemize}
\item hence \(\text{run} (\text{Suc } n) (\text{Some } s)\) \(\text{execs} = \text{None}\) \text{ using not-interrupt thread-not-empty-s} \text{ by simp}
\item thus \(\text{case by(\text{simp add:option.splits})}\)
\end{itemize}
\end{change}
\end{quote}

\begin{quote}
\begin{change}
\begin{itemize}
\item next
\item case \((6n\ \text{execs } s\ t\ u\ \text{execs2})\)
\item assume \(\text{not-interrupt}\): \(~\text{interrupt} (\text{Suc } n)\)
\item assume \(\text{thread-not-empty-s}\): \(\text{thread-empty} (\text{execs } (\text{current } s))\)
\item assume \(\text{prec-s}\): \(\text{precondition} (\text{next-state } s\ \text{execs}) (\text{next-action } s\ \text{execs})\)
\item assume \(\text{IH}\): \((\forall t\ u\ \text{execs2}.)\)
\item \(\text{equivalent-states} (\text{Some } (\text{step } (\text{next-state } s\ \text{execs}) (\text{next-action } s\ \text{execs}))) (\text{next-exec} s\ \text{execs2} \rightarrow\)
\item \(\text{equivalent-states}\)
\item \(\text{purged-relation } u\ (\text{next-exec} s\ \text{execs})\)
\item \((\text{next-exec} s\ \text{execs})\) \((\text{next-exec} t\ \text{execs2}) u\)
\end{itemize}
\end{change}
\end{quote}

\begin{quote}
\begin{change}
\begin{itemize}
\item \{\n\item \(\text{fix } t'\)
\item \(\text{assume } t.t = \text{Some } t'\)
\item \(\text{fix } rs\)
\item \(\text{assume } rs: \text{run} (\text{Suc } n) (\text{Some } s)\) \(\text{execs} = \text{Some } rs\)
\item \(\text{fix } rt\)
\item \(\text{assume } rt: \text{run} (\text{Suc } n) (\text{Some } t')\) \(\text{execs2} = \text{Some } rt\)
\item \(\text{assume vpeq-s-t}: \forall v. \text{ifp}^{**} v u \rightarrow v\text{peq } v s t'\)
\item \(\text{assume current-s-t}: current s = current t'\)
\item \(\text{assume purged-a-a2}: \text{purged-relation } u\ \text{execs2}\)
\end{itemize}
\end{change}
\end{quote}

\begin{quote}
\begin{change}
\begin{itemize}
\item The following terminology is used: states rs and rt (for: run-s and run-t) are the states after a run. States ns and nt (for: next-s and next-t) are the states after one step.
\item We prove two properties: the states rs and rt have equal active domains (current-rs-rt) and are vpeq for all domains v that may influence u (vpeq-rs-rt). Both are proven using the IH. To use the IH, we have to prove that the properties hold for the next step (in this case, state s executes an action). Statement current-ns-nt states that after one step states ns and nt have the same active domain. Statement vpeq-ns-nt states that after one step states ns and nt are vpeq for all domains v that may influence u (vpeq-rs-rt).
\end{itemize}
\end{change}
\end{quote}

\begin{quote}
\begin{change}
\begin{itemize}
\item Some lemma’s used in the remainder of this case.
\item \(\text{from ifp-reflexive and vpeq-s-t have vpeq-s-t-u vpeq u s t' by auto}\)
\item \(\text{from step-atomicity and current-s-t current-next-state}\)
\item \(\text{have current-ns-nt: current (step (next-state s execs) (next-action s execs)) = current (step (next-state t' execs2) (next-action t' execs2))}\)
\item \(\text{unfolding step-def}\)
\item \(\text{by (cases next-action s execs,cases next-action t' execs2,simp,simp,cases next-action t' execs2,simp,simp)}\)
\item \(\text{from vpeq-s-t have vpeq-curr-s-t: ifp}^{**} (current s) u \rightarrow v\text{peq (current s) s t' by auto}\)
\item \(\text{from prec-s involved-ifp[THEN spec.THEN spec.where } x\text{=next-state s execs and } s\text{=next-action s execs]}\)
\item \(\text{vpeq-s-t have vpeq-involved: ifp}^{**} (current s) u \rightarrow (\forall d\in involved (next-action s execs) . vpeq d s t')\)
\end{itemize}
\end{change}
\end{quote}
using current-next-state
unfolding involved-def precondition-def B-def
by (cases next-action s execs simp auto metis converse-rtranclp-into-rtranclp)
from current-s-t next-execs-consistent vpeq-curr-s-t vpeq-involved
have next-execs-t ifp^{* *}(current s) \Rightarrow next-execs t' execs = next-execs execs
unfolding next-execs-def
by (auto)
from current-s-t purged-a-a2 thread-not-empty-s next-action-consistent[THEN spec,THEN spec,where x1=s and x=t'] vpeq-curr-s-t vpeq-involved
have next-action-s-t ifp^{* *}(current s) \Rightarrow next-action t' execs2 = next-action s execs
by (unfold next-action-def, unfold purged-relation-def, auto)
from purged-a-a2 current-s-t next-execs-consistent[THEN spec,THEN spec,THEN spec,where x2=s and x1=t']
and x=execs]
vpeq-curr-s-t vpeq-involved
have purged-na-na2: purged-relation u (next-execs s execs) (next-execs t' execs2)
unfolding next-execs-def purged-relation-def
by (auto)
from purged-a-a2 and purged-relation-def and thread-not-empty-s and current-s-t have thread-not-empty-t:
ifp^{* *}(current s) \Rightarrow \neg thread-empty(execs2 (current t')) by auto
from step-atomicity current-s-t current-next-state have current-ns-t: current (step (next-state s execs) (next-action s execs)) = current t'
unfolding step-def
by (cases next-action s execs, auto)
from step-atomicity and current-s-t have current-ns-nt: current s = current (step t' (next-action t' execs2))
unfolding step-def
by (cases next-action t' execs2, auto)
from purged-a-a2 have purged-na-na -ifp^{* *}(current s) \Rightarrow purged-relation u (next-execs s execs) execs2
by (unfold next-execs-def, unfold purged-relation-def, auto)

--- The proof is by case distinction. If the current domain can interact with u (case curr-ifp-u), then either in state t the precondition holds (case t-prec) or not. If it holds, then lemma vpeq-ns-nt-ifp-u applies. Otherwise, the proof is trivial as the theorem holds vacuously. If the domain cannot interact with u, (case curr-not-ifp-u), then lemma vpeq-ns-nt-not-ifp-u applies.

have current-rs-rt: current rs = current rt
proof (cases ifp^{* *}(current s) u rule :case-split[case-names curr-ifp-u curr-not-ifp-u])
case curr-ifp-u
show ?thesis
proof (cases precondition (next-state t' execs2) (next-action t' execs2) rule :case-split[case-names prec-t prec-not-t])
case prec-t
have thread-not-empty-t: \neg thread-empty(execs2 (current t')) using thread-not-empty-t curr-ifp-u by auto
from current-ns-nt next-execs-t next-action-s-t purged-a-a2 curr-ifp-u prec-s vpeq-ns-nt-ifp-u[where a=(next-action s execs)] vpeq-s-t current-s-t
have equivalent-states (Some (step (next-state s execs) (next-action s execs))) (Some (step (next-state t' execs2) (next-action t' execs2))) u
unfolding purged-relation-def next-state-def
by auto
from this
H{where u=u and ?execs2.0=(next-execs t' execs2) and t=Some (step (next-state t' execs2) (next-action t' execs2))}]
case current-ns-nt purged-na-na2
have equivalent-states (run n (Some (step (next-state s execs) (next-action s execs))) (next-execs s execs))
 by auto
from prec-t thread-not-empty-t prec-s and this and not-interrupt and thread-not-empty-s and next-action-s-t
show ?thesis using rs rt by auto
next case prec-not-t from curr-ifp-u prec-not-t thread-not-empty-t not-interrupt show ?thesis using rt by simp qed
next case curr-not-ifp-u show ?thesis proof (cases thread-empty(execs2 (current t')) rule :case-split[case-names t-empty t-not-empty])
case t-not-empty show ?thesis proof (cases precondition (next-state t' execs2) (next-action t' execs2) rule :case-split[case-names t-prec t-not-prec])
case t-prec from curr-not-ifp-u t-prec IH [
where w=u and ?execs2.0=(next-execs t' execs2) and t=Some (step (next-state t' execs2) (next-action t' execs2))]
current-ns'nt next-execs-t purged-na-na2 vpeq-ns-nt-not-ifp-u current-s-t vpeq-s-t prec-s purged-a-a2 have equivalent-states (run n (Some (step (next-state s execs) (next-action s execs))) (next-execs s execs)) (run n (Some (step (next-state t' execs2) (next-action t' execs2))) (next-execs t' execs2)) u by auto
from this t-prec curr-not-ifp-u t-not-empty prec-s not-interrupt thread-not-empty-s show ?thesis using rs rt by auto
next case t-not-prec from t-not-prec t-not-empty not-interrupt show ?thesis using rt by simp qed
next case t-empty from curr-not-ifp-u and prec-s and vpeq-s-t and locally-respects and vpeq-ns-t current-next-state locally-respects-next-state have vpeq-ns-t (\' v . ifp'** v u ----> vpeq v (step (next-state s execs) (next-action s execs))) t') by blast
from curr-not-ifp-u IH [where t=Some t' and w=u and ?execs2.0=?execs2] and current-ns-t and next-execs-t and purged-na-a and vpeq-ns-t and this have equivalent-states (run n (Some (step (next-state s execs) (next-action s execs))) (next-execs s execs)) (run n (Some t' execs2) u by auto
from this not-interrupt thread-not-empty-s t-empty prec-s show ?thesis using rs rt by auto qed
qed
{
fix v assume ia: ifp'** v u

have vpeq v rs rt proof (cases ifp'** (current s) u rule :case-split[case-names curr-ifp-u curr-not-ifp-u])
case curr-ifp-u show ?thesis proof (cases precondition (next-state t' execs2) (next-action t' execs2) rule :case-split[case-names t-prec t-not-prec])
case t-prec have thread-not-empty-t' ~thread-empty(execs2 (current t')) using thread-not-empty-t curr-ifp-u by auto
from current-ns-nt next-execs-t next-action-s-t purged-a-a2 curr-ifp-u t-prec prec-s vpeq-ns-nt-ifp-u [where a=(next-action s execs)] vpeq-s-t current-s-t have equivalent-states (Some (step (next-state s execs) (next-action s execs))) (Some (step (next-state t' execs2) (next-action t' execs2))) u
unfolding purged-relation-def next-state-def
by auto
from this
IH[where u=u and ?execs2.0=(next-execs t’ execs2) and t=Some (step (next-state t’ execs2) (next-action t’ execs2))]
current-ns-nt purged-na-na2
have equivalent-states (run n (Some (step (next-state s execs) (next-action s execs))) (next-execs s execs))
(run n (Some (step (next-state t’ execs2) (next-action t’ execs2))) (next-execs t’ execs2)) u
by auto
from ia curr-ifp-u t-prec thread-not-empty-t prec-s and this and not-interrupt and thread-not-empty-s and next-action-s-t
show ?thesis using rs rt by auto
next
case t-not-prec
from curr-ifp-u t-not-prec thread-not-empty-t not-interrupt show ?thesis using rt by simp
qed
next
case curr-not-ifp-u
show ?thesis
proof (cases thread-empty(execs2 (current t’)) rule : case-split [case-names t-empty t-not-empty])
case t-not-empty
show ?thesis
proof (cases precondition (next-state t’ execs2) (next-action t’ execs2) rule : case-split [case-names t-prec t-not-prec])
case t-prec
from curr-not-ifp-u t-prec IH[where u=u and ?execs2.0=(next-execs t’ execs2) and t=Some (step (next-state t’ execs2) (next-action t’ execs2))]
current-ns-nt next-execs-t purged-na-na2 vpeq-ns-nt-not-ifp-u current-s-t vpeq-s-t prec-s purged-a-a2
have equivalent-states (run n (Some (step (next-state s execs) (next-action s execs))) (next-execs s execs))
(run n (Some (step (next-state t’ execs2) (next-action t’ execs2))) (next-execs t’ execs2))
by auto
from ia curr-ifp-u t-prec thread-not-empty-prec-s not-interrupt thread-not-empty-s show ?thesis using rs rt by auto
next
case t-not-prec
from t-not-prec thread-not-empty t not-interrupt show ?thesis using rt by simp
qed
next
case t-empty
from curr-not-ifp-u prec-s and vpeq-s-t and locally-respects and vpeq-ns-t current-next-state locally-respects-next-state
have vpeq-ns-t: (∀ v. ifp∗∗ v u → vpeq v (step (next-state s execs) (next-action s execs)) t’)
by blast
from curr-not-ifp-u IH[where t=Some t’ and u=u and ?execs2.0=execs2] and current-ns-t and next-execs-t
and purged-na-a and vpeq-ns-t and this
have equivalent-states (run n (Some (step (next-state s execs) (next-action s execs))) (next-execs s execs))
(run n (Some t’ execs2) u by auto
from ia this not-interrupt thread-not-empty-s t-empty prec-s show ?thesis using rs rt by auto
qed
qed
)
from current-ns-rt and this have equivalent-states (Some rs) (Some rt) u by auto
)
thus ?case by (simp add :option.splits, cases t,simp+)
qed
Thus ?thesis unfolding view-partitioned-ind-def by auto
qed

From the previous lemma, we can prove that the system is view partitioned. The previous lemma was inductive, this lemma just instantiates the previous lemma replacing s and t by the initial state.

**Lemma unwinding-implies-view-partitioned:**

**Shows** view-partitioned

**Proof**

```plaintext
from assms unwinding-implies-view-partitioned-ind have view-partitioned-inductive view-partitioned-ind by blast
have purged-relation : \forall u execs . purged-relation u execs (purge execs u)
by (unfold purged-relation-def, unfold purge-def, auto)
```

```plaintext
{ fix execs s t n u assume \_ : equivalent-states s t u from this view-partitioned-inductive purged-relation have equivalent-states (run n s execs) (run n t (purge execs u)) u unfolding view-partitioned-ind-def by auto
from this ifp-reflexive have run n s execs \parallel run n t (purge execs u) \rightsquigarrow (\lambda rs rt. vpeq u rs rt \land current rs = current rt)
using r-into-rtranclp unfolding B-def by
(cases run n s execs, simp, cases run n t (purge execs u), simp, auto)
}
thus ?thesis unfolding view-partitioned-def Let-def by auto
qed
```

Domains that many not interfere with each other, do not interfere with each other.

**Theorem unwinding-implies-NI-unrelated:**

**Shows** NI-unrelated

**Proof**

```plaintext
{ fix execs a n from assms unwinding-implies-view-partitioned have vp : view-partitioned by blast
from vp and vpeq-reflexive have \_ : \forall u . (run n (Some s0) execs \parallel run n (Some s0) (purge execs u) \rightsquigarrow (\lambda rs rt. vpeq u rs rt \land current rs = current rt))
unfolding view-partitioned-ind-def by auto
have run n (Some s0) execs \rightarrow (\lambda s-f. run n (Some s0) (purge execs (current s-f)) \rightarrow (\lambda s-f2. output-f s-f2 a \land current s-f = current s-f2))
proof(cases run n (Some s0) execs)
case None thus ?thesis unfolding B-def by simp
next
case (Some rs)
thus ?thesis proof(cases run n (Some s0) (purge execs (current rs)))
case None from Some this show ?thesis unfolding B-def by simp
next
case (Some rt)
from \_ run n (Some s0) execs = Some rs Some \_[THEN spec, where x=\current rs]
have vpeq : vpeq (current rs) rs rt \land current rs = current rt
unfolding B-def by auto
from this output-consistent have output-f rs a = output-f rt a
```
2.2.2 Security for indirectly interfering domains

Consider the following security policy over three domains $A$, $B$, and $C$: $A \leadsto B \leadsto C$, but $A \not\leadsto C$. The semantics of this policy is that $A$ may communicate with $C$, but only via $B$. No direct communication from $A$ to $C$ is allowed. We formalize these semantics as follows: without intermediate domain $B$, domain $A$ cannot flow information to $C$. In other words, from the point of view of domain $C$ the run where domain $B$ is inactive must be equivalent to the run where domain $B$ is inactive and domain $A$ is replaced by an attacker. Domain $C$ must be independent of domain $A$, when domain $B$ is inactive.

The aim of this subsection is to formalize the semantics where $A$ can write to $C$ via $B$ only. We define two ipurge functions. The first purges all domains $d$ that are intermediary for some other domain $v$.

An intermediary for $u$ is defined as a domain $d$ for which there exists an information flow from some domain $v$ to $u$ via $d$, but no direct information flow from $v$ to $u$ is allowed.

\begin{definition}
intermediary :: dom-t \Rightarrow dom-t \Rightarrow bool
where\ intermediary d u \equiv \exists v . \text{ifp}^{**} v d \land \text{ifp} d u \land \neg \text{ifp} v u \land d \neq u
\end{definition}

\begin{primrec}
remove-gateway-communications :: dom-t \Rightarrow action-t \Rightarrow action-t
\end{primrec}

\begin{definition}
remove-gateway-communications u [\_] = [\_]
remove-gateway-communications u (aseq # exec) = (if \exists a \in set aseq . \exists v . intermediary v u \land v \in involved (Some a) then [\_] else aseq) # (remove-gateway-communications u exec)
\end{definition}

\begin{definition}
ipurge-l ::
('action-t execution) \Rightarrow dom-t \Rightarrow ('action-t execution)
where
ipurge-l execs u \equiv \lambda d . if intermediary d u then [\_]
else if d = u then remove-gateway-communications u (execs u)
else execs d
\end{definition}

The second ipurge removes both the intermediaries and the indirect sources. An indirect source for $u$ is defined as a domain that may indirectly flow information to $u$, but not directly.

\begin{abbreviation}
ind-source :: dom-t \Rightarrow dom-t \Rightarrow bool
where\ ind-source d u \equiv \text{ifp}^{**} d u \land \neg \text{ifp} d u
\end{abbreviation}

\begin{definition}
ipurge-r ::
('action-t execution) \Rightarrow dom-t \Rightarrow ('action-t execution)
where
ipurge-r execs u \equiv \lambda d . if intermediary d u then [\_]
else if ind-source d u then SOME alpha . realistic-execution alpha
else if d = u then remove-gateway-communications u (execs u)
else execs d
\end{definition}

For a system with an intransitive policy to be called secure for domain $u$ any indirect source may not flow information towards $u$ when the intermediaries are purged out. This definition of security allows the information flow $A \leadsto B \leadsto C$, but prohibits $A \leadsto C$.

\begin{definition}
NI-indirect-sources :: bool
\end{definition}
where \( \text{NI-indirect-sources} \equiv \forall \text{execs a n. run n (Some s0) execs} \Rightarrow \)
\[(\lambda s . f . (\text{run n (Some s0)} (\text{ipurge-l execs (current s-f)}) || \text{run n (Some s0)} (\text{ipurge-r execs (current s-f)})) \Rightarrow \)
\[(\lambda s-l s-r . \text{output-f s-l a = output-f s-r a}))\]

This definition concerns indirect sources only. It does not enforce that an unrelated domain may not
flow information to \( u \). This is expressed by “secure”.

This allows us to define security over intransitive policies.

**Definition**

\[
\text{isecure} \equiv \text{NI-indirect-sources} \land \text{NI-unrelated}
\]

**Abbreviation**

\[
\text{iequivalent-states} \equiv \text{state-t option} \Rightarrow \text{state-t option} \Rightarrow \text{dom-t} \Rightarrow \text{bool}
\]

where \( \text{iequivalent-states} s t u \equiv s \| t \rightarrow (\lambda s t . (\forall v . \text{ifp v u} \land \neg \text{intermediary v u} \rightarrow \text{vpeq v s t}) \land \text{current s = current t}) \)

**Definition**

\[
\text{does-not-communicate-with-gateway} \equiv \forall a . a \in \text{actions-in-execution (execs u)} \rightarrow (\forall v . \text{intermediary v u} \rightarrow v \notin \text{involved (Some a)})
\]

**Definition**

\[
\text{iview-partitioned} \equiv \forall \text{execs ms mt n u. iequivalent-states ms mt u} \rightarrow (\text{run n ms (ipurge-l execs u)} || \text{run n mt (ipurge-r execs u)} \Rightarrow (\lambda rs rt . \text{vpeq u rs rt} \land \text{current rs = current rt}))
\]

**Definition**

\[
\text{ipurged-relation1} \equiv \text{dom-t} \Rightarrow (\text{dom-t} \Rightarrow \text{action-t execution}) \Rightarrow (\text{dom-t} \Rightarrow \text{action-t execution}) \Rightarrow \text{bool}
\]

where \( \text{ipurged-relation1 u execs1 execs2} \equiv \forall d . (\text{ifp d u} \rightarrow \text{execs1 d = execs2 d}) \land (\text{intermediary d u} \rightarrow \text{execs1 d = []}) \)

Proof that if the current is not an intermediary for \( u \), then all domains involved in the next action are
\( \text{vpeq} \).

**Lemma**

\[
\text{vpeq-involved-domains:}
\]

**Assumptions**

\[
\text{ifp-curr: ifp (current s) u}
\]

and \( \text{not-intermediary-curr: \neg \text{intermediary (current s) u}} \)

and \( \text{nogateway-comm: \text{does-not-communicate-with-gateway u execs}} \)

and \( \text{vpeq-s-t: \forall v . \text{ifp v u} \land \neg \text{intermediary v u} \rightarrow \text{vpeq v s t'}} \)

and \( \text{prec-s: precondition (next-state s execs) (next-action s execs}} \)

shows \( \forall d \in \text{involved (next-action s execs)} . \text{vpeq d s t'} \)

**Proof**

- \{
  - \text{fix v}
  - \text{assume involved: v \in involved (next-action s execs)}
  - \text{from this prec-s involved-ifp[THEN spec,THEN spec,where x1=next-state s execs and x=next-action s execs]}\]
  - \text{have ifp-v-curr: ifp v (current s)}
  - \text{using current-next-state}
  - \text{unfolding involved-def precondition-def B-def}
  - \text{by (cases next-action s execs.auto)}
  - \text{have vpeq v s t'}
  - \text{proof--}
    \{
      - \text{assume ifp v u \land \neg \text{intermediary v u}}
      - \text{from this vpeq-s-t}
      - \text{have vpeq v s t' by (auto)}
    \}
- \text{moreover}
{ assume not-intermediary-v: intermediary v u from ifp-curr not-intermediary-curr ifp-v-curr not-intermediary-v have curr-is-u: current s = u using rtranclp-trans r-into-rtranclp by (metis intermediary-def) from curr-is-u next-action-from-exec [THEN spec, THEN spec, where x=execs and x1=s] not-intermediary-v involved no-gateway-comm [unfolded does-not-communicate-with-gateway-def, THEN spec, where x=the (next-action s execs)] have False unfolding involved-def B-def by (cases next-action s execs, auto) hence vpeq v s t' by auto }

moreover
{ assume intermediary-v: ~ ifp v u from ifp-curr not-intermediary-curr ifp-v-curr intermediary-v have False unfolding intermediary-def by auto hence vpeq v s t' by auto }

ultimately show vpeq v s t' unfolding intermediary-def by auto qed

thus ?thesis by auto qed

Proof that purging removes communications of the gateway to domain u.

lemma ipurge-l-removes-gateway-communications:
shows does-not-communicate-with-gateway u (ipurge-l execs u)
proof--
{ fix aseq u execs a v assume 1: aseq ∈ set (remove-gateway-communications u (execs u)) assume 2: a ∈ set aseq assume 3: intermediary v u have 4: v \not\in involved (Some a) proof--
{ fix as'\_action-t fix aseq u exec v have aseq ∈ set (remove-gateway-communications u exec) \land a ∈ set aseq \land intermediary v u \longrightarrow v \not\in involved (Some a) by (induct exec, auto) }
from 1 2 3 this show ?thesis by metis qed }
}
from this show ?thesis unfolding does-not-communicate-with-gateway-def ipurge-l-def actions-in-execution-def by auto qed

Proof of view partitioning. The lemma is structured exactly as lemma unwinding_implies_view_partitioned_ind and uses the same convention for naming.

lemma iunwinding_implies_view_partitioned1:
shows iview-partitioned
proof
{
  fix u execs execs2 s t n
  have does-not-communicate-with-gateway u execs ∧ iequiv-alent-states s t u ∧ ipurged-relation1 u execs execs2
          → iequivalent-states (run n s execs) (run n t execs2) u
  proof (induct n s execs arbitrary: t u execs2 rule: run.induct)
  case (1 s execs t u execs2)
    show ?case by auto
  next
  case (2 n execs t u execs2)
    show ?case by simp
  next
  case (3 n s execs t u execs2)
    assume interrupt-s: interrupt (Suc n)
    assume IH: (∀ t u execs2. does-not-communicate-with-gateway u execs ∧ iequiv-alent-states (Some (cswitch (Suc n) s)) t u ∧ ipurged-relation1 u execs execs2 → iequivalent-states (run n (Some (cswitch (Suc n) s)) execs) (run n t execs2) u)
    { fix t' = 'state-t
      assume t = Some t'
      fix rs
      assume rt: run (Suc n) (Some t') execs2 = Some rt
      fix rt
      assume no-gateway-comm: does-not-communicate-with-gateway u execs
      assume vpeq-s-r: ∀ v. ifp v u ∧ ¬intermediary v u → vpeq v s t'
      assume current-s-r: current s = current t'
      assume purged-a-a2: ipurged-relation1 u execs execs2
      from current-s-t cswitch-independent-of-state
      have current-ns-nt: current (cswitch (Suc n) s) = current (cswitch (Suc n) t')
        by blast
      from cswitch-consistency vpeq-s-t
      have vpeq-ns-nt: ∀ v. ifp v u ∧ ¬intermediary v u → vpeq v (cswitch (Suc n) s) (cswitch (Suc n) t')
        by auto
      from no-gateway-comm current-ns-nt vpeq-ns-nt interrupt-s vpeq-reflexive current-s-t purged-a-a2 IH[where
        u=u and t=Some (cswitch (Suc n) t') and ?execs2.0=execs2]
      have current-rt-r: current rs = current rt using rs rt by(auto)
        { fix v
          assume ia: ifp v u ∧ ¬intermediary v u
          from no-gateway-comm-current-ns-nt vpeq-ns-nt vpeq-reflexive ia current-s-t purged-a-a2
          IH[where u=u and t=Some (cswitch (Suc n) t') and ?execs2.0=execs2]
          have vpeq v rs rt using rs rt by(auto)
        } from current-rt-r and this have iequiv-alent-states (Some rs) (Some rt) u by auto
    } thus ?case by(simp add:option.splits.cases t,simp+)
  next
  case (4 n execs s t u execs2)
    assume not-interrupt: ¬interrupt (Suc n)
    assume thread-empty-s: thread-empty(execs (current s))
    assume IH: (∀ t u execs2. does-not-communicate-with-gateway u execs ∧ iequiv-alent-states (Some s) t u ∧ ipurged-relation1 u execs execs2 → iequiv-alent-states (run n (Some s) execs) (run n t execs2) u)
\[
\]
\begin{verbatim}
{ 
  fix t'

  assume t: t = Some t' 
  fix rs 
  assume rs: run (Suc n) (Some s) execs = Some rs 
  fix rt 
  assume rt: run (Suc n) (Some t') execs2 = Some rt 

  assume no-gateway-comm: does-not-communicate-with-gateway u execs 
  assume vpeq-s-t: \( \forall \ ), ifp v u \land \neg \intermediary v u \rightarrow vpeq v s t' 
  assume current-s-t: current s = current t' 
  assume purged-a-a2: ipurged-relation1 u execs execs2 

  from ifp-reflexive vpeq-s-t have vpeq-u-s-t: vpeq u s t' unfolding intermediary-def by auto 
  from step-atomicity current-next-state current-s-t have current-s-nt: current s = current (step (next-state t' execs2)) 
  unfolding step-def
  by (cases next-action s execs,cases next-action t' execs2,simp,simp,cases next-action t' execs2,simp,simp) 
  from vpeq-s-t have vpeq-curr-s-t: ifp (current s) u \land \neg \intermediary (current s) u \rightarrow vpeq (current s) s t' by auto 
  have iequivalent-states (run (Suc n) (Some s) execs) (run (Suc n) (Some t') execs2) u 
  proof(cases thread-empty(exec2 (current t'))) 
  case True 
    from purged-a-a2 and vpeq-s-t and current-s-t IH[where t=Some t' and u=u and ?execs2.0=?execs2] 
  no-gateway-comm 
    have iequivalent-states (run n (Some s) execs) (run n (Some t') execs2) u using rs rt by(auto) 
  from this not-interrupt True thread-empty-s 
    show ?thesis using rs rt by(auto) 
  next 
  case False 
    have prec-t: precondition (next-state t' execs2) (next-action t' execs2) 
    proof-- 
      { 
        assume not-prec-t: \neg \text{precondition} (next-state t' execs2) (next-action t' execs2) 
        hence run (Suc n) (Some t') execs2 = None using not-interrupt False not-prec-t by (simp) 
        from this have False using rt by(simp add:option.splits) 
      } 
      thus ?thesis by auto 
  qed 

  from False purged-a-a2 thread-empty-s current-s-t 
  have 1: ind-source (current t') u \lor unrelated (current t') u unfolding ipurged-relation1-def intermediary-def 
  by auto 
  { 
    fix v 
    assume ifp-v: ifp v u 
    assume v-not-intermediary: \neg \intermediary v u 

    from 1 ifp-v v-not-intermediary have not-ifp-curr-v: \neg ifp (current t') v unfolding intermediary-def by auto 
    from not-ifp-curr-v prec-t locally-respects[THEN spec,THEN spec,THEN spec,where x1=next-state t' execs2 and x=v and x2=the (next-action t' execs2)] 
      current-next-state vpeq-reflexive 
    have vpeq v (next-state t' execs2) (step (next-state t' execs2)) (next-action t' execs2)) 
    unfolding step-def precondition-def B-def 
    by (cases next-action t' execs2,auto) 
    from this vpeq-transitive not-ifp-curr-v locally-respects-next-state 
  }
\end{verbatim}
have \( vpeq\text{-}nt \): \( vpeq\ v\ t' \ (\text{step} \ (\text{next-state} \ t' \ \text{execs2}) \ (\text{next-action} \ t' \ \text{execs2})) \)
by blast
from \( vpeq\text{-}s\text{-}t\) ifp\(\ v\ \text{not-intermediary} \ vpeq\text{-}nt\) vpeq\text{-}transitive vpeq\text{-}symmetric vpeq\text{-}reflexive
have \( vpeq\ v\ s \ (\text{step} \ (\text{next-state} \ t' \ \text{execs2}) \ (\text{next-action} \ t' \ \text{execs2})) \)
by (metis)
}
hence \( vpeq\text{-}ns\text{-}nt\): \( \forall \ v. \ \text{ifp}\ v\ u \land \neg \text{intermediary} \ v\ u \implies vpeq\ v\ s \ (\text{step} \ (\text{next-state} \ t' \ \text{execs2}) \ (\text{next-action} \ t' \ \text{execs2})) \)
by auto
from \( \text{False}\) purged-a\(\ a\text{-}a2\) current-s\(\text{-}t\) thread-empty-s have purged-a\(\ a\text{-}na2\) ipurged-relation\(1\) u execs (next-exec\(s\text{-}t\) execs2)
unfolding ipurged-relation\(1\)\text{-}def next-exec\(s\text{-}t\) execs2 by (auto)
from vpeq\ns\-nt no-gateway-comm
and \( \text{IH}[\text{where}\ t=\text{Some} \ (\text{step} \ (\text{next-state} \ t' \ \text{execs2}) \ (\text{next-action} \ t' \ \text{execs2})) \text{ and } ?\text{execs2.0}=(\text{next-exec}\(s\text{-}t\) execs2) \text{ and } u=u]\)
and current-s\-nt purged-a\(\ a\text{-}na2\)
have eq\-ns\-nt: iequivalent-states (run \( n \) (Some \( s \) execs)
\(\text{run} \ (\text{Some} \ (\text{step} \ (\text{next-state} \ t' \ \text{execs2}) \ (\text{next-action} \ t' \ \text{execs2}))) \)(next-exec\(s\text{-}t\) execs2)) u by auto
from prec\-t eq\-ns\-nt not-interrupt False thread-empty-s
show ?thesis using \( t \) \( rs \) \( \text{by} \) (auto)
qed
}
thus ?case by (simp add: option.splits cases \( t,\text{simp} \))
next case (5 \( n \) execs \( s \) \( t \) \( u \) execs2)
assume not-interrupt:\( \neg \text{interrupt} \ (\text{Suc} \ n) \)
assume thread\-not-empty\-s:\( \neg \text{thread-empty}(\text{execs} \ (\text{current} \ s)) \)
assume prec\-s:\ precondition \( (\text{next-state} \ s \ \text{execs}) \ (\text{next-action} \ s \ \text{execs}) \)
hence run \( (\text{Suc} \ n) \) (Some \( s \) execs = None using not-interrupt\( \neg \text{thread-not-empty-s} \) by simp
thus ?case by (simp add: option.splits)
next case (6 \( n \) execs \( s \) \( t \) \( u \) execs2)
assume not-interrupt:\( \neg \text{interrupt} \ (\text{Suc} \ n) \)
assume thread\-not-empty\-s:\( \neg \text{thread-empty}(\text{execs} \ (\text{current} \ s)) \)
assume prec\-s:\ precondition \( (\text{next-state} \ s \ \text{execs}) \ (\text{next-action} \ s \ \text{execs}) \)
assume \( \text{IH}:[\forall u \text{execs2.} \ \text{does-not-communicate-with-gateway}\ u \ (\text{next-exec}\(s\text{-}s\) execs) \land \text{iequivalent-states}\ (\text{Some} \ (\text{step} \ (\text{next-state} \ s \ \text{execs}) \ (\text{next-action} \ s \ \text{execs}))) \ t\ u \land \text{ipurged-relation}\(1\) u \ (\text{next-exec}\(s\text{-}s\) execs2) \ \text{iequivalent-states} \
\ (\text{run} \ (\text{Some} \ (\text{step} \ (\text{next-state} \ s \ \text{execs}) \ (\text{next-action} \ s \ \text{execs}))) \)(next-exec\(s\text{-}s\) execs2) u)

{  
fix \( t'\)
assume \( t\text{': } t = \text{Some} \ t'\)
fix \( rs\)
assume \( rs\text{': } \text{run} \ (\text{Suc} \ n) \ (\text{Some} \ s) \ \text{execs} = \text{Some} \ rs\)
fix \( rt\)
assume \( rt\text{': } \text{run} \ (\text{Suc} \ n) \ (\text{Some} \ t') \ \text{execs2} = \text{Some} \ rt\)
assume no\-gateway\-comm: \text{does-not-communicate-with-gateway}\ u \ \text{execs}
assume vpeq\text{-}s\-t:\( \forall v. \ \text{ifp}\ v\ u \land \neg \text{intermediary} \ v\ u \implies vpeq \ v\ s\ t'\)
assume current\-s\-t:\ current \( s = \text{current} \ t'\)
assume purged-a\(\ a\text{-}a2\): ipurged-relation\(1\) u execs execs2
from ifp\-reflexive vpeq\text{-}s\-t have vpeq\(\text{-}u\text{-}s\-t\): \( vpeq\ u\ s\ t'\)
unfolding intermediary\-def by auto
from \text{step\-atomicity} \text{ and current\-s\-t} \text{ current\-next-state}
have current\-ns\-nt: \text{current} \ (\text{step} \ (\text{next-state} \ s \ \text{execs}) \ (\text{next-action} \ s \ \text{execs})) = \text{current} \ (\text{step} \ (\text{next-state} \ t')  

execs2) (next-action t' execs2))
unfolding step-def
  by (cases next-action s execs,cases next-action t' execs2,simp,simp,cases next-action t' execs2,simp,simp)

from step-atomicity current-next-state current-s-t have current-ns-t: current (step (next-state s execs) (next-action s execs)) = current t'
unfolding step-def
  by (cases next-action s execs,auto)
from vpeq-s-t have vpeq-curr-s-t: ifp (current s) u ∧ ¬intermediary (current s) u → vpeq (current s) s t'
unfolding intermediary-def by auto
from current-s-t purged-a-a2
  have eq-execs ifp (current s) u ∧ ¬intermediary (current s) u → execs (current s) = execs2 (current s)
  by(auto simp add: ipurged-relation1-def)
from vpeq-involved-domains no-gateway-comm na vpeq-s-t vpeq-involved-domains prec-s
  have vpeq-involved: ifp (current s) u ∧ ¬intermediary (current s) u → (∀ d ∈ involved (next-action s execs) . vpeq d s t')
  by blast
from current-s-t next-execs-consistent[THEN spec,THEN spec,THEN spec,where x2=s and x1=t' and x=execs]
vpeq-curr-s-t vpeq-involved
  have next-execs-t: ifp (current s) u ∧ ¬intermediary (current s) u → next-execs t' execs = next-execs s execs
  by(auto simp add: next-execs-def)
from current-s-t and purged-a-a2 and thread-not-empty-s next-action-consistent[THEN spec,THEN spec,where x1=s and x=t']
vpeq-s-t vpeq-involved
  have next-action-s-t: ifp (current s) u ∧ ¬intermediary (current s) u → next-action t' execs2 = next-action s execs
  by(unfold next-action-def,unfold ipurged-relation1-def,auto)
from purged-a-a2 and thread-not-empty-s and current-s-t
  have thread-not-empty-t: ifp (current s) u ∧ ¬intermediary (current s) u → ¬thread-empty(execs2 (current t'))
unfolding ipurged-relation1-def by auto
have vpeq-ns-nt-1: a . precondition (next-state s execs) a ∧ precondition (next-state t' execs) a → ifp (current s) u ∧ ¬intermediary (current s) u →ausal (next-state s execs) s (step (next-state t' execs) a)
proof-
  fix a
  assume precs: precondition (next-state s execs) a ∧ precondition (next-state t' execs) a
  assume ifp-curr: ifp (current s) u ∧ ¬intermediary (current s) u
  from ifp-curr precs
  next-state-consistent[THEN spec,THEN spec,where x1=s and x=t']
vpeq-curr-s-t vpeq-s-t
  current-state-consistent[THEN spec,THEN spec,THEN spec,THEN spec,THEN spec,THEN spec,THEN spec,where x3=next-state s execs and x2=next-state t' execs and x=the a]
  show ∀ v . ifp v u ∧ ¬intermediary v u → vpeq v (step (next-state s execs) s (step (next-state t' execs) a))
  unfolding step-def precondition-def B-def
  by (cases a,auto)
qed

have no-gateway-comm-na: does-not-communicate-with-gateway u (next-execs s execs)
proof-
  { fix a
    assume a ∈ actions-in-execution (next-execs s execs u)
    from this no-gateway-comm[unfolded does-not-communicate-with-gateway-def,THEN spec,where x=a]
    next-execs-subset[THEN spec,THEN spec,THEN spec,where x2=s and x1=execs and x0=u]
    have ∃ v. intermediary v u → v ∈ involved (Some a)
    unfolding actions-in-execution-def
    by(auto)
  } thus ?thesis unfolding does-not-communicate-with-gateway-def by auto
have iequivalent-states \( \text{run} (\text{Suc } n) (\text{Some } s) \text{ execs} \) (\( \text{run} (\text{Suc } n) (\text{Some } t') \text{ execs2} \)) \( u \)
proof (cases ifp (current s) \( u \land \neg \text{intermediary} \) (current s) \( u \) rule :\text{case-split} [\text{case-names } T \ F])

case \( T \)
show \(?\text{thesis}\)
proof (cases thread-empty (execs2 (current t')) rule :\text{case-split} [\text{case-names } T2 \ F2])
case \( F2 \)
show \(?\text{thesis}\)
proof (cases precondition (next-state t' execs2) (next-action t' execs2) rule :\text{case-split} [\text{case-names } T3 \ F3])

case \( T3 \)
from \( T \) purged-a-a2 current-s-t
next-execs-consistent[THEN spec,THEN spec,where \( x1=s \) and \( x=t' \)] vpeq-curr-s-t vpeq-involved
have purged-na-na2 : ipurged-relation1 \( u \) (next-execs s execs) (next-execs t' execs2)
unfolding ipurged-relation1-def next-execs-def
by auto
from IH [where \( t=\text{Some} \) (step (next-state t' execs2) (next-action t' execs2)) and ?execs2.0=next-execs t' execs2 and \( u=\)]
purged-na-na2 current-ns-nt vpeq-ns-nt-1 [where \( a=(\text{next-action s execs}) \) ] T T spec-s
next-action-s-t eq-execs current-s-t no-gateway-comm-na
have eq-ns-nt : iequivalent-states \( \text{run} n \) (\( Some \) (step (next-state s execs) (next-action s execs))) (next-execs s execs)

\( \text{run} n \) (\( Some \) (step (next-state t' execs2) (next-action t' execs2))) (next-execs t' execs2)) \( u \)
unfolding next-state-def
by (auto,metis)
from this not-interrupt thread-not-empty-s prec-s F2 T3
have current-rs-rt : current rs = current rt using rs rt by auto
{}
fix \( v \)
assume ia : ifp \( v u \land \neg \text{intermediary} \) \( v u \)
from this eq-ns-nt not-interrupt thread-not-empty-s prec-s F2 T3
have vpeq v rs rt using rs rt by auto
}
from this and current-rs-rt show \(?\text{thesis}\) using rs rt by auto
next


case \( F3 \)
from \( F3 \) F2 not-interrupt show \(?\text{thesis}\) using rt by simp
qed


case \( T2 \)
from \( T2 \) T purged-a-a2 thread-not-empty-s current-s-t prec-s next-action-s-t vpeq-u-s-t
have ind-source : False unfolding ipurged-relation1-def by auto
thus \(?\text{thesis}\) by auto
qed


case \( F \)
hence 1 : ind-source \( \text{current s} \) \( u \lor \text{unrelated} \) \( \text{current s} \) \( u \lor \text{intermediary} \) \( \text{current s} \) \( u \)
unfolding intermediary-def
by auto
from purged-a-a2 and thread-not-empty-s
have 2 : \( \neg \text{intermediary} \) \( \text{current s} \) \( u \) unfolding ipurged-relation1-def by auto

let \(?nt = \text{if thread-empty} (\text{execs2} \ (\text{current t'})) \) then t' else \( \text{step} \) (next-state t' execs2) (next-action t' execs2)
let \(?na2 = \text{if thread-empty} (\text{execs2} \ (\text{current t'})) \) then execs2 else next-execs t' execs2

have prec-t : \( \neg \text{thread-empty} (\text{execs2} \ (\text{current t'})) \) \( \Rightarrow \) precondition (next-state t' execs2) (next-action t' execs2)
D31.1 – Formal specification of a generic MILS separation kernel

execs2) proof
  assume thread-not-empty-t: ¬thread-empty(execs2 (current t'))
  { assume not-prec-t: ¬precondition (next-state t' execs2) (next-action t' execs2)
    hence ran (Suc n) (Some t') execs2 = None using not-interrupt thread-not-empty-t not-prec-t by (simp)
    from this have False using rt by (simp add: option.splits)
  }
  thus ?thesis by auto
qed

show ?thesis
proof
{
  fix v
  assume ifp-v: ifp v u
  assume v-not-intermediary: ¬intermediary v u

  have not-ifp-curr-v: ¬ifp (current s) v
    proof
      assume ifp-curr-v: ifp (current s) v
      thus False
      proof
        { assume ind-source (current s) u
          from this ifp-curr-v ifp-v have intermediary v u unfolding intermediary-def by auto
          from this v-not-intermediary have False unfolding intermediary-def by auto
        }
        moreover
        { assume unrelated: unrelated (current s) u
          from this ifp-v ifp-curr-v have False using rtranclp-trans r-into-rtranclp by metis
        }
        ultimately show ?thesis using 1 2 by auto
      qed
    qed
  from this current-next-state[THEN spec,THEN spec,where x1=s and x=execs] prec-s
  locally-respects[THEN spec,THEN spec,where x=next-state s execs] vpeq-reflexive
  have vpeq v (next-state s execs) (step (next-state s execs) (next-action s execs))
  unfolding step-def precondition-def B-def
  by (cases next-action s execs,auto)
  from not-ifp-curr-v this locally-respects-next-state vpeq-transitive
  have vpeq-s-ns: vpeq v s (step (next-state s execs) (next-action s execs))
  by blast
  from not-ifp-curr-v current-s-t current-next-state[THEN spec,THEN spec,where x1=t' and x=execs2] prec-t
  locally-respects[THEN spec,THEN spec,where x=next-state t' execs2]
  F vpeq-reflexive
  have 0: ¬thread-empty (execs2 (current t')) → vpeq v (next-state t' execs2) (step (next-state t' execs2)
  (next-action t' execs2))
  unfolding step-def precondition-def B-def
  by (cases next-action t' execs2,auto)
  from 0 not-ifp-curr-v current-s-t locally-respects-next-state[THEN spec,THEN spec,THEN spec,where
  x2=t' and x1=v and x=execs2]
  vpeq-transitive
  have vpeq-t-nt: ¬ thread-empty (execs2 (current t')) → vpeq v t' (step (next-state t' execs2) (next-action
  t' execs2)) by metis
  from this vpeq-reflexive
have vpeq-t-nt: vpeq v t' ?nt by auto
from vpeq-s-t ifp-v v-not-intermediary
have vpeq v s t' by auto
from this vpeq-s-ns vpeq-t-nt vpeq-transitive vpeq-symmetric vpeq-reflexive
have vpeq v (step (next-state s execs) (next-action s execs)) ?nt
by (metis (hide-lams, no-types))
}
hence vpeq-ns-nt: ∀ v. ifp v u ∧ ¬intermediary v u → vpeq v (step (next-state s execs) (next-action s execs)) ?nt
by auto
from vpeq-s-t 2 F purged-a-a2 current-s-t thread-not-empty-s
have purged-na-na2: ipurged-relation1 u (next-execs s execs) ?na2 unfolding ipurged-relation1-def next-execs-def intermediary-def
by (auto)
from current-ns-nt current-ns-t current-next-state
have current-ns-nt: current (step (next-state s execs) (next-action s execs)) = current ?nt
by auto
from prec-s vpeq-ns-nt no-gateway-comm-na and IH[where t=Some ?nt and ?execs2.0=?na2 and u=u]
and current-ns-nt purged-na-na2
have eq-ns-nt: iequivalent-states (run n (Some (step (next-state s execs) (next-action s execs)))) (next-execs s execs)
(run n (Some ?nt) ?na2) u by auto
from this not-interrupt thread-not-empty-s prec-t prec-s
have current-rs-rt: current rs = current rt using rs rt
by (cases thread-empty (execs2 (current t'))) simp simp
{
  fix v
  assume ia: ifp v u ∧ ¬intermediary v u
  from this eq-ns-nt not-interrupt thread-not-empty-s prec-s prec-t
  have vpeq v rs rt
    using rs rt
    by (cases thread-empty(execs2 (current t')), simp simp)
}
from current-rs-rt and this show: ?thesis using rs rt by auto
qed
qed
}
thus ?case by(simp add: option.splits,cases t,simp+)
qed
}
hence iview-partitioned-inductive: ∀ u s t execs execs2 n. does-not-communicate-with-gateway u execs ∧ iequivalent-states s t u ∧ ipurged-relation1 u execs execs2 → iequivalent-states (run n s execs) (run n t execs2) u
by blast
have ipurged-relation: ∀ u execs . ipurged-relation1 u (ipurge-l execs u) (ipurge-r execs u)
by (unfold ipurged-relation1-def, unfold ipurge-l-def, unfold ipurge-r-def, auto)
{
  fix execs s t n u
  assume 1: iequivalent-states s t u
  from ifp-reflexive
  have dir-source: ∀ u . ifp u u ∧ ¬intermediary u u unfolding intermediary-def
    by auto
  from ipurge-l-removes-gateway-communications
  have does-not-communicate-with-gateway u (ipurge-l execs u)
    by auto
  from 1 this inview-partitioned-inductive ipurged-relation
  have iequivalent-states (run n s (ipurge-l execs u)) (run n t (ipurge-r execs u)) u by auto
  from this dir-source
  have run n s (ipurge-l execs u) ∥ run n t (ipurge-r execs u) → (λrs rt. vpeq u rs rt ∧ current rs = current rt)
using r-into-rtrancl unfolding B-def
by(cases run n s (ipurge-l execs u),simp,cases run n t (ipurge-r execs u),simp,auto)
}
thus ?thesis unfolding iview-partitioned-def Let-def by auto
qed

Returns True iff and only if the two states have the same active domain, or if one of the states is None.

**definition** mcurrents :: 'state-t option ⇒ 'state-t option ⇒ bool
where mcurrents m1 m2 ≡ m1 | m2 → (∀ s t . current s = current t)

Proof that switching/interrupts are purely time-based and happen independent of the actions done by the domains. As all theorems in this locale, it holds vacuously whenever one of the states is None, i.e., whenever at some point a precondition does not hold.

**lemma** current-independent-of-domain-actions:
**assumes** current-s-t: mcurrents s t
**shows** mcurrents (run n s execs) (run n t execs2)
**proof**
{}
fix n s execs t execs2
have mcurrents s t → mcurrents (run n s execs) (run n t execs2)
proof
(induct n s execs arbitrary: t execs2 rule: run.induct)
case (Suc n execs t execs2)
from this show ?case using current-s-t unfolding B-def by auto
next
case (2 n execs t execs2)
show ?case unfolding mcurrents-def by(auto)
next
case (3 n s execs t execs2)
assume interrupt: interrupt (Suc n)
assume IH: (∀ t execs2 . mcurrents (Some (cswitch (Suc n) s)) t → mcurrents (run n (Some (cswitch (Suc n) s)) t) execs) (run n t execs2))
{}
fix t'
assume t: t = (Some t')
assume curr: mcurrents (Some s) t
from t curr cswitch-independent-of-state[THEN spec,THEN spec,THEN spec,where x1=s] have current-ns-nt: current (cswitch (Suc n) s) = current (cswitch (Suc n) t')
unfolding mcurrents-def by simp
from current-ns-nt IH[where t:=Some (cswitch (Suc n) t') and ?execs2.0=execs2]
have mcurrents-ns-nt: mcurrents (run n (Some (cswitch (Suc n) s)) execs) (run n (Some (cswitch (Suc n) t'))) (run n t execs2)
unfolding mcurrents-def by(auto)
from mcurrents-ns-nt interrupt t
have mcurrents (run (Suc n) (Some s) execs) (run (Suc n) t execs2)
unfolding mcurrents-def B2-def B-def by(cases run n (Some (cswitch (Suc n) s)) execs, cases run (Suc n) t execs2,auto)
}
thus ?case unfolding mcurrents-def B2-def by(cases t,auto)
next
case (4 n execs s t execs2)
assume not-interrupt: ¬interrupt (Suc n)
assume thread-empty-s: thread-empty (execs (current s))
assume IH: (∀ t execs2 . mcurrents (Some s) t → mcurrents (run n (Some s) execs) (run n t execs2))
{}
fix t'
assume t: t = (Some t')
assume \( \text{curr}: \text{mcurrents} \ (\text{Some } s) \ t \)
{
assume \( \text{thread-empty-t}: \text{thread-empty}(\text{execs2} \ (\text{current } t')) \)
from \( t \ \text{curr} \not= \text{interrupt} \) \( \text{thread-empty-s} \) this \( \text{IH} \)\( \begin{cases} \text{where} \ \ ?\text{execs2.0} = \text{execs2} \ \text{and} \ t = \text{Some } t' \end{cases} \)
\(
\text{have} \ \text{mcurrents} \ (\text{run } (\text{Suc } n) \ (\text{Some } s) \ \text{execs}) \ (\text{run } (\text{Suc } n) \ t \ \text{execs2}) \\
\text{by } \text{auto}
\)
moreover
{
\assume \( \text{not-prec-t}: \not= \text{thread-empty}(\text{execs2} \ (\text{current } t')) \land \not= \text{precondition} \ (\text{next-state } t' \ \text{execs2}) \ (\text{next-action } t' \ \text{execs2}) \)
from \( t \) this \( \not= \text{interrupt} \)
\(
\text{have} \ \text{mcurrents} \ (\text{run } (\text{Suc } n) \ (\text{Some } s) \ \text{execs}) \ (\text{run } (\text{Suc } n) \ t \ \text{execs2}) \\
\text{unfolding} \ \text{mcurrents-def} \ \text{by} \ (\text{simp add: rewrite-B2-cases})
\)
moreover
{
\assume \( \text{step-t}: \not= \text{thread-empty}(\text{execs2} \ (\text{current } t')) \land \text{precondition} \ (\text{next-state } t' \ \text{execs2}) \ (\text{next-action } t' \ \text{execs2}) \)
\\( \text{have} \ \text{mcurrents} \ (\text{Some } s) \ (\text{Some } (\text{step } (\text{next-state } t' \ \text{execs2}) (\text{next-action } t' \ \text{execs2}))) \)
\\( \text{using} \ \text{step-atomicity} \ \text{curr } t \ \text{current-next-state} \ \text{unfolding} \ \text{mcurrents-def} \)
\\( \text{unfolding} \ \text{step-def} \)
\\( \text{by} \ (\text{cases next-action } t' \ \text{execs2,auto}) \)
from \( t \) \( \text{step-t} \not= \text{interrupt} \) \( \text{thread-empty-s} \) this \( \text{IH} \)\( \begin{cases} \text{where} \ ?\text{execs2.0} = \text{next-execs } t' \ \text{execs2} \ \text{and} \ t = \text{Some } (\text{step } (\text{next-state } t' \ \text{execs2}) (\text{next-action } t' \ \text{execs2}))) \)
\\( \text{have} \ \text{mcurrents} \ (\text{run } (\text{Suc } n) \ (\text{Some } s) \ \text{execs}) \ (\text{run } (\text{Suc } n) \ t \ \text{execs2}) \\
\text{by } \text{auto}
\)
ultimately have \( \text{mcurrents} \ (\text{run } (\text{Suc } n) \ (\text{Some } s) \ \text{execs}) \ (\text{run } (\text{Suc } n) \ t \ \text{execs2}) \) \text{by blast}
}
thus \( ?\text{case unfolding } \text{mcurrents-def } \text{B2-def} \ \text{by}(\text{cases } t,\text{auto}) \)
next
case \( (5 \ \text{n execs } s \ \text{t execs2}) \)
\( \text{assume} \ \text{not-interrupt-s}: \not= \text{interrupt} \ (\text{Suc } n) \)
\( \text{assume} \ \text{thread-not-empty-s}: \not= \text{thread-empty}(\text{execs} \ (\text{current } s)) \)
\( \text{assume} \ \text{not-prec-s}: \not= \text{precondition} \ (\text{next-state } s \ \text{execs}) \ (\text{next-action } s \ \text{execs}) \)
\( \text{hence} \ \text{run} \ (\text{Suc } n) \ (\text{Some } s) \ \text{execs} = \text{None} \) \text{using} \ \text{not-interrupt-s} \ \text{thread-not-empty-s} \ \text{by } \text{simp} \)
thus \( ?\text{case unfolding } \text{mcurrents-def } \text{by}(\text{simp add:option.splits}) \)
next
case \( (6 \ \text{n execs } s \ \text{t execs2}) \)
\( \text{assume} \ \text{not-interrupt}: \not= \text{interrupt} \ (\text{Suc } n) \)
\( \text{assume} \ \text{thread-not-empty-s}: \not= \text{thread-empty}(\text{execs} \ (\text{current } s)) \)
\( \text{assume} \ \text{prec-s}: \text{precondition} \ (\text{next-state } s \ \text{execs}) \ (\text{next-action } s \ \text{execs}) \)
\( \text{assume} \ \text{IH}: (\land t \ \text{execs2}, \) \text{mcurrents} \ (\text{Some } (\text{step } (\text{next-state } s \ \text{execs}) \ (\text{next-action } s \ \text{execs}))) \ t \rightarrow \ ) \text{mcurrents} \ (\text{run } n \ (\text{Some } (\text{step } (\text{next-state } s \ \text{execs}) \ (\text{next-action } s \ \text{execs}))) \ (\text{next-execs } s \ \text{execs}) \) \ (\text{run } n \ t \ \text{execs2})) \)
{
fix \( t' \)
\( \text{assume} \ t : t = (\text{Some } t') \)
\( \text{assume} \ \text{curr}: \text{mcurrents} \ (\text{Some } s) \ t \)
{
\( \text{assume} \ \text{thread-empty-t}: \text{thread-empty}(\text{execs2} \ (\text{current } t')) \)
\( \text{have} \ \text{mcurrents} \ (\text{Some } (\text{step } (\text{next-state } s \ \text{execs}) \ (\text{next-action } s \ \text{execs}))) \ (\text{Some } t') \)
\( \text{using} \ \text{step-atomicity} \ \text{curr } t \ \text{current-next-state} \ \text{unfolding} \ \text{mcurrents-def} \)
\( \text{unfolding} \ \text{step-def} \)
\( \text{by} \ (\text{cases next-action } s \ \text{execs,auto}) \)
from t curr not-interrupt thread-not-empty-s prec-s thread-empty-t this IH

\[ \text{where } \exists t \text{ where } \exists s \text{ execs}_{2.0} = \text{execs}_{2} \text{ and } \]

\[ \exists t = \text{Some } t' \]

have mcurrents (run (Suc n) (Some s) execs) (run (Suc n) t execs)
by auto
\]

moreover
\{ 
assume not-prec-t : ¬ thread-empty(execs_{2}(current t')) ∧ ¬ precondition(next-state t' execs_{2}) (next-action t' execs_{2})
from t this not-interrupt
have mcurrents (run (Suc n) (Some s) execs) (run (Suc n) t execs)
unfolding mcurrents-def B2-def by (auto)
\}

moreover
\{ 
assume step-t : ¬ thread-empty(execs_{2}(current t')) ∧ precondition(next-state t' execs_{2}) (next-action t' execs_{2})
have mcurrents (Some (step (next-state s execs) (next-action s execs))) (Some (step (next-state t' execs_{2}) (next-action t' execs_{2})))
using step-atomicity curr t current-next-state unfolding mcurrents-def
unfolding step-def
by (cases next-action s execs, simp, cases next-action t' execs_{2}, simp, simp, cases next-action t' execs_{2}, simp, simp)
from current-next-state t step-t curr not-interrupt thread-not-empty-s prec-s this IH
\[ \text{where } \exists t \text{ execs}_{2} \text{ and } t = \text{Some } (\text{step (next-state t' execs}_{2}) (next-action t' execs_{2})) \]

have mcurrents (run (Suc n) (Some s) execs) (run (Suc n) t execs)
by auto
\}

ultimately have mcurrents (run (Suc n) (Some s) execs) (run (Suc n) t execs) by blast
\}

thus ?case unfolding mcurrents-def B2-def by (cases t, auto)
qed
\}

thus ?thesis using current-s-t by auto
qed

theorem unwinding-implies-NI-indirect-sources:
shows NI-indirect-sources
proof-
\{ 
fix execs a n
from assms unwinding-implies-view-partitioned1
have vp : iview-partitioned by blast
from vp and vpeq-reflexive
have I : ∀ u . run n (Some s0) (ipurge-l execs u) ∥ run n (Some s0) (ipurge-r execs u) → (λrs rt. vpeq u rs rt ∧ current rs = current rt)
unfolding iview-partitioned-def by auto

have run n (Some s0) execs → (λs-f . run n (Some s0) (ipurge-l execs (current s-f))) ∥
run n (Some s0) (ipurge-r execs (current s-f)) → (λs-l s-r. output-f s-l a = output-f s-r a))
proof(cases run n (Some s0) execs)
case None 
thus ?thesis unfolding B-def by simp
next
case (Some s-f)
thus ?thesis
proof(cases run n (Some s0) (ipurge-l execs (current s-f)))
case None
   from Some this show thesis unfolding B-def by simp
next
case (Some s-ipurge-l)
   show thesis
   proof (cases run n (Some s0) (ipurge-r execs (current s-f)))
   case None
   from \run n (Some s0) execs = Some s-f\ Some this show thesis unfolding B-def by simp
next
case (Some s-ipurge-r)
   from cswitch-independent-of-state
   \run n (Some s0) execs = Some s-f\ \run n (Some s0) (ipurge-l execs (current s-f)) = Some s-ipurge-b 
\current-independent-of-domain-actions[\where\ n=n\ and\ s=Some s0\ and\ t=Some s0\ and\ execs=execs\ and\ ?execs2.0=(ipurge-l execs (current s-f))]
   have 2:: current s-ipurge-l = current s-f
   unfolding mcurrents-def B-def by auto
   from \run n (Some s0) execs = Some s-f\ \run n (Some s0) (ipurge-l execs (current s-f)) = Some s-ipurge-b 
\some I THEN spec,where x=current s-f\]
   have vpeq (current s-f) s-ipurge-l s-ipurge-r \ current s-ipurge-l = current s-ipurge-r 
unfolding B-def by auto
   from this 2 have output-f s-ipurge-l a = output-f s-ipurge-r a
   using output-consistent by auto
   from \run n (Some s0) execs = Some s-f\ \run n (Some s0) (ipurge-l execs (current s-f)) = Some s-ipurge-b 
this Some
   show thesis unfolding B-def by auto
   qed
   qed
   qed
}
thus thesis unfolding NI-indirect-sources-def by auto
qed

theorem unwinding-implies-isecure:
shows isecure
using unwinding-implies-NI-indirect-sources unwinding-implies-NI-unrelated assms unfolding isecure-def by(auto)
end
end

2.3 ISK (Interruptible Separation Kernel)

theory ISK
  imports SK
begin

At this point, the precondition linking action to state is generic and highly unconstrained. We refine the previous locale by given generic functions “precondition” and “realistic_trace” a definiton. This yields a total run function, instead of the partial one of locale Separation_Kernel.

This definition is based on a set of valid action sequences AS_set. Consider for example the following action sequence:

\[ \gamma = [COPY\ INIT, COPY\ CHECK, COPY\ COPY] \]

If action sequence \( \gamma \) is a member of AS_set, this means that the attack surface contains an action COPY, which consists of three consecutive atomic kernel actions. Interrupts can occur anywhere between these atomic actions.
Given a set of valid action sequences such as $\gamma$, generic function precondition can be defined. It now consists of 1.) a generic invariant and 2.) more refined preconditions for the current action.

These preconditions need to be proven inductive only according to action sequences. Assume, e.g., that $\gamma \in \text{AS}_\text{set}$ and that $d$ is the currently active domain in state $s$. The following constraints are assumed and must therefore be proven for the instantiation:

- “AS-precondition s d COPY_INIT” since COPY_INIT is the start of an action sequence.
- “AS-precondition (step s COPY_INIT) d COPY_CHECK” since (COPY_INIT, COPY_CHECK) is a sub sequence.
- “AS-precondition (step s COPY_CHECK) d COPY_COPY” since (COPY_CHECK, COPY_COPY) is a sub sequence.

Additionally, the precondition for domain $d$ must be consistent when a context switch occurs, or when ever some other domain $d'$ performs an action.

Locale Interruptible_Separation_Kernel refines locale Separation_Kernel in two ways. First, there is a definition of realistic executions. A realistic trace consists of action sequences from AS_set.

Secondly, the generic control function has been refined by additional assumptions. It is now assumed that control conforms to one of four possibilities:

1. The execution of the currently active domain is empty and the control function returns no action.
2. The currently active domain is executing the action sequence at the head of the execution. It returns the next kernel action of this sequence and updates the execution accordingly.
3. The action sequence is delayed.
4. The action sequence that is at the head of the execution is skipped and the execution is updated accordingly.

As for the state update, this is still completely unconstrained and generic as long as it respects the generic invariant and the precondition.
and invariant-after-cswitch: \( \forall s n . \) invariant \( s \rightarrow\) invariant \( (\text{cswitch } n \ s) \)

and precondition-after-cswitch: \( \forall s d n a. \) AS-precondition \( s \rightarrow\) AS-precondition \( (\text{cswitch } n \ s) \ d a \)

and AS-prec-first-action: \( \forall s d aseq . \) invariant \( s \land aseq \in \text{AS-set} \land \text{aseq} \neq [] \rightarrow \) AS-precondition \( s \rightarrow\) \( d \) (hd aseq)

and AS-prec-after-step: \( \forall s a a'. (\exists \text{ aseq } \in \text{AS-set} . \text{is-sub-seq} a a' \text{ aseq} ) \land \) invariant \( s \land\) AS-precondition \( s \) \( a' \land \) aborting \( s \) \( (\text{current} s) \) \( a \land\) waiting \( s \) \( (\text{current} s) \) \( d \) \( \rightarrow\) AS-precondition \( (\text{kstep } s \ a) \) (current) \( s \) \( a' \)

and AS-prec-dom-independent: \( \forall s d a a'. \) current \( s \neq d \land\) AS-precondition \( s \rightarrow\) AS-precondition \( (\text{kstep } s \ a') \ d a \)

and spec-of-invariant: \( \forall s a . \) invariant \( s \rightarrow\) invariant \( (\text{kstep } s \ a) \)

and kprecondition-def: kprecondition \( s a \equiv\) invariant \( s \land\) AS-precondition \( s \) (current) \( s \) \( a \)

and realistic-execution-def: realistic-execution \( aseq \equiv\) set \( aseq \subseteq\) AS-set

and control-spec: \( \forall s d aseqs .\) case control \( s \rightarrow\) case control \( d \) aseqs of \( (a,aseqs',s') \Rightarrow\)

\((\text{thread-empty } aseqs \land (a,aseqs') = (\text{None},[])) \lor (\ast\text{ Nothing happens }\ast)\)

\((\text{aseqs} \neq [] \land \text{hd } aseqs \neq [] \land \text{aborting } s' \land d \land\) waiting \( s' \land d \land\) (the a) \land\) aborting \( s' d \land (\text{the } a) \land (a,aseqs') = (\text{Some } (\text{hd } (\text{hd } aseqs)), (\text{tl } (\text{hd } aseqs)) \# (\text{tl } aseqs)) \lor (\ast\text{ Execute the first action of the current action sequence }\ast)\)

\((\text{aseqs} \neq [] \land \text{hd } aseqs \neq [] \land\) waiting \( s' d \land (\text{the } a) \land (a,aseqs',s') = (\text{Some } (\text{hd } (\text{hd } aseqs)), (\text{aset } aseqs)) \lor (\ast\text{ Nothing happens, waiting to execute the next action }\ast)\)

\((a,aseqs') = (\text{None},\text{tl } aseqs)\)

and next-action-after-cswitch: \( \forall s n d aseqs .\) \text{fst} \ (\text{control } (\text{cswitch } n \ s) \ d aseqs) = \text{fst} \ (\text{control } s \ d aseqs)

and next-action-after-next-state: \( \forall s d aseqs .\) \text{fstat} \ (\text{control } (\text{cswitch } n \ s) \ d aseqs) = \text{fstat} \ (\text{control } s \ d aseqs)

and next-action-after-step: \( \forall aseqs s d n aseqs .\) \text{current} \( s \neq d \rightarrow\) \text{fstat} \ (\text{control } (\text{cswitch } n \ s) \ d aseqs) = \text{fstat} \ (\text{control } d aseqs)

and spec-of-waiting: \( \forall s a .\) waiting \( s \land\) AS-precondition \( s \rightarrow\) \( \) kstep \( s a = s \)

begin
We can now formulate a total run function, since based on the new assumptions the case where the pre
condition does not hold, will never occur.

function run-total : \text{time-t} \rightarrow \text{state-t} \Rightarrow ('\text{dom-t} \Rightarrow '\text{action-t} \text{execution}) \Rightarrow '\text{state-t}

where run-total 0 \text{ s execs} = s
| interrupt (Suc n) \Rightarrow run-total (Suc n) \text{ s execs} = run-total n (\text{cswitch } (Suc n) s) \text{ execs}
| \text{~}\text{interrupt} (Suc n) \Rightarrow \text{thread-empty } (\text{execs } (\text{current } s)) \Rightarrow run-total (Suc n) \text{ s execs} = run-total n \text{ s execs}
| \text{~}\text{interrupt} (Suc n) \Rightarrow \text{thread-empty } (\text{execs } (\text{current } s)) \Rightarrow run-total (Suc n) \text{ s execs} = run-total n (\text{step } (\text{next-state } s \text{ execs}) (\text{next-action } s \text{ execs})) (\text{next-execs } s \text{ execs})

using not0-implies-Suc by (metis prod-cases3.auto)

termination by lexicographic-order

The major part of the proofs in this locale consist of proving that function run-total is equivalent to
function run, i.e., that the precondition does always hold. This assumes that the executions are realistic.
This means that the execution of each domain contains action sequences that are from AS_set. This
ensures, e.g., that a COPY_CHECK is always preceded by a COPY_INIT.

definition realistic-executions :: ('\text{dom-t} \Rightarrow '\text{action-t} \text{execution}) \Rightarrow \text{bool}

where realistic-executions \text{execs} \equiv \forall d .\) realistic-execution \( (\text{execs } d)\)

Lemma run_total_equal_run is proven by doing induction. It is however not inductive and can there-
fore not be proven directly: a realistic execution is not necessarily realistic after performing one ac-
tion. We generalize to do induction. Predicate realistic_executions_ind is the inductive version of realistic-
executions. All action sequences in the tail of the executions must be complete action sequences (i.e.,
they must be from \text{AS-set}). The first action sequence, however, is being executed and is therefore not
necessarily an action sequence from \text{AS-set}, but it is the last part of some action sequence from \text{AS-set}.

definition realistic-AS-partial :: 'action-t list \Rightarrow \text{bool}

where realistic-AS-partial \text{aseq} \equiv \exists n \text{ aseq}' .\) n \leq \text{length } \text{aseq}' \land \text{aseq}' \in \text{AS-set} \land \text{aseq} = \text{lastn } n \text{ aseq}'}

EURO-MILS D31.1 Page 36 of 90
definition realistic-executions-ind :: ('dom-t ⇒ 'action-t execution) ⇒ bool
where realistic-executions-ind execs ≡ ∀ d . (case execs d of [] ⇒ True | (aseq#aseqs) ⇒ realistic-AS-partial aseq ∧ set aseqs ⊆ AS-set)

We need to know that invariably, the precondition holds. As this precondition consists of 1.) a generic invariant and 2.) more refined preconditions for the current action, we have to know that these two are invariably true.

definition precondition-ind :: 'state-t ⇒ ('dom-t ⇒ 'action-t execution) ⇒ bool
where precondition-ind s execs ≡ invariant s ∧ (∀ d . fst(control s d (execs d)) ⇒ AS-precondition s d)

Proof that “execution is realistic” is inductive, i.e., assuming the current execution is realistic, the execution returned by the control mechanism is realistic.

lemma next-execution-is-realistic-partial:
assumes na-def: next-exec s execs d = aseq # aseqs
and d-is-curr: d = current s
and realistic: realistic-executions-ind execs
and thread-not-empty: ¬thread-empty(execs (current s))
shows realistic-AS-partial aseq ∧ set aseqs ⊆ AS-set
proof
let ?c = control s (current s) (execs (current s))
{| assume c-empty: let (a,aseqs',s') = ?c in
 (a,aseqs') = (None,[])
from na-def d-is-curr c-empty
have ?thesis
unfolding realistic-executions-ind-def next-execs-def by (auto)
|}
moreover
{| let ?ct= execs (current s)
let ?execs' = (tl (hd ?ct)) #(tl ?ct)
let ?a' = Some (hd (hd ?ct))
assume hd-thread-not-empty: hd (execs (current s)) ≠ []
assume c-executing: let (a,aseqs',s') = ?c in
 (a,aseqs') = (?a', ?execs')
from na-def c-executing d-is-curr
have as-defs: aseq = tl (hd ?ct) ∧ aseqs = tl ?ct
unfolding next-execs-def by (auto)
from realistic[unfolded realistic-executions-ind-def,THEN spec,where x=d] d-is-curr
have subset: set (tl ?execs') ⊆ AS-set
unfolding Let-def realistic-AS-partial-def
by (cases execs d,auto)
from d-is-curr thread-not-empty hd-thread-not-empty realistic[unfolded realistic-executions-ind-def,THEN spec,where x=d]
obtain n aseq' where n-aseq': n ≤ length aseq' ∧ aseq' ∈ AS-set ∧ hd ?ct = lastn n aseq'
unfolding realistic-AS-partial-def
by (cases execs d,auto)
from this hd-thread-not-empty have n > 0 unfolding lastn-def by(cases n,auto)
from this n-aseq' lastn-one-less[where n=n and x=aseq' and a=hd (hd ?ct) and y=tl (hd ?ct)] hd-thread-not-empty
have n - 1 ≤ length aseq' ∧ aseq' ∈ AS-set ∧ tl (hd ?ct) = lastn (n - 1) aseq'
by auto
from this as-defs subset have ?thesis
unfolding realistic-AS-partial-def
by auto
|}
moreover
{|
let \( ?\text{ct} = \text{execs} \) (current \( s \))
let \( ?\text{execs} ' = ?\text{ct} \)
let \( ?a' = \text{Some} \) (hd (hd ?\text{ct}))
assume c-waiting: let \((a, \text{aseqs}' , s') = ?c\) in
\((a, \text{aseqs}') = (?a' , ?\text{execs}')\)
from na-def c-waiting d-is-curr
have as-defs: aseq = hd ?\text{execs}' \land aseq = tl ?\text{execs}'
unfolding next-execs-def by (auto)
from realistic[unfolded realistic-executions-ind-def, THEN spec, where \( x=d \)] d-is-curr set-tl-is-subset[where \( x=?\text{execs}' \)]
have subset: set (tl ?\text{execs'}) \subseteq AS-set
unfolding Let-def realistic-AS-partial-def
by (cases execs d/auto)
from na-def c-waiting d-is-curr
have ?\text{execs}' \neq [] unfolding next-execs-def by auto
from realistic[unfolded realistic-executions-ind-def, THEN spec, where \( x=d \)] d-is-curr thread-not-empty
obtain n aseq' where witness: \( n \leq \text{length} aseq' \land aseq' \in AS-set \land \text{hd}(\text{execs } d) = \text{lastn} n aseq'\)
unfolding realistic-AS-partial-def by (cases execs d/auto)
from d-is-curr this subset as-defs have ?\text{thesis}
unfolding realistic-AS-partial-def
by auto

moreover
{
let \( ?\text{ct} = \text{execs} \) (current \( s \))
let \( ?\text{execs} ' = \text{tl} ?\text{ct} \)
let \( ?a' = \text{None} \)
assume c-aborting: let \((a, \text{aseqs}' , s') = ?c\) in
\((a, \text{aseqs}') = (?a' , ?\text{execs}')\)
from na-def c-aborting d-is-curr
have as-defs: aseq = hd ?\text{execs}' \land aseq = tl ?\text{execs}'
unfolding next-execs-def by (auto)
from realistic[unfolded realistic-executions-ind-def, THEN spec, where \( x=d \)] d-is-curr set-tl-is-subset[where \( x=?\text{execs}' \)]
have subset: set (tl ?\text{execs}') \subseteq AS-set
unfolding Let-def realistic-AS-partial-def
by (cases execs d/auto)
from na-def c-aborting d-is-curr
have ?\text{execs}' \neq [] unfolding next-execs-def by auto
from empty-in-AS-set this
realistic[unfolded realistic-executions-ind-def, THEN spec, where \( x=d \)] d-is-curr
have length (hd ?\text{execs}') \leq length (hd ?\text{execs}') \land (hd ?\text{execs}') \in AS-set \land \text{hd} ?\text{execs}' = \text{lastn} (\text{length} (hd ?\text{execs}')) (hd ?\text{execs}')
unfolding lastn-def
by (cases execs (current \( s \)),auto)
from this subset as-defs have ?\text{thesis}
unfolding realistic-AS-partial-def
by auto
}
ultimately
show ?\text{thesis}
using control-spec[THEN spec, THEN spec, THEN spec, where \( x2=s \) and \( x1=\text{current } s \) and \( x=\text{execs} \) (current \( s \))] d-is-curr thread-not-empty
by (auto simp add: Let-def)
qed

The lemma that proves that the total run function is equivalent to the partial run function, i.e., that in this refinement the case of the run function where the precondition is False will never occur.
lemma run-total-equals-run:
  assumes realistic-exec: realistic-executions execs
  and invariant: invariant s
  shows strict-equal (run n (Some s) execs) (run-total n s execs)
proof-
{ 
  fix n ms s execs
  have strict-equal ms s ∧ realistic-executions-ind execs ∧ precondition-ind s execs ⃞ strict-equal (run n ms execs) (run-total n s execs)
proof (induct n ms execs arbitrary: s rule: run.induct)
case (1 s execs sa)
  show ?case by auto
next
case (2 n execs s)
  show ?case unfolding strict-equal-def by auto
next
case (3 n s execs sa)
  assume interrupt: interrupt (Suc n)
  assume IH: (∧sa. strict-equal (Some (cswitch (Suc n) s)) sa ∧ realistic-executions-ind execs ∧ precondition-ind sa execs ⃞ strict-equal (run n (Some (cswitch (Suc n) s)) execs) (run-total n sa execs))
  
  { 
    assume equal-s-sa: strict-equal (Some s) sa
    assume realistic: realistic-executions-ind execs
    assume inv-sa: precondition-ind sa execs
    have inv-nsa: precondition-ind (cswitch (Suc n) sa) execs
    proof-
    { 
      fix d
      have fst (control (cswitch (Suc n) sa) d (execs d)) ⃞ AS-precondition (cswitch (Suc n) sa) d
        using next-action-after-cswitch inv-sa[unfolded precondition-ind-def, THEN conjunct2, THEN spec, where x=d]
        precondition-after-cswitch
        unfolding Let-def B-def precondition-ind-def
        by (cases fst (control (cswitch (Suc n) sa) d (execs d)), auto)
    } 
    thus ?thesis using inv-sa invariant-after-cswitch unfolding precondition-ind-def by auto
  qed
from equal-s-sa realistic inv-nsa inv-sa IH[where sa=cswitch (Suc n) sa]
  have equal-ns-nt: strict-equal (run n (Some (cswitch (Suc n) s)) execs) (run-total n (cswitch (Suc n) sa) execs)
unfolding strict-equal-def by(auto)
  }
from this interrupt show ?case by auto
next
case (4 n execs s sa)
  assume no-interrupt: ¬interrupt (Suc n)
  assume thread-empty: thread-empty(execs (current s))
  assume IH: (∧sa. strict-equal (Some s) sa ∧ realistic-executions-ind execs ∧ precondition-ind sa execs ⃞ strict-equal (run n (Some s) execs) (run-total n sa execs))
  have current-s-sa: strict-equal (Some s) sa ⃞ current s = current sa unfolding strict-equal-def by auto
  
  { 
    assume equal-s-sa: strict-equal (Some s) sa
    assume realistic: realistic-executions-ind execs
    assume inv-sa: precondition-ind sa execs
    from equal-s-sa realistic inv-sa IH[where sa=sa]
    have equal-ns-nt: strict-equal (run n (Some s) execs) (run-total n sa execs)
unfolding strict-equal-def by (auto)
}
from this current-s-sa thread-empty not-interrupt show ?case by auto
next
case (5 n execs s sa)
assume not-interrupt: ⊼-interrupt (Suc n)
assume thread-not-empty: ⊼-thread-empty (execs (current s))
assume not-prec: ⊼-precondition (next-state s execs) (next-action s execs)
— In locale ISK, the precondition can be proven to hold at all times. This case cannot happen, and we can prove False.
{
assume equal-s-sa: strict-equal (Some s) sa
assume realistic: realistic-executions-ind execs
assume inv-sa: precondition-ind sa execs
from equal-s-sa have s-sa: s = sa unfolded strict-equal-def by auto
from inv-sa have
next-action sa execs → AS-precondition sa (current sa)
unfolding precondition-ind-def B-def next-action-def
by (cases next-action sa execs, auto)
from this next-state-precondition
have next-action sa execs ↪ AS-precondition (next-state sa execs) (current sa)
unfolding precondition-ind-def B-def
by (cases next-action sa execs, auto)
from inv-sa this s-sa next-state-invariant current-next-state
have prec-s: precondition (next-state s execs) (next-action s execs)
unfolding precondition-ind-def kprecondition-def precondition-def B-def
by (cases next-action sa execs, auto)
from this not-prec have False by auto
}
thus ?case by auto
next
case (6 n execs s sa)
assume not-interrupt: ⊼-interrupt (Suc n)
assume thread-not-empty: ⊼-thread-empty (execs (current s))
assume prec: precondition (next-state s execs) (next-action s execs)
assume IH: (∀sa. strict-equal (Some (step (next-state s execs) (next-action s execs))) sa ∧
realistic-executions-ind (next-execs s execs) ∧ precondition-ind sa (next-execs s execs) →
strict-equal (run n (Some (step (next-state s execs) (next-action s execs))) (next-execs s execs)) (run-total
n sa (next-execs s execs)))
have current-s-sa: strict-equal (Some s) sa → current s = current sa unfolded strict-equal-def by auto
{
assume equal-s-sa: strict-equal (Some s) sa
assume realistic: realistic-executions-ind execs
assume inv-sa: precondition-ind sa execs
from equal-s-sa have s-sa: s = sa unfolded strict-equal-def by auto

let ?a = next-action s execs
let ?ns = step (next-state s execs) ?a
let ?na = next-execs s execs
let ?c = control s (current s) (execs (current s))

have equal-ns-nsa: strict-equal (Some ?ns) ?ns unfolded strict-equal-def by auto
from inv-sa equal-s-sa have inv-s: invariant s unfolded strict-equal-def precondition-ind-def by auto

— Two things are proven inductive. First, the assumptions that the execution is realistic (statement realistic-na).
This proof uses lemma next-execution-is-realistic-partial. Secondly, the precondition: if the precondition holds for
the current action, then it holds for the next action (statement invariant-na).

```

have realistic-na: realistic-executions-ind ?na
proof
{
  fix d
  have case ?na d of [] ⇒ True | aseq ≠ aseqs ⇒ realistic-AS-partial aseq ∧ set aseqs ⊆ AS-set proof(cases ?na d,simp,rename-tac aseq aseqs,simp,cases d = current s)
  case False
    fix aseq aseqs
    assume next-execs s execs d = aseq ≠ aseqs
    from False this realistic[unfolded realistic-executions-ind-def \THEN spec,where x=d]
    show realistic-AS-partial aseq ∧ set aseqs ⊆ AS-set unfolding next-execs-def by simp
  next
    case True
    fix aseq aseqs
    assume na-def: next-execs s execs d = aseq ≠ aseqs
    from next-execution-is-realistic-partial na-def True realistic thread-not-empty
    show realistic-AS-partial aseq ∧ set aseqs ⊆ AS-set by blast
  qed
}
thus ?thesis unfolding realistic-executions-ind-def by auto
qed
have invariant-na: precondition-ind ?ns ?na
proof
  from spec-of-invariant inv-sa next-state-invariant s-sa have inv-ns: invariant ?ns unfolding precondition-ind-def step-def by (cases next-action sa execs,auto)
  have ∀ d. fst (control ?ns d (?na d)) ⇒ AS-precondition ?ns d proof
  {
    fix d
    {
      let ?a′ = fst (control ?ns d (?na d))
      assume snd-action-not-none: ?a′ ≠ None
      have AS-precondition ?ns d (the ?a′) proof (cases d = current s)
      case True
      {
        have ?thesis proof (cases ?a)
        case (Some a)
        — Assuming that the current domain executes some action a, and assuming that the action a′ after that is not None (statement snd-action-not-none), we prove that the precondition is inductive, i.e., it will hold for a′. Two cases arise: either action a is delayed (case waiting) or not (case executing).
        show ?thesis proof (cases ?na d = execs (current s) rule:case-split[case-names waiting executing])
        case executing — The kernel is executing two consecutive actions a and a′. We show that [a,a′] is a subsequence in some action in AS-set. The PO’s ensure that the precondition is inductive.
          from executing True Some control-spec[THEN spec,THEN spec,THEN spec,where x2=s and x1=d and x=execs d]
          have a-def: a = hd (execs (current s)) ∧ ?na d = (tl (hd (execs (current s)))) #(tl (execs (current s)))) unfolding next-action-def next-execs-def Let-def by(auto)
          from a-def True snd-action-not-none control-spec[THEN spec,THEN spec,THEN spec,where x2=?ns and x1=d and x=?na d]
```
second-elt-is-hd-tl[where x= hd (execs (current s)) and a=hd(tl(hd (execs (current s))))) and x'=tl
(tl(hd (execs (current s)))))

have na-def: the ?a' = (hd (execs (current s)))!1

unfolding next-execs-def

by(auto)

from Some realistic[unfolded realistic-executions-ind-def, THEN spec, where x=d] True thread-not-empty

obtain n aseq' where witness: n ≤ length aseq' ∧ aseq' ∈ AS-set ∧ hd(execs d) = lastn n aseq'

unfolding realistic-AS-partial-def by (cases execs d, auto)

from True executing length-lt-2-implies-tl-empty[where x=hd (execs (current s))]

Some control-spec[THEN spec, THEN spec, THEN spec, where x2=s and x1=d and x=execs d]

snd-action-not-none control-spec[THEN spec, THEN spec, THEN spec, where x2=?ns and x1=d and
x=?na d]

have in-action-sequence: length (hd (execs (current s))) ≥ 2

unfolding next-action-def next-execs-def

by auto

from this witness-sequence: length (hd (execs (current s))) ≥ 2

a-def na-def True in-action-sequence

x-is-hd-snd-tl[where x=hd (execs (current s))]

have 1: ∃ aseq' ∈ AS-set . is-sub-seq a (the ?a') aseq'

by(auto)

from True Some inv-sa[unfolded precondition-ind-def, THEN conjunct2, THEN spec, where x=current s] s-sa

have 2: AS-precondition s (current s) a

unfolding strict-equal-def next-action-def B-def by auto

from executing True Some control-spec[THEN spec , THEN spec , THEN spec , where x2=s and x1=d and
x=execs d]

have not-aborting: ¬aborting (next-state s execs) (current s) (the ?a)

unfolding next-action-def next-state-def next-execs-def

by(auto)

from executing True Some control-spec[THEN spec , THEN spec , THEN spec , where x2=s and x1=d and
x=execs d]

have not-waiting: ¬waiting (next-state s execs) (current s) (the ?a)

unfolding next-action-def next-state-def next-execs-def

by(auto)

from True this

1 2 inv-s

sub-seq-in-prefixes[where X=AS-set] Some next-state-invariant

current-next-state[ THEN spec, THEN spec, where x1=s and x=execs]

AS-prec-after-step[THEN spec, THEN spec, THEN spec, where x2=next-state s execs and x1=a and
x=the ?a']

next-state-precondition not-aborting not-waiting

show ?thesis

unfolding step-def

by(auto)

next case waiting — The kernel is delaying action a. Thus the action after a, which is a’, is equal to a.

from tl-hd-x-not-ntl-s[where x=execs d] True waiting control-spec[THEN spec , THEN spec , THEN spec , THEN
spec, where x2=s and x1=d and x=execs d]

Some

have a-def: ?na d = execs (current s) ∧ next-state s execs = s ∧ waiting s d (the ?a)

unfolding next-action-def next-execs-def next-state-def

by(auto)

from Some waiting a-def True snd-action-not-none control-spec[THEN spec , THEN spec , THEN
spec, where x2=?ns and x1=d and x=?na d]

have na-def: the ?a' = hd (hd (execs (current s)))

unfolding next-action-def next-execs-def

by(auto)
\[ \textbf{from \ spec-of-waiting \ a-def \ True} \]
\[ \textbf{have \ no-step: \ step \ s \ ?a = s \ unfolding \ step-def \ by \ (cases \ next-action \ s \ execs, \auto)} \]
\[ \textbf{from \ no-step \ Some \ True \ a-def} \]
\[ \text{inv-sa[unfolded \ precondition-ind-def, THEN conjunct2, THEN spec, where \ x = current \ s] s-sa} \]
\[ \text{have \ 2: \ AS-precondition \ s \ (current \ s) \ (the \ ?a')} \]
\[ \text{unfolding \ next-action-def \ B-def} \]
\[ \text{by (auto)} \]
\[ \textbf{from \ a-def \ na-def \ this \ True \ no-step} \]
\[ \text{show \ ?thesis} \]
\[ \text{unfolding \ step-def} \]
\[ \text{by (auto)} \]
\[ \textbf{qed} \]
\[ \text{next} \]
\[ \text{case \ None} \]

Assuming that the current domain does not execute an action, and assuming that the action a’ after that is not None (statement snd-action-not-none), we prove that the precondition is inductive, i.e., it will hold for a’. This holds, since the control mechanism will ensure that action a’ is the start of a new action sequence in AS-set.

\[ \textbf{from \ None \ True \ snd-action-not-none \ control-spec[THEN \ spec, THEN spec, THEN spec, where \ x2 = ?ns \ and \ x1 = d \ and \ x = ?na \ d]} \]
\[ \text{control-spec[THEN \ spec, THEN spec, THEN spec, where \ x2 = s \ and \ x1 = d \ and \ x = execs \ d]} \]
\[ \text{have \ na-def: \ the \ ?a' = hd \ (tl \ (execs \ (current \ s))) \ 
\text{unfolding \ next-action-def \ next-execs-def} \]
\[ \text{by (auto)} \]
\[ \textbf{from \ True \ None \ snd-action-not-none \ control-spec[THEN \ spec, THEN spec, THEN spec, where \ x2 = ?ns \ and \ x1 = d \ and \ x = ?na \ d]} \]
\[ \text{this} \]
\[ \text{have \ 1: \ tl \ (execs \ (current \ s)) \ \# \ [ ] \ \and \ hd \ (tl \ (execs \ (current \ s))) \ \# \ [ ]} \]
\[ \text{by \ auto} \]
\[ \text{from \ this \ realistic[unfolded \ realistic-executions-ind-def, THEN spec, where \ x = d] \ True \ thread-not-empty} \]
\[ \text{have \ hd \ (tl \ (execs \ (current \ s))) \ \in \ AS-set} \]
\[ \text{by \ (cases \ execs \ d, \auto)} \]
\[ \textbf{from \ True \ snd-action-not-none \ this} \]
\[ \text{inv-ns \ this \ na-def \ \in} \]
\[ \text{AS-prec-first-action[THEN \ spec, THEN spec, THEN spec, where \ x2 = ?ns \ and \ x = hd \ (tl \ (execs \ (current \ s))]} \]
\[ \text{and \ x1 = d]} \]
\[ \text{show \ ?thesis \ by \ auto} \]
\[ \textbf{qed} \]
\[ \text{thus \ ?thesis \ using \ control-spec[THEN \ spec, THEN spec, THEN spec, where \ x2 = ?ns \ and \ x1 = current \ s \ and \ x = ?na} \]
\[ \text{\ (current \ s)}]} \]
\[ \text{thread-not-empty \ True \ snd-action-not-none} \]
\[ \text{by \ (auto \ simp \ add: \ Let-def)} \]
\[ \textbf{next} \]
\[ \text{case \ False} \]
\[ \text{from \ False \ have \ equal-na-a: \ ?na \ d = execs \ d} \]
\[ \text{unfolding \ next-execs-def \ by \ auto} \]
\[ \text{from \ this \ False \ current-next-state \ next-action-after-step} \]
\[ \text{have \ ?a' = fst \ (control \ (next-state \ s \ execs) \ d \ (next-execs \ s \ execs \ d))} \]
\[ \text{unfolding \ next-action-def \ by \ auto} \]
\[ \text{from \ inv-sa[unfolded \ precondition-ind-def, THEN conjunct2, THEN spec, where \ x = d] \ equal-na-a \ this} \]
\[ \text{next-action-after-next-state[THEN \ spec, THEN spec, THEN spec, where \ x = d \ and \ x2 = s \ and \ x1 = execs]} \]
\[ \text{snd-action-not-none \ False} \]
\[ \text{have \ AS-precondition \ s \ d \ (the \ ?a')} \]
\[ \text{unfolding \ precondition-ind-def \ next-action-def \ B-def \ by \ (cases \ fst \ (control \ sa \ d \ (execs \ d)), \auto)} \]
\[ \textbf{from \ equal-na-a \ False \ this \ next-state-precondition \ current-next-state} \]

---

**EURO-MILS D31.1**

Page 43 of 90
AS-prec-dom-independent | THEN spec, THEN spec, THEN spec, THEN spec, where x3 = next-state s execs and x2 = d and x = the ?a and x1 = the ?a' |
show ?thesis unfolding step-def by (cases next-action s execs, auto) qed
hence fst (control ?ns d (?na d)) → AS-precondition ?ns d unfolding B-def by (cases fst (control ?ns d (?na d)), auto)
thus ?thesis by auto qed
from this inv-ns show ?thesis unfolding precondition-ind-def B-def Let-def by (auto)
hence fst (control ?ns d (?na d)) ↭ AS-precondition ?ns d unfolding B-def by blast
have 1: strict-equal (Some ?ns) s unfolding strict-equal-def by simp
have 2: realistic-executions-ind execs proof− |
  { fix d have case execs d of [] ⇒ True | aseq ≠ aseqs ⇒ realistic-AS-partial aseq ∧ set aseqs ⊆ AS-set proof (cases execs d, simp)
case (Cons aseq aseqs) from Cons realistic-exec [unfolded realistic-executions-def, THEN spec, where x = d]
  have 0: length aseq ≤ length aseq ∧ aseq ∈ AS-set ∧ aseq = lastn (length aseq) aseq unfolding lastn-def realistic-executions-def by auto
  hence 1: realistic-AS-partial aseq unfolding realistic-AS-partial-def by auto
from Cons realistic-exec [unfolded realistic-executions-def, THEN spec, where x = d]
  have 2: set aseqs ⊆ AS-set unfolding realistic-executions-def by auto
from Cons 1 2 show ?thesis by auto qed
thus ?thesis unfolding realistic-executions-ind-def by auto qed
have 3: precondition-ind s execs proof− |
  { fix d assume not-empty: fst (control s d (execs d)) ≠ None
  from not-empty realistic-exec [unfolded realistic-executions-def, THEN spec, where x = d]
  have current-aseq-is-realistic: hd (execs d) ∈ AS-set using control-spec [THEN spec, THEN spec, THEN spec, where x = execs d and x1 = d and x2 = s]
  unfolding realistic-executions-def by (cases execs d, auto)
from not-empty current-aseq-is-realistic invariant AS-prec-first-action [THEN spec, THEN spec, THEN spec,
where \( x_2 = s \) and \( x_1 = d \) and \( x = \text{hd} (\text{execs} d) \)

have AS-precondition \( s d \) (the \((\text{fst} (\text{control} s d (\text{execs} d)))\))

using control-spec[THEN \( \text{spec} \), THEN \( \text{spec} \), THEN \( \text{spec} \), where \( x = \text{execs} d \) and \( x_1 = d \) and \( x_2 = s \)]

by auto

) 

hence \( \text{fst} (\text{control} s d (\text{execs} d)) \rightarrow \text{AS-precondition} s d \)

unfolding B-def
by (cases \( \text{fst} (\text{control} s d (\text{execs} d)) \), auto)

) 

from this invariant show \(?\text{thesis}\) unfolding precondition-ind-def by auto

qed 

from thm-inductive 1 2 3 show \(?\text{thesis}\) by auto

qed

Theorem unwinding \_\_implies\_\_isecure gives security for all realistic executions. For unrealistic executions, it holds vacuously and therefore does not tell us anything. In order to prove security for this refinement (i.e., for function \( \text{run\_\_total} \), we have to prove that purging yields realistic runs.

**Lemma realistic-purge:**

shows \( \forall \ \text{execs} d \ . \ \text{realistic-executions} \ \text{execs} \rightarrow \ \text{realistic-executions} \ (\text{purge} \ \text{execs} d) \)

proof–

\{ 
  fix \ \text{execs} d 
  assume \ \text{realistic-executions} \ \text{execs} 
  hence \ \text{realistic-executions} \ (\text{purge} \ \text{execs} d) 
  using someI \ [\ \text{where} \ P = \lambda \ x . \ \text{realistic-execution} \ x \ and \ x = \ \text{execs} d] 
  unfolding \ \text{realistic-executions-def} \ \text{purge-def} \ by(\text{simp}) 
\}

thus \(?\text{thesis}\) by auto

qed

**Lemma remove-gateway-comm-subset:**

shows \( \text{set} (\text{remove-gateway-communications} \ d \ \text{exec}) \subseteq \text{set} \ \text{exec} \cup \{[]\} \)

by (induct \ \text{exec} , auto)

**Lemma realistic-ipurge-l:**

shows \( \forall \ \text{execs} d \ . \ \text{realistic-executions} \ \text{execs} \rightarrow \ \text{realistic-executions} \ (\text{ipurge-l} \ \text{execs} d) \)

proof–

\{ 
  fix \ \text{execs} d 
  assume \ 1 : \ \text{realistic-executions} \ \text{execs} 
  from \empty-in-\text{AS-set} \ \text{remove-gateway-comm-subset}[\text{where} \ d = d \ and \ \text{exec} = \text{execs} d] \ I \ \text{have} \ \text{realistic-executions} \ (\text{ipurge-l} \ \text{execs} d) 
  unfolding \ \text{realistic-execution-def} \ \text{realistic-executions-def} \ \text{ipurge-l-def} \ by(auto) 
\}

thus \(?\text{thesis}\) by auto

qed

**Lemma realistic-ipurge-r:**

shows \( \forall \ \text{execs} d \ . \ \text{realistic-executions} \ \text{execs} \rightarrow \ \text{realistic-executions} \ (\text{ipurge-r} \ \text{execs} d) \)

proof–

\{ 
  fix \ \text{execs} d 
  assume \ 1 : \ \text{realistic-executions} \ \text{execs} 
  from \empty-in-\text{AS-set} \ \text{remove-gateway-comm-subset}[\text{where} \ d = d \ and \ \text{exec} = \text{execs} d] \ I \ \text{have} \ \text{realistic-executions} \ (\text{ipurge-r} \ \text{execs} d) 
  using someI \ [\ \text{where} \ P = \lambda \ x . \ \text{realistic-execution} \ x \ and \ x = \ \text{execs} d] 
  unfolding \ \text{realistic-execution-def} \ \text{realistic-executions-def} \ \text{ipurge-r-def} \ by(auto) 
\}
We now have sufficient lemma’s to prove security for run_total. The definition of security is similar to that in Section 2.2. It now assumes that the executions are realistic and concerns function run_total instead of function run.

**definition** N-unrelated-total::bool
where
N-unrelated-total
≡ ∀ execs a n . realistic-executions execs →
(let s-f = run-total n s0 execs in
output-f s-f a = output-f (run-total n s0 (purge execs (current s-f))) a
∧ current s-f = current (run-total n s0 (purge execs (current s-f))))

**definition** N-indirect-sources-total::bool
where
N-indirect-sources-total
≡ ∀ execs a n . realistic-executions execs →
(let s-f = run-total n s0 execs in
output-f (run-total n s0 (ipurge-l execs (current s-f))) a
= output-f (run-total n s0 (ipurge-r execs (current s-f))) a)

**definition** isecure-total::bool
where
isecure-total ≡ NI-unrelated-total ∧ NI-indirect-sources-total

**theorem** unwinding-implies-secure-total:
shows isecure-total
**proof**
- from assms unwinding-implies-secure have secure-partial: NI-unrelated unfolding isecure-def by blast
- from assms unwinding-implies-secure have isecure-l-partial: NI-indirect-sources unfolding isecure-def by blast

**have** NI-unrelated-total: NI-unrelated-total
**proof**
{ fix execs a n
  assume realistic: realistic-executions execs
  from assms invariant-s0 realistic run-total-equals-run[where n=n and s=s0 and execs=execs]
  have 1: strict-equal (run n (Some s0) execs) (run-total n s0 execs) by auto

  have let s-f = run-total n s0 execs in output-f s-f a = output-f (run-total n s0 (purge execs (current s-f))) a
  ∧ current s-f = current (run-total n s0 (purge execs (current s-f)))
  **proof** (cases run n (Some s0) execs)
  case None
  thus ?thesis using 1 unfolding NI-unrelated-total-def strict-equal-def by auto
  next
  case (Some s-f)
  from realistic-purge assms invariant-s0 realistic run-total-equals-run[where n=n and s=s0 and execs=purge execs (current s-f)]
  have 2: strict-equal (run n (Some s0) (purge execs (current s-f))) (run-total n s0 (purge execs (current s-f)))
  **proof** by auto
  show ?thesis proof(cases run n (Some s0) (purge execs (current s-f)))
  case None
  from 2 None show ?thesis using 2 unfolding NI-unrelated-total-def strict-equal-def by auto
  next
  case (Some s-f2)
  from run n (Some s0) execs = Some s-f3 Some 1 2 secure-partial[unfolded NI-unrelated-def,THEN spec,THEN spec,THEN spec,where x=n and x2=execs]
show ?thesis
unfolding strict-equal-def NI-unrelated-def
by(simp add: Let-def B-def B2-def)
qed
qed
}
thus ?thesis unfolding NI-unrelated-def by auto
qed
have NI-indirect-sources-total: NI-indirect-sources-total
proof-
{
  fix execs a n
  assume realistic: realistic-executions execs
  from assms invariant-s0 realistic run-total-equals-run[where n=n and s=s0 and execs=execs]
  have 1: strict-equal (run n (Some s0) execs) (run-total n s0 execs) by auto

  have let s-f = run-total n s0 execs in output-f (run-total n s0 (ipurge-l execs (current s-f))) a = output-f (run-total n s0 (ipurge-r execs (current s-f))) a
    proof (cases run n (Some s0) execs)
    case None
      thus ?thesis using 1 unfolding NI-unrelated-total-def strict-equal-def by auto
    next
    case (Some s-f)
      from realistic-ipurge-l assms invariant-s0 realistic run-total-equals-run[where n=n and s=s0 and execs=ipurge-l execs (current s-f)]
      have 2: strict-equal (run n (Some s0) (ipurge-l execs (current s-f))) (run-total n s0 (ipurge-l execs (current s-f)))
        by auto
      from realistic-ipurge-r assms invariant-s0 realistic run-total-equals-run[where n=n and s=s0 and execs=ipurge-r execs (current s-f)]
      have 3: strict-equal (run n (Some s0) (ipurge-r execs (current s-f))) (run-total n s0 (ipurge-r execs (current s-f)))
        by auto

    show ?thesis proof(cases run n (Some s0) (ipurge-l execs (current s-f)))
      case None
      from 2 None show ?thesis using 2 unfolding NI-unrelated-total-def strict-equal-def by auto
    next
    case (Some s-ipurge-l)
      show ?thesis
      proof(cases run n (Some s0) (ipurge-r execs (current s-f)))
      case None
      from 3 None show ?thesis using 2 unfolding NI-unrelated-total-def strict-equal-def by auto
    next
    case (Some s-ipurge-r)
      from run n (Some s0) execs = Some s-f \ run n (Some s0) (ipurge-l execs (current s-f)) = Some s-ipurge-h
      Some [1 2 3 isecure1-partial[unfolded NI-indirect-sources-def,THEN spec,THEN spec,THEN spec,where x=n and x2=execs]
      show ?thesis
      unfolding strict-equal-def NI-unrelated-def
      by(simp add: Let-def B-def B2-def)
    qed
    qed
  }
  thus ?thesis unfolding NI-indirect-sources-total-def by auto
qed
2.4 CISK (Controlled Interruptible Separation Kernel)

theory CISK
imports ISK
begin

This section presents a generic model of a Controlled Interruptible Separation Kernel (CISK). It formulates security, i.e., intransitive noninterference. For a presentation of this model, see Section 2 of [3].

First, a locale is defined that defines all generic functions and all proof obligations (see Section 2.3 of [3]).

locale Controllable-Interruptible-Separation-Kernel = — CISK

fixes kstep :: 'state-t ⇒ 'action-t ⇒ 'state-t — Executes one atomic kernel action
and output-f :: 'state-t ⇒ 'action-t ⇒ 'output-t — Returns the observable behavior
and s0 :: 'state-t — The initial state
and current :: 'state-t ⇒ 'dom-t — Returns the currently active domain
and cswitch :: time-t ⇒ 'state-t ⇒ 'state-t — Performs a context switch
and interrupt :: time-t ⇒ bool — Returns true iff an interrupt occurs in the given state at the given time
and kinvolved :: 'action-t ⇒ 'dom-t-set — Returns the set of domains that are involved in the given action
and ifp :: 'dom-t ⇒ 'state-t ⇒ bool — The security policy.
and vpeq :: 'dom-t ⇒ 'state-t ⇒ 'state-t ⇒ bool — View partitioning equivalence
and AS-set :: ('action-t-list) set — Returns a set of valid action sequences, i.e., the attack surface
and invariant :: 'state-t ⇒ bool — Returns an inductive state-invariant
and AS-precondition :: 'state-t ⇒ 'dom-t ⇒ 'action-t ⇒ bool — Returns the preconditions under which the given action can be executed.
and aborting :: 'state-t ⇒ 'dom-t ⇒ 'action-t ⇒ bool — Returns true iff the action is aborted.
and waiting :: 'state-t ⇒ 'dom-t ⇒ 'action-t ⇒ bool — Returns true iff execution of the given action is delayed.
and set-error-code :: 'state-t ⇒ 'action-t ⇒ 'state-t — Sets an error code when actions are aborted.

assumes vpeq-transitive: ∀ a b c u. (vpeq u a b ∧ vpeq u b c) → vpeq u a c
and vpeq-symmetric: ∀ a b u. vpeq u a b → vpeq u b a
and vpeq-reflexive: ∀ a u. vpeq u a a
and ifp-reflexive: ∀ u. ifp u u
and weakly-step-consistent: ∀ s t u a. vpeq u s t ∧ vpeq (current s) s t ∧ invariant s ∧ AS-precondition s (current s) a ∧ invariant t ∧ AS-precondition t (current t) a ∧ current s = current t → vpeq u (kstep s a) (kstep t a)
and locally-respects: ∀ a s u. ¬ifp (current s) u ∧ invariant s ∧ AS-precondition s (current s) a → vpeq u a s (kstep s a)
and output-consistent: ∀ a s t. vpeq (current s) s t ∧ current s = current t → (output-f s a) = (output-f t a)
and step-atomicity: ∀ s a u. current (kstep s a) = current s
and cswitch-independent-of-state: ∀ n s t. current s = current t → current (cswitch n s) = current (cswitch n t)
and cswitch-consistency: ∀ u s t n . vpeq u s t → vpeq u (cswitch n s) (cswitch n t)
and empty-in-AS-set: [] ∈ AS-set
and invariant-s0: invariant s0
and invariant-after-cswitch: ∀ s n . invariant s → invariant (cswitch n s)
and precondition-after-cswitch: ∀ s d n a. AS-precondition s d a → AS-precondition (cswitch n s) d a
and AS-prec-first-action: ∀ s a s a seq . invariant s ∧ a seq ∈ AS-set ∧ a seq ≠ [] → AS-precondition s d (hd a seq)
and AS-prec-after-step: ∀ s a a' . (∃ a seq ∈ AS-set . is-sub-seq a a' seq) ∧ invariant s ∧ AS-precondition s (current s) a ∧ ¬aborting s (current s) a ∧ ¬waiting s (current s) a → AS-precondition (kstep s a) (current s) a'
and AS-prec-dom-independent: ∀ s d a a'. current s /∈ d ∧ AS-precondition s d a → AS-precondition (kstep s a') d a

and spec-of-invariant: ∀ s a . invariant s → invariant (kstep s a)

and aborting-switch-independent: ∀ n s . aborting (cswitch n s) = aborting s

and aborting-error-update: ∀ s d a a'. current s /∈ d ∧ aborting s d a → aborting (set-error-code s a') d a

and aborting-after-step: ∀ s a d . current s /∈ d → aborting (kstep s a) d = aborting s d

and aborting-consistent: ∀ s t u . vpeq u s t → aborting s u = aborting t u

and waiting-switch-independent: ∀ n s . waiting (cswitch n s) = waiting s

and waiting-error-update: ∀ s d a a'. current s /∈ d ∧ waiting s d a → waiting (set-error-code s a') d a

and waiting-consistent: ∀ s t u a . vpeq (current s) s t ∧ (′ d ∈ kinvolved a . vpeq d s t) ∧ vpeq u s t → waiting s u a = waiting t u a

and spec-of-waiting: ∀ s a . waiting s (current s) a → kstep s a = s

and set-error-consistent: ∀ s t u a . vpeq u s t → vpeq u (set-error-code s a) (set-error-code t a)

and set-error-locally-respects: ∀ s u a . -ifp (current s) u → vpeq u s (set-error-code s a)

and set-current-error-code: ∀ s a . current (set-error-code s a) = current s

and precondtion-after-set-error-code: ∀ s d a a'. AS-precondition s d a ∧ aborting s (current s) a' → AS-precondition (set-error-code s a') d a

and invariant-after-set-error-code: ∀ s a . invariant s → invariant (set-error-code s a)

and involved-ifp: ∀ s a . ∀ d ∈ (kinvolved a) . AS-precondition s (current s) a → ifp d (current s)

begin

2.4.1 Execution semantics

Control is based on generic functions aborting, waiting and set-error-code. Function aborting decides whether a certain action is aborting, given its domain and the state. If so, then function set-error-code will be used to update the state, possibly communicating to other domains that an action has been aborted. Function waiting can delay the execution of an action. This behavior is implemented in function CISK-control.

function CISK-control :: 'state-t ⇒ 'dom-t ⇒ 'action-t execution ⇒ ('action-t option × 'action-t execution × 'state-t)

where CISK-control s d [] = (None,[]), s — The thread is empty

| CISK-control s d ([]#[]) = (None,[]), s — The current action sequence has been finished and the thread has no next action sequences to execute

| CISK-control s d ([]#(as' #execs')) = (None,as' #execs', s) — The current action sequence has been finished. Skip to the next sequence

| CISK-control s d ((a#as) #execs') = (if aborting s d a then

| (None, execs',set-error-code s a)
| else if waiting s d a then
| (Some a, (a#as) #execs', s)
| else
| (Some a, as' #execs', s)) — Executing an action sequence

by pat-completeness auto

termination by lexicographic-order

Function run defines the execution semantics. This function is presented in [3] by pseudo code (see Algorithm 1). Before defining the run function, we define accessor functions for the control mechanism. Functions next-action, next_execs and next_state correspond to “control.a”, “control.x” and “control.s” in [3].

abbreviation next-action: 'state-t ⇒ ('dom-t ⇒ 'action-t execution) ⇒ 'action-t option

where next-action ≡ Kernel.next-action current CISK-control

abbreviation next-exec: 'state-t ⇒ ('dom-t ⇒ 'action-t execution) ⇒ ('dom-t ⇒ 'action-t execution)

where next-exec ≡ Kernel.next-exec current CISK-control

abbreviation next-state: 'state-t ⇒ ('dom-t ⇒ 'action-t execution) ⇒ 'state-t

where next-state ≡ Kernel.next-state current CISK-control

A thread is empty iff either it has no further action sequences to execute, or when the current action
sequence is finished and there are no further action sequences to execute.

abbreviation thread-empty :: term action-t execution ⇒ bool
where thread-empty exec ≡ exec = [] ∨ exec = [[[]]]

The following function defines the execution semantics of CISK, using function CISK_control.

function run :: time-t ⇒ state-t ⇒ (dom-t ⇒ term action-t execution) ⇒ state-t
where run 0 s execs = s
| interrupt (Suc n) ⇒ run (Suc n) s execs = run n (cswitch (Suc n) s) execs
| ~interrupt (Suc n) ⇒ ~thread-empty(execs (current s)) ⇒ run (Suc n) s execs = run n s execs
| ~interrupt (Suc n) ⇒ ~thread-empty(execs (current s)) ⇒ run (Suc n) s execs = (let control-a = next-action s execs;
control-s = next-state s execs;
control-x = next-execs s execs in
case control-a of None ⇒ run n control-s control-x
| (Some a) ⇒ run n (kstep control-s a) control-x)

using not0-implies-Suc by (metis prod-cases3,auto)
termination by lexicographic-order

2.4.2 Formulations of security

The definitions of security as presented in Section 2.2 of [3].

abbreviation kprecondition
  where kprecondition s a ≡ invariant s ∧ AS-precondition s (current s) a

definition realistic-execution
  where realistic-execution aseq ≡ set aseq ⊆ AS-set

definition realistic-executions :: (dom-t ⇒ term action-t execution) ⇒ bool
where realistic-executions execs ≡ ∀ d. realistic-execution (execs d)

abbreviation involved where involved ≡ Kernel.involved

abbreviation step where step ≡ Kernel.step kstep

abbreviation ipurge where ipurge ≡ Separation-Kernel.ipurge realistic-execution ifp

abbreviation ipurge-l where ipurge-l ≡ Separation-Kernel.ipurge-l kinvolved

abbreviation ipurge-r where ipurge-r ≡ Separation-Kernel.ipurge-r realistic-execution kinvolved

definition NI-unrelated::bool
where NI-unrelated
  ≡ ∃ execs a n . realistic-executions execs →
  (let s-f = run n s0 execs in
   output-f s-f a = output-f (run n s0 (ipurge.execs (current s-f))) a)

definition NI-indirect-sources::bool
where NI-indirect-sources
  ≡ ∃ execs a n. realistic-executions execs →
  (let s-f = run n s0 execs in
   output-f (run n s0 (ipurge-l.execs (current s-f))) a =
   output-f (run n s0 (ipurge-r.execs (current s-f))) a)

definition isecure::bool
where isecure ≡ NI-unrelated ∧ NI-indirect-sources

2.4.3 Proofs

The final theorem is unwinding_implies_isecure_CISK. This theorem shows that any interpretation of locale CISK is secure.

To prove this theorem, the refinement framework is used. CISK is a refinement of ISK, as the only idference is the control function. In ISK, this function is a generic function called control, in CISK it is interpreted in function CISK_control. It is proven that function CISK_control satisfies all the proof obligations concerning generic function control. In other words, CISK_control is proven to be an interpretation of control. Therefore, all theorems on run_total apply to the run function of CISK as well.
lemma next-action-consistent:
shows \( \forall s t \text{ execs}. \; \text{vpeq} (\text{current } s) \; s \; t \land (\forall d \in \text{involved} (\text{next-action } s \; \text{execs}) \; \text{vpeq} d \; s \; t) \land \text{current } s = \text{current } t \implies \text{next-action } s \; \text{execs} = \text{next-action } t \; \text{execs} \)

proof-
{
  fix \( s \; t \; \text{execs} \)
  assume \( \text{vpeq} : \text{vpeq} (\text{current } s) \; s \; t \)
  assume \( \text{vpeq-involved} : \forall d \in \text{involved} (\text{next-action } s \; \text{execs}) \; \text{vpeq} d \; s \; t \)
  assume \( \text{current-s-t} : \text{current } s = \text{current } t \)
  from \( \text{aborting-consistent} \; \text{current-s-t} \; \text{vpeq} \)
  have \( \text{aborting } t \; (\text{current } s) = \text{aborting } s \; (\text{current } s) \) by auto
  from \( \text{current-s-t} \) this \( \text{waiting-consistent} \; \text{vpeq-involved} \)
  have \( \text{next-action } s \; \text{execs} = \text{next-action } t \; \text{execs} \)
  unfolding Kernel.next-action-def
  by (cases \((s, (\text{current } s), \text{execs} (\text{current } s))\)) \text{ rule: CISK-control.cases,auto} 
}
thus ?thesis by auto
qed

lemma next-execs-consistent:
shows \( \forall s t \text{ execs}. \; \text{vpeq} (\text{current } s) \; s \; t \land (\forall d \in \text{involved} (\text{next-action } s \; \text{execs}) \; \text{vpeq} d \; s \; t) \land \text{current } s = \text{current } t \implies \text{fst} (\text{snd} (\text{CISK-control } s \; (\text{current } s) \; (\text{execs} (\text{current } s)))) = \text{fst} (\text{snd} (\text{CISK-control } t \; (\text{current } s) \; (\text{execs} (\text{current } s)))) \)

proof-
{
  fix \( s \; t \; \text{execs} \)
  assume \( \text{vpeq} : \text{vpeq} (\text{current } s) \; s \; t \)
  assume \( \text{vpeq-involved} : \forall d \in \text{involved} (\text{next-action } s \; \text{execs}) \; \text{vpeq} d \; s \; t \)
  assume \( \text{current-s-t} : \text{current } s = \text{current } t \)
  from \( \text{aborting-consistent} \; \text{current-s-t} \; \text{vpeq} \)
  have \( \text{aborting } t \; (\text{current } s) = \text{aborting } s \; (\text{current } s) \) by auto
  from \( \text{current-s-t} \) this \( \text{waiting-consistent} \; \text{vpeq-involved} \)
  unfolding Kernel.next-action-def\; Kernel.involved-def
  by (cases \((s, (\text{current } s), \text{execs} (\text{current } s))\)) \text{ rule: CISK-control.cases,auto \; split add: split-if-asm} 
}
thus ?thesis by auto
qed

lemma next-state-consistent:
shows \( \forall s t u \text{ execs}. \; \text{vpeq} (\text{current } s) \; s \; t \land \text{vpeq } u \; s \; t \land \text{current } s = \text{current } t \implies \text{vpeq } u \; (\text{next-state } s \; \text{execs}) \) (next-state \( t \; \text{execs} \))

proof-
{
  fix \( s \; t \; u \; \text{execs} \)
  assume \( \text{vpeq-s-t} : \text{vpeq} (\text{current } s) \; s \; t \land \text{vpeq } u \; s \; t \)
  assume \( \text{current-s-t} : \text{current } s = \text{current } t \)
  from \( \text{vpeq-s-t} \; \text{current-s-t} \)
  have \( \text{vpeq } u \; (\text{next-state } s \; \text{execs}) \; (\text{next-state } t \; \text{execs}) \)
  unfolding Kernel.next-state-def
  using \( \text{aborting-consistent} \; \text{set-error-consistent} \)
  by (cases \((s, (\text{current } s), \text{execs} (\text{current } s))\)) \text{ rule: CISK-control.cases,auto} 
}
thus ?thesis by auto
qed
lemma current-next-state:
shows ∀ s execs . current (next-state s execs) = current s
proof-
{ 
  fix s execs
  have current (next-state s execs) = current s
    unfolding Kernel.next-state-def
    using current-set-error-code
    by (cases (s,(current s),execs (current s)) rule: CISK-control.cases.auto)
} 
thus ?thesis by auto
qed

lemma locally-respects-next-state:
shows ∀ s u execs . ~ifp (current s) u → vpeq u s (next-state s execs)
proof-
{ 
  fix s u execs
  assume ~ifp (current s) u 
  hence vpeq u s (next-state s execs)
    unfolding Kernel.next-state-def
    using vpeq-reflexive set-error-locally-respects
    by (cases (s,(current s),execs (current s)) rule: CISK-control.cases.auto)
} 
thus ?thesis by auto
qed

lemma CISK-control-spec:
shows ∀ s d aseqs .
  case CISK-control s d aseqs of
    (a, aseqs', s') ⇒
      thread-empty aseqs ∧ (a, aseqs') = (None, []) ∨
      aseqs ≠ [] ∧ hd aseqs ≠ [] ∧ ¬ aborting s' d (the a) ∧ ¬ waiting s' d (the a) ∧ (a, aseqs') = (Some (hd (hd aseqs)), tl (hd aseqs) # tl aseqs) ∨
      aseqs ≠ [] ∧ hd aseqs ≠ [] ∧ waiting s' d (the a) ∧ (a, aseqs', s') = (Some (hd (hd aseqs)), aseqs, s) ∨ (a, aseqs') = (None, tl aseqs)
proof-
{ 
  fix s d aseqs
  have case CISK-control s d aseqs of
    (a, aseqs', s') ⇒
      thread-empty aseqs ∧ (a, aseqs') = (None, []) ∨
      aseqs ≠ [] ∧ hd aseqs ≠ [] ∧ ¬ aborting s' d (the a) ∧ ¬ waiting s' d (the a) ∧ (a, aseqs') = (Some (hd (hd aseqs)), tl (hd aseqs) # tl aseqs) ∨
      aseqs ≠ [] ∧ hd aseqs ≠ [] ∧ waiting s' d (the a) ∧ (a, aseqs', s') = (Some (hd (hd aseqs)), aseqs, s) ∨ (a, aseqs') = (None, tl aseqs)
      by (cases (s,d,aseqs) rule: CISK-control.cases.auto)
} 
thus ?thesis by auto
qed

lemma next-action-after-cswitch:
shows ∀ s n d aseqs . fst (CISK-control (cswitch n s) d aseqs) = fst (CISK-control s d aseqs)
proof-
{ 
  fix s n d aseqs
have \( \text{fst} (\text{CISK-control (cswitch n s) d aseqs}) = \text{fst} (\text{CISK-control s d aseqs}) \)

using aborting-switch-independent waiting-switch-independent

by (cases (s,d,aseqs) rule: CISK-control.cases.auto)

}\)

thus \(?\text{thesis}\) by \text{auto}

 qed


lemma \text{next-action-after-next-state}:

shows \( \forall s \text{ execs d . current s \neq d \longrightarrow} \text{fst} (\text{CISK-control (next-state s execs) d (execs d)}) = \text{None} \lor \text{fst} (\text{CISK-control (next-state s execs) d (execs d)}) = \text{fst} (\text{CISK-control s d (execs d)}) \)

proof--

{\}

fix s execs d aseqs

assume I: \text{current s \neq d}

have \( \text{fst} (\text{CISK-control (next-state s execs) d aseqs}) = \text{None} \lor \text{fst} (\text{CISK-control (next-state s execs) d aseqs}) = \text{fst} (\text{CISK-control s d aseqs}) \)

proof (cases (s,d,aseqs) rule: CISK-control.cases,simp,simp,simp)

case (4 sa da a as execs')

thus \(?\text{thesis}\)

 unfolding Kernel.next-state-def

 using aborting-error-update waiting-error-update 1

 by (cases (sa,current sa,execs (current sa)) rule: CISK-control.cases,auto split: split-if-asm)

qed

\}

thus \(?\text{thesis}\) by \text{auto}

qed


lemma \text{next-action-after-step}:

shows \( \forall s a d aseqs . \text{current s \neq d \longrightarrow} \text{fst} (\text{CISK-control (step s a) d aseqs}) = \text{fst} (\text{CISK-control s d aseqs}) \)

proof--

{\}

fix s a d aseqs

assume I: \text{current s \neq d}

from this aborting-after-step

have \( \text{fst} (\text{CISK-control (step s a) d aseqs}) = \text{fst} (\text{CISK-control s d aseqs}) \)

 unfolding Kernel.step-def

by (cases (s,d,aseqs) rule: CISK-control.cases,simp,simp,simp,cases a,auto)

}\)

thus \(?\text{thesis}\) by \text{auto}

qed


lemma \text{next-state-precondition}:

shows \( \forall s d a \text{ execs . AS-precondition s d a \longrightarrow} \text{AS-precondition (next-state s execs) d a} \)

proof--

{\}

fix s a d execs

assume \text{AS-precondition s d a}

hence \text{AS-precondition (next-state s execs) d a}

 unfolding Kernel.next-state-def

 using precondition-after-set-error-code

 by (cases (s,(current s),execs (current s)) rule: CISK-control.cases,auto)

}\)

thus \(?\text{thesis}\) by \text{auto}

qed


lemma \text{next-state-invariant}:
D31.1 – Formal specification of a generic MILS separation kernel

shows $\forall s \text{ execs. invariant } s \rightarrow \text{ invariant } (\text{next-state } s \text{ execs})$
proof−
{ 
  fix $s \text{ execs}$
  assume invariant $s$
  hence invariant $\ (\text{next-state } s \text{ execs})$
  unfolding Kernel.next-state-def
  using invariant-after-set-error-code
  by (cases $(s, (\text{current } s), \text{execs } (\text{current } s))$ rule: CISK-control.cases.auto)
}
thus $?\thesis$ by auto
qed

lemma next-action-from-execs
shows $\forall s \text{ execs } . \text{next-action } s \text{ execs } \mapsto (\lambda a . a \in \text{actions-in-execution } (\text{execs } (\text{current } s)))$
proof−
{ 
  fix $s \text{ execs}$
  { 
    fix $a$
    assume $1: \text{next-action } s \text{ execs } = \text{Some } a$
    from $1$ have $a \in \text{actions-in-execution } (\text{execs } (\text{current } s))$
    unfolding Kernel.next-action-def actions-in-execution-def
    by (cases $(s, (\text{current } s), \text{execs } (\text{current } s))$ rule: CISK-control.cases.auto split add: split-if-asm)
  }
  hence $\text{next-action } s \text{ execs } \mapsto (\lambda a . a \in \text{actions-in-execution } (\text{execs } (\text{current } s)))$
  unfolding B-def
  by (cases next-action $s \text{ execs}$.auto)
}
thus $?\thesis$ unfolding $B$-def by (auto)
qed

lemma next-execs-subset:
shows $\forall s \text{ execs } u . \text{actions-in-execution } (\text{next-execs } s \text{ execs } u) \subseteq \text{actions-in-execution } (\text{execs } u)$
proof−
{ 
  fix $s \text{ execs } u$ 
  have $\text{actions-in-execution } (\text{next-execs } s \text{ execs } u) \subseteq \text{actions-in-execution } (\text{execs } u)$
  unfolding Kernel.next-execs-def actions-in-execution-def
  by (cases $(s, (\text{current } s), \text{execs } (\text{current } s))$ rule: CISK-control.cases.auto split add: split-if-asm)
}
thus $?\thesis$ by auto
qed

theorem unwinding-implies-secure-CISK:
shows $\text{secure}$
proof−
interpret int: Interruptible-Separation-Kernel kstep output-f $s0$ current cswitch interrupt kprecondition realistic-execution CISK-control kinvolved ifp vpeq AS-set invariant AS-precondition aborting waiting
proof (unfold-locales)
show $\forall a b c u . \text{vpeq } u a b \land \text{vpeq } u b c \rightarrow \text{vpeq } u a c$
  using vpeq-transitive by blast
show $\forall a b u . \text{vpeq } u a b \rightarrow \text{vpeq } u b a$
  using vpeq-symmetric by blast
show $\forall a u . \text{vpeq } u a a$
  using vpeq-reflexive by blast
show ∀ u. ifp u u using ifp-reflexive by blast
show ∀ s t u. vpeq u s t ∧ vpeq (current s) s t ∧ kprecondition s a ∧ kprecondition t a ∧ current s = current t → vpeq u (kstep s a) (kstep t a) using weakly-step-consistent by blast
show ∀ a s u. ¬ifp (current s) u ∧ kprecondition s a → vpeq u s (kstep s a) using locally-respects by blast
show ∀ a s t. vpeq (current s) s t ∧ current s = current t → (output-f s a) = (output-f t a) using output-consistent by blast
show ∀ s a. current (kstep s a) = current s using step-atomicity by blast
show ∀ n s t. current s = current t → current (cswitch n s) = current (cswitch n t) using cswitch-independent-of-state by blast
show ∀ u s t n. vpeq u s t → vpeq u (switch n s) (switch n t) using cswitch-consistency by blast
show ∀ s t execs. vpeq (current s) s t ∧ (∀ d ∈ involved (next-action s execs) . vpeq d s t) ∧ current s = current t → next-action s execs = next-action t execs using next-action-consistent by blast
show ∀ s t execs. vpeq (current s) s t ∧ vpeq u s t ∧ current s = current t → vpeq u (next-state t execs) using next-state-t-exec by auto
show ∀ s u execs. current (next-state t execs) = current s using current-next-state by auto
show ∀ s u execs. ¬ifp (current s) u → vpeq u s (next-state t execs) using locally-respects-next-state by auto
show [] ∈ AS-set using empty-in-AS-set by blast
show ∀ s n. invariant s → invariant (cswitch n s) using invariant-after-cswitch by blast
show ∀ s d n a. AS-precondition s d a → AS-precondition (cswitch n s) d a using pre-condition-after-cswitch by blast
show invariant s0 using invariant-s0 by blast
show ∀ s a a' . (∃ aseq∈AS-set . is-sub-seq a a' aseq) ∧ invariant s ∧ AS-precondition s (current s) a ∧ ∼ aborting s (current s) a ∧ ¬ waiting s (current s) a → AS-precondition (kstep s a) (current s) a' using AS-prec-after-step by blast
show ∀ s d a a' . current s = d ∧ AS-precondition s d a → AS-precondition (kstep s a') d a using AS-prec-dom-independent by blast
show ∀ s a . invariant s → invariant (kstep s a) using spec-of-invariant by blast
show ∀ s a . kprecondition s a ≡ kprecondition s a by auto
show ∀ aseq. realistic-execution aseq ≡ set aseq ∈ AS-set unfolding realistic-execution-def by auto
show ∀ s a . ∀ d ∈ involved a. kprecondition s (the a) → ifp d (current s) using involved-ifp unfolding Kernel.involved-def by (auto split: option.splits)
show ∀ s execs. next-action s execs → (∀ a. a ∈ actions-in-execution (execs (current s))) using next-action-from-execs by blast
show ∀ s execs u. actions-in-execution (next-execs s execs u) ⊆ actions-in-execution (execs u)
using next-execs-subset by blast
show ∀ s d aseqs.
case CISK-control s d aseqs of
(a, aseqs', s') ⇒
thread-empty aseqs ∧ (a, aseqs') = (None, []) ∨
aseqs ≠ [] ∧ hd aseqs ≠ [] ∧ ¬ aborting s' d (the a) ∧ ¬ waiting s' d (the a) ∧ (a, aseqs') = (Some (hd (hd aseqs)), tl (hd aseqs) ≠ [] ∧ aseqs) ∨
aseqs ≠ [] ∧ hd aseqs ≠ [] ∧ waiting s' d (the a) ∧ (a, aseqs', s') = (Some (hd (hd aseqs)), aseqs, s) ∨ (a, aseqs') = (None, nil aseqs)
using CISK-control-spec by blast
show ∀ s n d aseqs. fst (CISK-control (cswitch n s) d aseqs) = fst (CISK-control s d aseqs)
using next-action-after-cswitch by auto
show ∀ s execs.
current s ≠ d →
fst (CISK-control (next-state s execs) d (execs d)) = None ∨ fst (CISK-control (next-state s execs) d (execs d) d)) = fst (CISK-control s d (execs d))
using next-action-after-next-state by auto
show ∀ s d aseqs. current s ≠ d →
fst (CISK-control (step s a) d aseqs) = fst (CISK-control s d aseqs)
using next-action-after-step by auto
show ∀ s d a execs. AS-precondition s d a → AS-precondition (next-state s execs) d a
using next-state-precondition by auto
show ∀ s execs. invariant s → invariant (next-state s execs)
using next-state-invariant by auto
show ∀ s a. waiting s (current s) a → kstep s a = s
using spec-of-waiting by blast
qed

note interpreted = Interruptible-Separation-Kernel kstep output-f s0 current cswitch kprecondition realistic-execution
CISK-control kinvolved ifp vpeq AS-set invariant AS-precondition aborting waiting

CISK-control kinvolved ifp vpeq AS-set invariant AS-precondition aborting waiting - interrupt]

have run-equals-run-total:
∧ n s execs . run n s execs = Interruptible-Separation-Kernel.run-total.kstep current cswitch interrupt CISK-control n s execs

proof-
fix n s execs
show run n s execs = Interruptible-Separation-Kernel.run-total.kstep current cswitch interrupt CISK-control n s execs
using interpreted int.step-def
by (induct n s execs rule: run-total-induct,auto split: option.splits)
qed

from interpreted
have & Interruptible-Separation-Kernel.isecure-total kstep output-f s0 current cswitch interrupt realistic-execution
CISK-control kinvolved ifp
by (metis int.unwinding-implies-secure-total)
from 0 run-equals-run-total
have 1: NI-unrelated
by (metis realistic-executions-def int.isecure-total-def int.realistic-executions-def int.NI-unrelated-total-def
NI-unrelated-def)
from 0 run-equals-run-total
have 2: NI-indirect-sources
by (metis realistic-executions-def int.NI-indirect-sources-total-def int.isecure-total-def int.realistic-executions-def
NI-indirect-sources-def)
from 1 2 show ?thesis unfolding isecure-def by auto
3 Instantiation by a separation kernel with concrete actions

In the previous section, no concrete actions for the step function were given. The foremost point we want to make by this instantiation is to show that we can instantiate the CISK model of the previous section with an implementation that, for the step function, as actions, provides events and interprocess communication (IPC). System call invocations that can be interrupted at certain interrupt points are split into several atomic steps. A communication interface of events and IPC is less “trivial” than it may seem at a first glance, for example the L4 microkernel API only provided IPC as communication primitive [1]. In particular, the concrete actions illustrate how an application of the CISK framework can be used to separate policy enforcement from other computations unrelated to policy enforcement.

Our separation kernel instantiation also has a notion of partitions. A partition is a logical unit that serves to encapsulate a group of CISK threads by, amongst others, enforcing a static per-partition access control policy to system resources. In the following instantiation, while the subjects of the step function are individual threads, the information flow policy ifp is defined at the granularity of partitions, which is realistic for many separation kernel implementations.

Lastly, as a limited manipulation of an access control policy is often needed, we also provide an invariant for having a dynamic access control policy whose maximal closure is bounded by the static per-partition access control policy. That the dynamic access control policy is a subset of a static access control policy is expressed by the invariant sp_subset. A use case for this is when you have statically configured access to files by subjects, but whether a file can be read/written also depends on whether the file has been dynamically opened or not. The instantiation provides infrastructure for such an invariant on the relation of a dynamic policy to a static policy, and shows how the invariant is maintained, without modeling any API for an open/close operation.

3.1 Model of a separation kernel configuration

theory step-configuration
  imports Main
begin

3.1.1 Type definitions

The separation kernel partitions are considered to be the “subjects” of the information flow policy ifp. A file provider is a partition that, via a file API (read/write), provides files to other partitions. The configuration statically defines which partitions can act as a file provider and also the access rights (right/write) of other partitions to the files provided by the file provider. Some separation kernels include a management for address spaces (tasks), that may be hierarchically structured. Such a task hierarchy is not part of this model.

typedcl partition-id-t
typedcl thread-id-t
typedcl page-t — physical address of a memory page
typedcl filep-t — name of file provider

datatype obj-id-t =
  PAGE page-t
  | FILEP filep-t

datatype mode-t =
READ — The subject has right to read from the memory page, from the files served by a file provider.

WRITE — The subject has right to write to the memory page, from the files served by a file provider.

PROVIDE — The subject has right serve as the file provider. This mode is not used for memory pages or ports.

3.1.2 Configuration

The information flow policy is implicitly specified by the configuration. The configuration does not contain the communication rights between partitions (subjects). However, the rights can be derived from the configuration. For example, if two partitions $p$ and $p'$ can access a file $f$, then $p$ and $p'$ can communicate. See below.

consts
configured-subj-obj :: partition-id-t ⇒ obj-id-t ⇒ mode-t ⇒ bool

Each user thread belongs to a partition. The relation is fixed at system startup. The configuration specifies how many threads a partition can create, but this limit is not part of the model.

consts
partition :: thread-id-t ⇒ partition-id-t

end

3.2 Formulation of a subject-subject communication policy and an information flow policy, and showing both can be derived from subject-object configuration data

theory step-policies
imports step-configuration
begin

3.2.1 Specification

In order to use CISK, we need an information flow policy $ifp$ relation. We also express a static subject-subject $sp$-$spec$-$subj$-$obj$ and subject-object $sp$-$spec$-$subj$-$subj$ access control policy for the implementation of the model. The following locale summarizes all properties we need.

locale policy-axioms =
fixes $sp$-$spec$-$subj$-$obj$ :: $'a$ ⇒ obj-id-t ⇒ mode-t ⇒ bool

and $sp$-$spec$-$subj$-$subj$ :: $'a$ ⇒ $'a$ ⇒ bool

and $ifp$ :: $'a$ ⇒ $'a$ ⇒ bool

assumes $sp$-$spec$-$file$-$provider$: $\forall$ p1 p2 f m1 m2.

$sp$-$spec$-$subj$-$obj$ p1 (FILEP f) m1 ∧
$sp$-$spec$-$subj$-$obj$ p2 (FILEP f) m2 → $sp$-$spec$-$subj$-$subj$ p1 p2

and $sp$-$spec$-$no$-$wronly$-$pages$: $\forall$ p x. $sp$-$spec$-$subj$-$obj$ p (PAGE x) WRITE →→ $sp$-$spec$-$subj$-$obj$ p (PAGE x) READ

and $ifp$-$reflexive$:
$\forall$ p. $ifp$ p p

and $ifp$-$compatible$-$with$-$sp$-$spec$:
$\forall$ a b. $sp$-$spec$-$subj$-$subj$ a b →→ $ifp$ a b ∧ $ifp$ b a

and $ifp$-$compatible$-$with$-$ipc$:
$\forall$ a b c x. ($sp$-$spec$-$subj$-$subj$ a b ∧
$sp$-$spec$-$subj$-$obj$ b (PAGE x) WRITE ∧ $sp$-$spec$-$subj$-$obj$ c (PAGE x) READ)
→→ $ifp$ a c

begin end
3.2.2 Derivation

The configuration data only consists of a subject-object policy. We derive the subject-subject policy and the information flow policy from the configuration data and prove that properties we specified in Section 3.2.1 are satisfied.

locale abstract-policy-derivation =
fixes configuration-subj-obj :: 'a ⇒ obj-id-t ⇒ mode-t ⇒ bool
begin

definition sp-spec-subj-obj a x m ≡
configuration-subj-obj a x m ∨ (∃ y . x = PAGE y ∧ m = READ ∧ configuration-subj-obj a x WRITE)

definition sp-spec-subj-subj a b ≡
∃ f m1 m2 . sp-spec-subj-obj a (FILEP f) m1 ∧ sp-spec-subj-obj b (FILEP f) m2

definition ifp a b ≡
sp-spec-subj-subj a b
∨ sp-spec-subj-subj b a
∨ (∃ c y . sp-spec-subj-subj a c
∧ sp-spec-subj-obj c (PAGE y) WRITE
∧ sp-spec-subj-obj b (PAGE y) READ)
∨ (a = b)

Show that the policies specified in Section 3.2.1 can be derived from the configuration and their definitions.

lemma correct:
shows policy-axioms sp-spec-subj-obj sp-spec-subj-subj ifp
proof (unfold-locales)
show sp-spec-file-provider:
∀ p1 p2 f m1 m2 .
sp-spec-subj-obj p1 (FILEP f) m1 ∧
sp-spec-subj-obj p2 (FILEP f) m2 ⟹ sp-spec-subj-subj p1 p2
unfolding sp-spec-subj-subj-def by auto
show sp-spec-no-wronly-pages:
∀ p x . sp-spec-subj-obj p (PAGE x) WRITE ⟹ sp-spec-subj-obj p (PAGE x) READ
unfolding sp-spec-subj-obj-def by auto
show ifp-reflexive:
∀ p . ifp p p
unfolding ifp-def by auto
show ifp-compatible-with-sp-spec:
∀ a b . sp-spec-subj-subj a b ⟹ ifp a b ∧ ifp b a
unfolding ifp-def by auto
show ifp-compatible-with-ipc:
∀ a b c x . (sp-spec-subj-subj a b
∧ sp-spec-subj-obj b (PAGE x) WRITE ∧ sp-spec-subj-obj c (PAGE x) READ)
⟹ ifp a c
unfolding ifp-def by auto
qed
end

type-synonym sp-subj-subj-t = partition-id-t ⇒ partition-id-t ⇒ bool

type-synonym sp-subj-obj-t = partition-id-t ⇒ obj-id-t ⇒ mode-t ⇒ bool

interpretation Policy: abstract-policy-derivation configured-subj-obj,
interpretation Policy-properties: policy-axioms Policy.sp-spec-subj-obj Policy.sp-spec-subj-subj Policy.ifp
using Policy.correct by auto
lemma example-how-to-use-properties-in-proofs:
  shows ∀ p. Policy.ifp p p
  using Policy-properties.ifp-reflexive by auto

end

3.3 Separation kernel state and atomic step function

theory step
  imports step-policies
begin

3.3.1 Interrupt points

To model concurrency, each system call is split into several atomic steps, while allowing interrupts between the steps. The state of a thread is represented by an “interrupt point” (which corresponds to the value of the program counter saved by the system when a thread is interrupted).

datatype ipc-direction-t = SEND \ | \ RECV

datatype ipc-stage-t = PREP \ | \ WAIT \ | \ BUF page-t

datatype ev-consume-t = EV-CONSUME-ALL \ | \ EV-CONSUME-ONE

datatype ev-wait-stage-t = EV-PREP \ | \ EV-WAIT \ | \ EV-FINISH

datatype ev-signal-stage-t = EV-SIGNAL-PREP \ | \ EV-SIGNAL-FINISH

datatype int-point-t =
  SK-IPC ipc-direction-t ipc-stage-t thread-id-t page-t — The thread is executing a sending / receiving IPC.
  | SK-EV-WAIT ev-wait-stage-t ev-consume-t — The thread is waiting for an event.
  | SK-EV-SIGNAL ev-signal-stage-t thread-id-t — The thread is sending an event.
  | NONE — The thread is not executing any system call.

3.3.2 System state

typedec obj-t — value of an object

  Each thread belongs to a partition. The relation is fixed (in this instantiation of a separation kernel).

consts
  partition :: thread-id-t ⇒ partition-id-t

  The state contains the dynamic policy (the communication rights in the current state of the system, for example).

record thread-t =

  ev-counter :: nat — event counter

record state-t =

  sp-impl-subj-subj :: sp-subj-subj-t — current subject-subject policy
  sp-impl-subj-obj :: sp-subj-obj-t — current subject-object policy
  current :: thread-id-t — current thread
  obj :: obj-id-t ⇒ obj-t — values of all objects
  thread :: thread-id-t ⇒ thread-t — internal state of threads

  Later (Section 3.4), the system invariant sp-subset will be used to ensure that the dynamic policies (sp_impl,...) are a subset of the corresponding static policies (sp_spec,...).
3.3.3 Atomic step

**Helper functions**  Set new value for an object.

**definition** `set-object-value :: obj-id-t ⇒ obj-t ⇒ state-t ⇒ state-t where`

`set-object-value obj-id val s` = `s (fun-upd (obj s) obj-id val)`

Return a representation of the opposite direction of IPC communication.

**definition** `opposite-ipc-direction :: ipc-direction-t ⇒ ipc-direction-t where`

`opposite-ipc-direction dir` ≡ `case dir of SEND ⇒ RECV | RECV ⇒ SEND`

Add an access right from one partition to an object. In this model, not available from the API, but shows how dynamic changes of access rights could be implemented.

**definition** `add-access-right :: partition-id-t =⇒ obj-id-t =⇒ mode-t =⇒ state-t =⇒ state-t where`

`add-access-right part-id obj-id m s` = `s (sp-impl-subj-obj q q q′ q′′. (part-id = q ∧ obj-id = q′ ∧ m = q′′))` ∨ `sp-impl-subj-obj s q q′ q′′())`

Add a communication right from one partition to another. In this model, not available from the API.

**definition** `add-comm-right :: partition-id-t ⇒ partition-id-t ⇒ state-t ⇒ state-t where`

`add-comm-right p p′ s` ≡ `s (sp-impl-subj-subj s q q′ q′′. (p = q ∧ p′ = q′))` ∨ `sp-impl-subj-subj s q q′ q′′())`

**Model of IPC system call**  We model IPC with the following simplifications:

1. The model contains the system calls for sending an IPC (SEND) and receiving an IPC (RECV), often implementations have a richer API (e.g. combining SEND and RECV in one invocation).

2. We model only a copying (“BUF”) mode, not a memory-mapping mode.

3. The model always copies one page per syscall.

**definition** `ipc-precondition :: thread-id-t ⇒ ipc-direction-t ⇒ thread-id-t ⇒ page-t ⇒ state-t ⇒ bool where`

`ipc-precondition tid dir partner page s ≡`

`let sender = (case dir of SEND ⇒ tid | RECV ⇒ partner) in`

`let receiver = (case dir of SEND ⇒ partner | RECV ⇒ tid) in`

`let local-access-mode = (case dir of SEND ⇒ READ | RECV ⇒ WRITE) in`

`sp-impl-subj-subj s (partition sender) (partition receiver)`

`∧ sp-impl-subj-obj s (partition tid) (PAGE page) local-access-mode)`

**definition** `atomic-step-ipc :: thread-id-t ⇒ ipc-direction-t ⇒ ipc-stage-t ⇒ thread-id-t ⇒ page-t ⇒ state-t ⇒ state-t where`

`atomic-step-ipc tid dir stage partner page s ≡`

`case stage of`

`PREP ⇒`

`s`

`WAIT ⇒`

`s`

`BUF page′ ⇒`

`(case dir of`

`SEND ⇒`

`set-object-value (PAGE page′) (obj s (PAGE page)) s)`

`RECV ⇒ s)}`
Model of event syscalls

**Definition**

\[
\text{ev-signal-precondition} \quad \text{tid partner s} \equiv \\
\text{sp-impl-subj-subj s (partition tid) (partition partner)}
\]

**Definition**

\[
\text{atomic-step-ev-signal} \quad \text{tid partner s} = \\
\text{s ( thread := fun-upd (thread s) partner (thread s partner ( \{ ev-counter := Suc (ev-counter (thread s partner) ) \})) )}
\]

**Definition**

\[
\text{atomic-step-ev-wait-one} \quad \text{tid s} = \\
\text{s ( thread := fun-upd (thread s) tid (thread s tid ( ev-counter := (ev-counter (thread s tid) - 1)) ))}
\]

**Definition**

\[
\text{atomic-step-ev-wait-all} \quad \text{tid s} = \\
\text{s ( thread := fun-upd (thread s) tid (thread s tid ( ev-counter := 0)) )}
\]

**Instantiation of CISK aborting and waiting**

In this instantiation of CISK, the *aborting* function is used to indicate security policy enforcement. An IPC call aborts in its PREP stage if the precondition for the calling thread does not hold. An event signal call aborts in its EV-SIGNAL-PREP stage if the precondition for the calling thread does not hold.

**Definition**

\[
\text{aborting} \quad \text{s tid a} \equiv \\
\text{case a of SK-IPC dir PREP partner page} \Rightarrow \\
\text{ipc-precondition tid dir partner page s} \Rightarrow \\
\text{SK-EV-SIGNAL EV-SIGNAL-PREP partner} \Rightarrow \\
\text{ev-signal-precondition tid partner s} \Rightarrow \\
\text{False}
\]

The *waiting* function is used to indicate synchronization. An IPC call waits in its WAIT stage while the precondition for the partner thread does not hold. An EV_WAIT call waits until the event counter is not zero.

**Definition**

\[
\text{waiting} :: \text{s tid a} \equiv \\
\text{case a of SK-IPC dir WAIT partner page} \Rightarrow \\
\text{ipc-precondition partner (opposite-ipc-direction dir) tid (SOME page' \cdot True) s} \Rightarrow \\
\text{SK-EV-WAIT EV-PREP - } \Rightarrow \text{False} \Rightarrow \\
\text{SK-EV-WAIT EV-WAIT - } \Rightarrow \text{ev-counter (thread s tid) = 0} \Rightarrow \\
\text{SK-EV-WAIT EV-FINISH - } \Rightarrow \text{False} \Rightarrow \\
\text{False}
\]

**The atomic step function.**

In the definition of *atomic-step* the arguments to an interrupt point are not taken from the thread state – the argument given to *atomic-step* could have an arbitrary value. So, seen in isolation, *atomic-step* allows more transitions than actually occur in the separation kernel. However, the CISK framework (1) restricts the atomic step function by the *waiting* and *aborting* functions as well (2) the set of realistic traces as attack sequences rAS-set (Section 3.8). An additional condition is that (3) the dynamic policy used in *aborting* is a subset of the static policy. This is ensured by the invariant \(sp-subset\).

**Definition**

\[
\text{atomic-step} :: \text{ipt} \Rightarrow \text{state-t} \Rightarrow \text{int-point-t} \Rightarrow \text{state-t} \quad \text{where} \\
\text{atomic-step s ipt} \equiv \\
\text{case ipt of} \\
\text{SK-IPC dir stage partner page} \Rightarrow \\
\text{atomic-step-ipc (current s) dir stage partner page s}
\]
**3.4 Preconditions and invariants for the atomic step**

The dynamic/implementation policies have to be compatible with the static configuration.

**definition**

\[
\text{sp-subset } s \equiv \\
\left( \forall \ p1\ p2. \sp-impl-subj-subj\ s\ p1\ p2 \rightarrow \Policy.\sp-spec-subj-subj\ p1\ p2 \right) \land \left( \forall \ p1\ p2\ m. \sp-impl-subj-obj\ s\ p1\ p2\ m \rightarrow \Policy.\sp-spec-subj-obj\ p1\ p2\ m \right)
\]

The following predicate expresses the precondition for the atomic step. The precondition depends on the type of the atomic action.

**definition**

\[
\text{atomic-step-precondition } s\ tid\ ipt \equiv \\
\text{case } ipt\ of \\
\text{SK-IPC dir \ WAIT partner page } \Rightarrow \\
\left( ^* \text{the thread managed it past PREP stage} ^* \right) \text{ipc-precondition tid dir partner page } s \\
\text{SK-IPC dir } (\text{BUF page}') \text{ partner page } \Rightarrow \\
\left( ^* \text{both the calling thread and its communication partner} \\
\text{managed it past PREP and WAIT stages} ^* \right) \text{ipc-precondition tid dir partner page } s \\
\land \text{ipc-precondition partner } (\text{opposite-ipc-direction } dir) \text{ tid page} \ ' s \\
\text{SK-EV-SIGNAL EV-SIGNAL-FINISH partner } \Rightarrow \\
\text{ev-signal-precondition tid partner } s \\
\land \Rightarrow \\
\left( ^* \text{No precondition for other interrupt points} ^* \right) \text{True}
\]

The invariant to be preserved by the atomic step function. The invariant is independent from the type of the atomic action.

**definition**

\[
\text{atomic-step-invariant } s \equiv \text{sp-subset } s
\]

**3.4.1 Atomic steps of SK_IPC preserve invariants**

**lemma**

\[
\text{set-object-value-invariant:} \\
\text{shows } \text{atomic-step-invariant } s = \text{atomic-step-invariant } (\text{set-object-value } ob\ va\ s)
\]

**proof**

\[
\text{show } \text{thesis using} \text{ assms} \\
\text{unfolding} \text{atomic-step-invariant-def atomic-step-precondition-def ipc-precondition-def}
\]
lemma set-thread-value-invariant:
  shows atomic-step-invariant s = atomic-step-invariant (s (thread := thrst))
proof –
    sp-subset-def set-object-value-def Let-def
  by (simp split add: int-point-t.splits ipc-stage-t.splits ipc-direction-t.splits)
qed

lemma atomic-ipc-preserves-invariants:
  fixes s :: state-t
  and tid :: thread-id-t
  assumes atomic-step-invariant s
  shows atomic-step-invariant (atomic-step-ipc tid dir stage partner page s)
proof –
  show ?thesis
    proof (cases stage)
      case PREP
        from this assms show ?thesis
          unfolding atomic-step-ipc-def atomic-step-invariant-def by auto
      next
      case WAIT
        from this assms show ?thesis
          unfolding atomic-step-ipc-def atomic-step-invariant-def by auto
      next
      case BUF
        show ?thesis
          using assms BUF set-object-value-invariant unfolding atomic-step-ipc-def
          by (simp split add: ipc-direction-t.splits)
    qed
qed

lemma atomic-ev-wait-one-preserves-invariants:
  fixes s :: state-t
  and tid :: thread-id-t
  assumes atomic-step-invariant s
  shows atomic-step-invariant (atomic-step-ev-wait-one tid s)
proof –
  from assms show ?thesis
    unfolding atomic-step-ev-wait-one-def atomic-step-invariant-def sp-subset-def
    by auto
qed

lemma atomic-ev-wait-all-preserves-invariants:
  fixes s :: state-t
  and tid :: thread-id-t
  assumes atomic-step-invariant s
  shows atomic-step-invariant (atomic-step-ev-wait-all tid s)
proof –
  from assms show ?thesis
    unfolding atomic-step-ev-wait-all-def atomic-step-invariant-def sp-subset-def
    by auto
lemma atomic-ev-signal-preserves-invariants:
  fixes s :: state-t
  and tid :: thread-id-t
  assumes atomic-step-invariant s
  shows atomic-step-invariant (atomic-step-ev-signal tid partner s)
  proof
    from assms show ?thesis
    unfolding atomic-step-ev-signal-def atomic-step-invariant-def sp-subset-def
    by auto
  qed

3.4.2 Summary theorems on atomic step invariants

Now we are ready to show that an atomic step from the current interrupt point in any thread preserves invariants.

theorem atomic-step-preserves-invariants:
  fixes s :: state-t
  and tid :: thread-id-t
  assumes atomic-step-invariant s
  shows atomic-step-invariant (atomic-step s a)
  proof (cases a)
    case SK-IPC
      then show ?thesis unfolding atomic-step-def
      using assms atomic-ipc-preserves-invariants
      by simp
    next case (SK-EV-WAIT ev-wait-stage consume)
      then show ?thesis unfolding atomic-step-def
      proof (cases consume)
        case EV-CONSUME-ALL
          then show ?thesis unfolding atomic-step-def
          using SK-EV-WAIT assms atomic-ev-wait-all-preserves-invariants
          by (simp split: ev-wait-stage-t.splits)
        next case EV-CONSUME-ONE
          then show ?thesis unfolding atomic-step-def
          using SK-EV-WAIT assms atomic-ev-wait-one-preserves-invariants
          by (simp split: ev-wait-stage-t.splits)
        qed
      next case SK-EV-SIGNAL
        then show ?thesis unfolding atomic-step-def
        using assms atomic-ev-signal-preserves-invariants
        by (simp add: ev-signal-stage-t.splits)
      next case NONE
        then show ?thesis unfolding atomic-step-def
        using assms
        by auto
    qed

Finally, the invariants do not depend on the current thread. That is, the context switch preserves the invariants, and an atomic step that is not a context switch does not change the current thread.

theorem cswitch-preserves-invariants:
  fixes s :: state-t
  and new-current :: thread-id-t
  assumes atomic-step-invariant s
  shows atomic-step-invariant (s (current := new-current []))
  proof –
let ?s1 = s ( current := new-current )

have sp-subset s = sp-subset ?s1
  unfolding sp-subset-def by auto
from assms this show ?thesis
unfolding atomic-step-invariant-def by metis
qed

theorem atomic-step-does-not-change-current-thread:
  shows current (atomic-step s ipt) = current s

proof -
  show ?thesis
    unfolding atomic-step-def
    and atomic-step-ipc-def
    and set-object-value-def Let-def
    and atomic-step-ev-wait-one-def atomic-step-ev-wait-all-def
    and atomic-step-ev-signal-def
    by ( simp split add: int-point-t.splits ipc-stage-t.splits ipc-direction-t.splits ev-consume-t.splits ev-wait-stage-t.splits ev-signal-stage-t.splits )
qed

end

3.5 The view-partitioning equivalence relation

theory step-vpeq
  imports step step-invariants
begin

  The view consists of

  1. View of object values.

  2. View of subject-subject dynamic policy. The threads can discover the policy at runtime, e.g. by calling ipc() and observing success or failure.

  3. View of subject-object dynamic policy. The threads can discover the policy at runtime, e.g. by calling open() and observing success or failure.

definition vpeq-obj :: partition-id-t ⇒ state-t ⇒ state-t ⇒ bool where
vpeq-obj u s t ≡ ∀ obj-id . Policy.sp-spec-subj-obj u obj-id READ → (obj s) obj-id = (obj t) obj-id

definition vpeq-subj-subj :: partition-id-t ⇒ state-t ⇒ state-t ⇒ bool where
vpeq-subj-subj u s t ≡ ∀ v . ((Policy.sp-spec-subj-subj u v → sp-impl-subj-subj s u v = sp-impl-subj-subj t u v)
  ∧ (Policy.sp-spec-subj-subj v u → sp-impl-subj-subj s v u = sp-impl-subj-subj t v u))

definition vpeq-subj-obj :: partition-id-t ⇒ state-t ⇒ state-t ⇒ bool where
vpeq-subj-obj u s t ≡ ∀ ob m p1 .
  (Policy.sp-spec-subj-obj u ob m → sp-impl-subj-obj s u ob m = sp-impl-subj-obj t u ob m)
  ∧ (Policy.sp-spec-subj-obj p1 ob PROVIDE ∧ (Policy.sp-spec-subj-obj u ob READ ∨ Policy.sp-spec-subj-obj u ob WRITE) →
    sp-impl-subj-obj s p1 ob PROVIDE = sp-impl-subj-obj t p1 ob PROVIDE)

definition vpeq-local :: partition-id-t ⇒ state-t ⇒ state-t ⇒ bool where
vpeq-local \( u s t \equiv \forall \ tid . (\text{partition tid}) = u \rightarrow (\text{thread s tid}) = (\text{thread t tid}) \)

**Definition**

\[ vpeq \ u s t \equiv vpeq-obj \ u s t \land vpeq-subj-subj \ u s t \land vpeq-subj-obj \ u s t \land vpeq-local \ u s t \]

### 3.5.1 Elementary properties

**Lemma** vpeq-rel:

- shows vpeq-refl: \( vpeq \ u s s \)
- and vpeq-sym [sym]: \( vpeq \ u s t \implies vpeq \ u t s \)
- and vpeq-trans [trans]: \( \left[\left[ vpeq \ u s1 s2 \ ; vpeq \ u s2 s3 \right] \right] \implies vpeq \ u s1 s3 \)

**Unfolding** vpeq-def vpeq-obj-def vpeq-subj-subj-def vpeq-subj-obj-def vpeq-local-def

**By auto**

Auxiliary equivalence relation.

**Lemma** set-object-value-ign:

- assumes eq-obs: "Policy.sp-spec-subj-obj u x READ"
- shows vpeq u s (set-object-value x y s)

**Proof**

- from assms show ?thesis
- unfolding vpeq-def vpeq-obj-def vpeq-subj-subj-def vpeq-subj-obj-def vpeq-local-def
- by auto

**Qed**

Context-switch and fetch operations are also consistent with vpeq and locally respect everything.

**Theorem** cswitch-consistency-and-respect:

- fixes \( u \colon\colon \text{partition-id-t} \)
- and \( s \colon\colon \text{state-t} \)
- and \( \text{new-current} \colon\colon \text{thread-id-t} \)
- assumes atomic-step-invariant \( s \)
- shows vpeq u s (s (\( \text{current} := \text{new-current} \)))

**Proof**

- show ?thesis
- unfolding vpeq-def vpeq-obj-def vpeq-subj-subj-def vpeq-subj-obj-def vpeq-local-def
- by auto

**Qed**

end

### 3.6 Atomic step locally respects the information flow policy

**Theory** step-vpeq-locally-respects

**Imports** step step-invariants step-vpeq

**Begin**

The notion of locally respects is common usage. We augment it by assuming that the atomic-step-invariant holds (see [3]).

#### 3.6.1 Locally respects of atomic step functions

**Lemma** ipc-respects-policy:
assumes no : ¬ Policy.ifp (partition tid) u 
and inv : atomic-step-invariant s 
and prec: atomic-step-precondition s tid (SK-IPC dir stage partner pag) 
and ipt-case: ipt = SK-IPC dir stage partner page 
shows vpeq u s (atomic-step-ipc tid dir stage partner page s)

proof (cases stage)
\begin{itemize}
\item case PREP
\begin{itemize}
\item unfolding atomic-step-ipc-def
\item using vpeq-refl by simp
\end{itemize}
\item next case WAIT
\begin{itemize}
\item unfolding atomic-step-ipc-def
\item using vpeq-refl by simp
\end{itemize}
\item next case (BUF mypage)
\begin{itemize}
\item show ?thesis
\item proof (cases dir)
\item case RECV
\begin{itemize}
\item unfolding atomic-step-ipc-def
\item using vpeq-refl BUF by simp
\end{itemize}
\item next case SEND
\begin{itemize}
\item have Policy.sp-spec-subj-subj (partition tid) (partition partner)
\item and Policy.sp-spec-subj-obj (partition partner) (PAGE mypage) WRITE
\item using BUF SEND inv prec ipt-case
\item unfolding atomic-step-invariant-def sp-subset-def
\item unfolding atomic-step-precondition-def ipc-precondition-def opposite-ipc-direction-def
\item by auto
\item hence ¬ Policy.sp-spec-subj-obj u (PAGE mypage) READ
\item using no Policy-properties.ifp-compatible-with-ipc
\item by auto
\item thus ?thesis
\item using BUF SEND assms
\item unfolding atomic-step-ipc-def set-object-value-def
\item unfolding vpeq-def vpeq-obj-def vpeq-subj-obj-def vpeq-subj-subj-def vpeq-local-def
\item by auto
\item qed
\end{itemize}
\item qed
\end{itemize}

lemma ev-signal-respects-policy:
assumes no : ¬ Policy.ifp (partition tid) u 
and inv : atomic-step-invariant s 
and prec: atomic-step-precondition s tid (SK-EV-SIGNAL EV-SIGNAL-FINISH partner) 
and ipt-case: ipt = SK-EV-SIGNAL EV-SIGNAL-FINISH partner 
shows vpeq u s (atomic-step-ev-signal tid partner s)

proof =
\begin{itemize}
\item from inv no have ¬ sp-impl-subj-subj s (partition tid) u
\item unfolding Policy.ifp-def atomic-step-invariant-def sp-subset-def
\item by auto
\item with prec have 1 (partition partner) \neq u
\item unfolding atomic-step-precondition-def ev-signal-precondition-def
\item by (auto simp add: ev-signal-stage-t.splits)
\item then have 2: vpeq-local u s (atomic-step-ev-signal tid partner s)
\item unfolding vpeq-local-def atomic-step-ev-signal-def
\item by simp
\end{itemize}
lemma ev-wait-all-respects-policy:
assumes noː ¬ Policy.ifp (partition tid) u
  and invː atomic-step-invariant s
  and precː atomic-step-precondition s tid ipt
  and ipt-caseː ipt = SK-EV-WAIT ev-wait-stage EV-CONSUME-ALL
shows vpeq u s (atomic-step-ev-wait-all tid s)
proof –
from assms have 1ː (partition tid) \not \in u
unfolding Policy.ifp-def
by simp
then have 2ː vpeq-local u s (atomic-step-ev-wait-all tid s)
unfolding vpeq-local-def atomic-step-ev-wait-all-def
by simp
have 3ː vpeq-obj u s (atomic-step-ev-wait-all tid s)
unfolding vpeq-obj-def atomic-step-ev-wait-all-def
by simp
have 4ː vpeq-subj-subj u s (atomic-step-ev-wait-all tid s)
unfolding vpeq-subj-subj-def atomic-step-ev-wait-all-def
by simp
have 5ː vpeq-subj-obj u s (atomic-step-ev-wait-all tid s)
unfolding vpeq-subj-obj-def atomic-step-ev-wait-all-def
by simp
with 2 3 4 5 show ?thesis
unfolding vpeq-def
by simp
qed

lemma ev-wait-one-respects-policy:
assumes noː ¬ Policy.ifp (partition tid) u
  and invː atomic-step-invariant s
  and precː atomic-step-precondition s tid ipt
  and ipt-caseː ipt = SK-EV-WAIT ev-wait-stage EV-CONSUME-ONE
shows vpeq u s (atomic-step-ev-wait-one tid s)
proof –
from assms have 1ː (partition tid) \not \in u
unfolding Policy.ifp-def
by simp
then have 2ː vpeq-local u s (atomic-step-ev-wait-one tid s)
unfolding vpeq-local-def atomic-step-ev-wait-one-def
by simp
have 3ː vpeq-obj u s (atomic-step-ev-wait-one tid s)
unfolding vpeq-obj-def atomic-step-ev-wait-one-def
by simp
have 4:vpeq-subj-subj u s (atomic-step-ev-wait-one tid s)
unfolding vpeq-subj-subj-def atomic-step-ev-wait-one-def
by simp
have 5:vpeq-subj-obj u s (atomic-step-ev-wait-one tid s)
unfolding vpeq-subj-obj-def atomic-step-ev-wait-one-def
by simp
with 2 3 4 5 show ?thesis
unfolding vpeq-def
by simp
qed

3.6.2 Summary theorems on view-partitioning locally respects

Atomic step locally respects the information flow policy (ifp). The policy ifp is not necessarily the same as sp_spec_subj_subj.

theorem atomic-step-respects-policy:
assumes no oc ¬ Policy.ifp (partition (current s)) u
and inv: atomic-step-invariant s
and prec: atomic-step-precondition s (current s) ipt
shows vpeq u s (atomic-step s ipt)
proof −
show ?thesis
using assms ipc-respects-policy vpeq-refl
ev-signal-respects-policy ev-wait-one-respects-policy
ev-wait-all-respects-policy
unfolding atomic-step-def
by (auto split add: int-point-t splits ev-consume-t splits ev-wait-stage-t splits ev-signal-stage-t splits)
qed

end

3.7 Weak step consistency

theory step-vpeq-weakly-step-consistent
imports step step-invariants step-vpeq
begin

The notion of weak step consistency is common usage. We augment it by assuming that the atomic-step-invariant holds (see [3]).

3.7.1 Weak step consistency of auxiliary functions

lemma ipc-precondition-weakly-step-consistent:
assumes eq-tid: vpeq (partition tid) s1 s2
and inv1: atomic-step-invariant s1
and inv2: atomic-step-invariant s2
shows ipc-precondition tid dir partner page s1 = ipc-precondition tid dir partner page s2
proof −
let ?sender = case dir of SEND ⇒ tid | RECV ⇒ partner
let ?receiver = case dir of SEND ⇒ partner | RECV ⇒ tid
let ?local-access-mode = case dir of SEND ⇒ READ | RECV ⇒ WRITE
let ?A = sp-impl-subj-subj s1 (partition ?sender) (partition ?receiver)
= sp-impl-subj-subj s2 (partition ?sender) (partition ?receiver)
let ?B = sp-impl-subj-obj s1 (partition tid) (PAGE page) ?local-access-mode
= sp-impl-subj-obj s2 (partition tid) (PAGE page) ?local-access-mode
D31.1 – Formal specification of a generic MILS separation kernel

have A :: ?A
proof (cases Policy.sp-spec-subj-subj (partition ?sender) (partition ?receiver))
  case True
  thus ?A
    using eq-tid unfolding vpeq-def vpeq-subj-subj-def
    by (simp split add: ipc-direction-t.splits)
  next case False
  have sp-subset s1 and sp-subset s2
    using inv1 inv2 unfolding atomic-step-invariant-def sp-subset-def by auto
  hence ~ sp-impl-subj-subj s1 (partition ?sender) (partition ?receiver)
    and ~ sp-impl-subj-subj s2 (partition ?sender) (partition ?receiver)
    using False unfolding sp-subset-def by auto
  thus ?A by auto
qed

have B :: ?B
proof (cases Policy.sp-spec-subj-obj (partition tid) (PAGE page) ?local-access-mode)
  case True
  thus ?B
    using eq-tid unfolding vpeq-def vpeq-subj-obj-def
    by (simp split add: ipc-direction-t.splits)
  next case False
  have sp-subset s1 and sp-subset s2
    using inv1 inv2 unfolding atomic-step-invariant-def sp-subset-def by auto
  hence ~ sp-impl-subj-obj s1 (partition tid) (PAGE page) ?local-access-mode
    and ~ sp-impl-subj-obj s2 (partition tid) (PAGE page) ?local-access-mode
    using False unfolding sp-subset-def by auto
  thus ?B by auto
qed

show ?thesis using A B unfolding ipc-precondition-def by auto
qed

lemma ev-signal-precondition-weakly-step-consistent:
assumes eq-tid :: vpeq (partition tid) s1 s2
  and inv1 :: atomic-step-invariant s1
  and inv2 :: atomic-step-invariant s2
shows ev-signal-precondition tid partner s1 = ev-signal-precondition tid partner s2
proof –
let ?A = sp-impl-subj-subj s1 (partition tid) (partition partner)
  = sp-impl-subj-subj s2 (partition tid) (partition partner)
have A :: ?A
proof (cases Policy.sp-spec-subj-subj (partition tid) (partition partner))
  case True
  thus ?A
    using eq-tid unfolding vpeq-def vpeq-subj-subj-def
    by (simp split add: ipc-direction-t.splits)
  next case False
  have sp-subset s1 and sp-subset s2
    using inv1 inv2 unfolding atomic-step-invariant-def sp-subset-def by auto
  hence ~ sp-impl-subj-subj s1 (partition tid) (partition partner)
    and ~ sp-impl-subj-subj s2 (partition tid) (partition partner)
    using False unfolding sp-subset-def by auto
  thus ?A by auto
qed

show ?thesis using A unfolding ev-signal-precondition-def by auto
qed
lemma set-object-value-consistent:
assumes eq-obs: vpeq u s1 s2
shows vpeq u (set-object-value x y s1) (set-object-value x y s2)
proof –
let ?s1' = set-object-value x y s1 and ?s2' = set-object-value x y s2
have E1: vpeq-obj u ?s1' ?s2'
proof –
{ fix x'
  assume 1: Policy.sp-spec-subj-obj u x' READ
  have obj ?s1' x' = obj ?s2' x' proof (cases x = x')
    case True
    thus obj ?s1' x' = obj ?s2' x' unfolding set-object-value-def by auto
    next case False
    hence 2: obj ?s1' x' = obj s1 x'
        and 3: obj ?s2' x' = obj s2 x'
        unfolding set-object-value-def by auto
        have 4: obj s1 x' = obj s2 x'
        using 1 eq-obs unfolding vpeq-obj-def by auto
        from 2 3 4 show obj ?s1' x' = obj ?s2' x'
            by simp
    qed }
thus vpeq-obj u ?s1' ?s2' unfolding vpeq-obj-def by auto
qed
have E4: vpeq-subj-subj u ?s1' ?s2'
proof –
  have sp-impl-subj-subj ?s1' = sp-impl-subj-subj s1
    and sp-impl-subj-subj ?s2' = sp-impl-subj-subj s2
    unfolding set-object-value-def by auto
    thus vpeq-subj-subj u ?s1' ?s2'
        using eq-obs unfolding vpeq-def vpeq-subj-subj-def by auto
    qed
have E5: vpeq-subj-obj u ?s1' ?s2'
proof –
  have sp-impl-subj-obj ?s1' = sp-impl-subj-obj s1
    and sp-impl-subj-obj ?s2' = sp-impl-subj-obj s2
    unfolding set-object-value-def by auto
    thus vpeq-subj-obj u ?s1' ?s2'
        using eq-obs unfolding vpeq-def vpeq-subj-obj-def by auto
    qed
from eq-obs have E6: vpeq-local u ?s1' ?s2'
unfolding vpeq-def vpeq-local-def set-object-value-def
by simp
from E1 E4 E5 E6
show ?thesis unfolding vpeq-def
by auto
qed

3.7.2 Weak step consistency of atomic step functions

lemma ipc-weakly-step-consistent:
assumes eq-obs: vpeq u s1 s2
  and eq-act: vpeq (partition tid) s1 s2
  and inv1: atomic-step-invariant s1
  and inv2: atomic-step-invariant s2
  and prec1: atomic-step-precondition s1 tid ipt
  and prec2: atomic-step-precondition s1 tid ipt
  and ipt-case: ipt = SK-IPC dir stage partner page
shows $vpeq \ u$

\[ (atomic-step-ipc \ tid \ stage \ partner \ page \ s1) \]
\[ (atomic-step-ipc \ tid \ stage \ partner \ page \ s2) \]

proof –

have $\forall \ \text{mypage} \ . \ \left[ \ dir = \text{SEND}; \ stage = \text{BUF mypage} \right] \implies \ ?thesis$

proof –

fix mypage

assume $dir-send: \ dir = \text{SEND}$
assume $stage-buf: \ stage = \text{BUF mypage}$

have $\text{Policy.sp-spec-subj-obj \ (partition tid) \ (PAGE \ page) \ READ}$
using inv1 prec1 dir-send stage-buf ipt-case

unfolding $\text{atomic-step-invariant-def sp-subset-def}$
unfolding $\text{atomic-step-precondition-def ipc-precondition-def opposite-ipc-direction-def}$

by auto

hence $obj \ s1 \ (PAGE \ page) = \ obj \ s2 \ (PAGE \ page)$

using eq-act unfolding $vpeq-def \ vpeq-obj-def \ vpeq-local-def$
by auto

thus $vpeq \ u$

\[ (atomic-step-ipc \ tid \ stage \ partner \ page \ s1) \]
\[ (atomic-step-ipc \ tid \ stage \ partner \ page \ s2) \]

using dir-send stage-buf eq-obs set-object-value-consistent

unfolding $\text{atomic-step-ipc-def}$

by auto

qed

thus $?thesis$

using eq-obs unfolding $\text{atomic-step-ipc-def}$

by (cases stage, auto, cases dir, auto)

qed

lemma $ev\text{-wait-one-weakly-step-consistent}$:

assumes $eq-obs: \ vpeq \ u \ s1 \ s2$
and $eq-act: \ vpeq \ (\text{partition tid}) \ s1 \ s2$
and $inv1: \ atomic-step-invariant \ s1$
and $inv2: \ atomic-step-invariant \ s2$
and $prec1: \ atomic-step-precondition \ s1 \ (\text{current s1}) \ ipt$
and $prec2: \ atomic-step-precondition \ s1 \ (\text{current s1}) \ ipt$

shows $vpeq \ u$

\[ (atomic-step-ev\text{-wait-one} \ tid \ s1) \]
\[ (atomic-step-ev\text{-wait-one} \ tid \ s2) \]

using assms

unfolding $vpeq-def \ vpeq-subj-subj-def \ vpeq-obj-def \ vpeq-subj-obj-def \ vpeq-local-def$

atomic-step-ev\text{-wait-one-def}$

by simp

lemma $ev\text{-wait-all-weakly-step-consistent}$:

assumes $eq-obs: \ vpeq \ u \ s1 \ s2$
and $eq-act: \ vpeq \ (\text{partition tid}) \ s1 \ s2$
and $inv1: \ atomic-step-invariant \ s1$
and $inv2: \ atomic-step-invariant \ s2$
and $prec1: \ atomic-step-precondition \ s1 \ (\text{current s1}) \ ipt$
and $prec2: \ atomic-step-precondition \ s1 \ (\text{current s1}) \ ipt$

shows $vpeq \ u$

\[ (atomic-step-ev\text{-wait-all} \ tid \ s1) \]
\[ (atomic-step-ev\text{-wait-all} \ tid \ s2) \]

using assms

unfolding $vpeq-def \ vpeq-subj-subj-def \ vpeq-obj-def \ vpeq-subj-obj-def \ vpeq-local-def$

atomic-step-ev\text{-wait-all-def}$
**Lemma**  ev-signal-weakly-step-consistent:

**Assumes**

- `eq-obs` \(vpeq u s1 s2\)
- `eq-act` \(vpeq (\text{partition tid}) s1 s2\)
- `inv1` : atomic-step-invariant \(s1\)
- `inv2` : atomic-step-invariant \(s2\)
- `prec1` : atomic-step-precondition \(s1\) (current \(s1\)) \(ipt\)
- `prec2` : atomic-step-precondition \(s1\) (current \(s1\)) \(ipt\)

**Shows**

\(vpeq u\)
- (atomic-step-ev-signal tid partner \(s1\))
- (atomic-step-ev-signal tid partner \(s2\))

**Using** `assms`  

**Unfolding** `vpeq-def` `vpeq-subj-subj-def` `vpeq-obj-def` `vpeq-subj-obj-def` `vpeq-local-def`

**By** `simp`

The use of `extend-f` is to provide infrastructure to support use in dynamic policies, currently not used.

**Definition** `extend-f` :\( \vdash (\text{partition-id-t} \Rightarrow \text{partition-id-t} \Rightarrow \text{bool}) \Rightarrow (\text{partition-id-t} \Rightarrow \text{partition-id-t} \Rightarrow \text{bool}) \Rightarrow (\text{partition-id-t} \Rightarrow \text{partition-id-t} \Rightarrow \text{bool}) \)  

**Definition** `extend-subj-subj` :\( \vdash (\text{partition-id-t} \Rightarrow \text{partition-id-t} \Rightarrow \text{bool}) \Rightarrow \text{state-t} \Rightarrow \text{state-t} \)  

**Lemma** `extend-subj-subj-consistent`:

**Fixes** \(f \vdash \text{partition-id-t} \Rightarrow \text{partition-id-t} \Rightarrow \text{bool}\)

**Assumes** \(vpeq u s1 s2\)

**Shows** \(vpeq u (\text{extend-subj-subj} f s1) (\text{extend-subj-subj} f s2)\)

**Proof**

1. Let \(?g1 = \text{sp-impl-subj-subj} s1\) and \(?g2 = \text{sp-impl-subj-subj} s2\)
2. Have \(\forall v. \text{Policy.sp-spec-subj-subj} u v \wedge ?g1 u v = ?g2 u v\)
3. Using `assms`  
   **Unfolding** `vpeq-def` `vpeq-subj-subj-def`  
   **By** `auto`
4. Hence \(\forall v. \text{Policy.sp-spec-subj-subj} u v \wedge \text{extend-f} f ?g1 u v = \text{extend-f} f ?g2 u v\)
5. Using `assms`  
   **Unfolding** `vpeq-def` `vpeq-subj-subj-def`  
   **By** `auto`
6. Hence \(1: vpeq-subj-subj u (\text{extend-subj-subj} f s1) (\text{extend-subj-subj} f s2)\)
7. **Unfolding** `vpeq-subj-subj-def` `extend-subj-subj-def`  
   **By** `auto`
8. Have \(?v = \text{vpeq-obj} u (\text{extend-subj-subj} f s1) (\text{extend-subj-subj} f s2)\)
9. Using `assms`  
   **Unfolding** `vpeq-def` `vpeq-obj-def` `extend-subj-subj-def`  
   **By** `fastforce`
10. Have \(?v = \text{vpeq-local} u (\text{extend-subj-subj} f s1) (\text{extend-subj-subj} f s2)\)
11. Using `assms`  
   **Unfolding** `vpeq-def` `vpeq-local-def` `extend-subj-subj-def`  
   **By** `fastforce`

**From** \(1 2 3 4\) **Show** `?thesis`

**Using** `assms`  

**Unfolding** `vpeq-def`  

**By** `fast`

**Qed**

### 3.7.3 Summary theorems on view-partitioning weak step consistency

The atomic step is weakly step consistent with view partitioning. Here, the “weakness” is that we assume that the two states are vp-equivalent not only w.r.t. the observer domain \(u\), but also w.r.t. the caller domain \(\text{step.partition tid}\).

**Theorem** `atomic-step-weakly-step-consistent`:
assumes \( \text{eq-obs} : \text{vpeq } u \ s1 \ s2 \)
and \( \text{eq-act} : \text{vpeq } (\text{partition } (\text{current } s1)) \ s1 \ s2 \)
and \( \text{inv1} : \text{atomic-step-invariant } s1 \)
and \( \text{inv2} : \text{atomic-step-invariant } s2 \)
and \( \text{prec1} : \text{atomic-step-precondition } s1 \ (\text{current } s1) \ \text{ipt} \)
and \( \text{prec2} : \text{atomic-step-precondition } s2 \ (\text{current } s2) \ \text{ipt} \)
and \( \text{eq-curr} : \text{current } s1 = \text{current } s2 \)
shows \( \text{vpeq } u \ (\text{atomic-step } s1 \ \text{ipt}) \ (\text{atomic-step } s2 \ \text{ipt}) \)
proof –
show \( \text{?thesis} \)
using \( \text{assms} \)
  ipc-weakly-step-consistent
  ev-wait-all-weakly-step-consistent
  ev-wait-one-weakly-step-consistent
  ev-signal-weakly-step-consistent
  vpeq-refl \ ev-signal-stage-t.exhaust
unfolding atomic-step-def
apply cases ipt, auto
apply (simp split add: ev-consume-t.splits ev-wait-stage-t.splits)
by (simp split add: ev-signal-stage-t.splits)
qed
end

3.8 Separation kernel model

theory separation-kernel-model
  imports ..|Theories−step|step
           ..|Theories−step|step-invariants
           ..|Theories−step|step-vpeq
           ..|Theories−step|step-vpeq-locally-respects
           ..|Theories−step|step-vpeq-weakly-step-consistent
  begin

  First (Section 3.8.1) we instantiate the CISK generic model. Functions that instantiate a generic
  function of the CISK model are prefixed with an ‘r’, ‘r’ standing for ‘Rushby’; as CISK is derived
  originally from a model by Rushby [3]. For example, ‘rifp’ is the instantiation of the generic ‘ifp’.

  Later (Section 3.8.5) all CISK proof obligations are discharged, e.g., weak step consistency, output
  consistency, etc. These will be used in Section 3.9.

3.8.1 Initial state of separation kernel model

We assume that the initial state of threads and memory is given. The initial state of threads is arbitrary,
but the threads are not executing the system call. The purpose of the following definitions is to obtain
the initial state without potentially dangerous axioms. The only axioms we admit without proof are
formulated using the “consts” syntax and thus safe.

consts
  initial-current :: thread-id-t
  initial-obj :: obj-id-t ⇒ obj-t

definition \( s0 = \text{state-t where} \)
  \( s0 \equiv (\{ \text{sp-impl-subj-subj } = \text{Policy}.\text{sp-spec-subj-subj}, \)
  \text{sp-impl-subj-obj } = \text{Policy}.\text{sp-spec-subj-obj}, \)
  \text{current } = \text{initial-current}, \)
  \text{obj } = \text{initial-obj}, \)
thread = λ . ( | ev-counter = 0 | )

lemma initial-invariant:
shows atomic-step-invariant s0
proof
  have sp-subset s0
  unfolding sp-subset-def s0-def by auto
  thus ?thesis
  unfolding atomic-step-invariant-def by auto
qed

3.8.2 Types for instantiation of the generic model

To simplify formulations, we include the state invariant atomic-step-invariant in the state data type. The initial state s0 serves as witness that rstate-t is non-empty.

typedef rstate-t = { s . atomic-step-invariant s }
  using initial-invariant by auto

definition abs :: state-t ⇒ rstate-t (↑ -)
  where abs = Abs-rstate-t

definition rep :: rstate-t ⇒ state-t (↓ -)
  where rep = Rep-rstate-t

lemma rstate-invariant:
shows atomic-step-invariant (↓ s)
unfolding rep-def by (metis Rep-rstate-t mem-Collect-eq)

lemma rstate-down-up [simp]:
shows (↑↓ s) = s
unfolding rep-def abs-def using Rep-rstate-t-inverse by auto

lemma rstate-up-down [simp]:
assumes atomic-step-invariant s
shows (↓↑ s) = s
using asms Abs-rstate-t-inverse unfolding rep-def abs-def by auto

A CISK action is identified with an interrupt point.

type-synonym raction-t = int-point-t

definition rcurrent :: rstate-t ⇒ thread-id-t where
rcurrent s = current (↓ s)

definition rstep :: rstate-t ⇒ raction-t ⇒ rstate-t where
rstep s a ≡ ↑ (atomic-step (↓ s) a)

Each CISK domain is identified with a thread id.

type-synonym rdom-t = thread-id-t

The output function returns the contents of all memory accessible to the subject. The action argument of the output function is ignored.

datatype visible-obj-t = VALUE obj-t | EXCEPTION

type-synonym routput-t = page-t ⇒ visible-obj-t

definition routput-f :: rstate-t ⇒ raction-t ⇒ routput-t where
routput-f s a p ≡
  if sp-impl-subj-obj (↓ s) (partition (rcurrent s)) (PAGE p) READ then
    VALUE (obj (↓ s) (PAGE p))
  else
EXCEPTION

The precondition for the generic model. Note that atomic-step-invariant is already part of the state.

**definition** rprecondition :: rstate-t ⇒ rdom-t ⇒ raction-t ⇒ bool where
rprecondition s d a ≡ atomic-step-precondition (↓s) d a

**abbreviation** rinvariant
where rinvariant s ≡ True — The invariant is already in the state type.

Translate view-partitioning and interaction-allowed relations.

**definition** rvpeq :: rdom-t ⇒ rstate-t ⇒ rstate-t ⇒ bool where
rvpeq u s1 s2 ≡ vpeq (partition u) (↓s1) (↓s2)

**definition** rifp :: rdom-t ⇒ rdom-t ⇒ bool where
rifp u v = Policy.ifp (partition u) (partition v)

3.8.3 Possible action sequences

An SK-IPC consists of three atomic actions PREP, WAIT and BUF with the same parameters.

**definition** is-SK-IPC :: raction-t list ⇒ bool
where is-SK-IPC aseq ≡ ∃ dir partner page .
    aseq = [SK-IPC dir PREP partner page, SK-IPC dir WAIT partner page, SK-IPC dir (BUF (SOME page’). True)) partner page]

An SK-EV-WAIT consists of three atomic actions, one for each of the stages EV-PREP, EV-WAIT and EV-FINISH with the same parameters.

**definition** is-SK-EV-WAIT :: raction-t list ⇒ bool
where is-SK-EV-WAIT aseq ≡ ∃ consume .
    aseq = [SK-EV-WAIT EV-PREP consume ,
             SK-EV-WAIT EV-WAIT consume ,
             SK-EV-WAIT EV-FINISH consume ]

An SK-EV-SIGNAL consists of two atomic actions, one for each of the stages EV-SIGNAL-PREP and EV-SIGNAL-FINISH with the same parameters.

**definition** is-SK-EV-SIGNAL :: raction-t list ⇒ bool
where is-SK-EV-SIGNAL aseq ≡ ∃ partner .
    aseq = [SK-EV-SIGNAL EV-SIGNAL-PREP partner ,
             SK-EV-SIGNAL EV-SIGNAL-FINISH partner]

The complete attack surface consists of IPC calls, events, and noops.

**definition** rAS-set :: raction-t list set
where rAS-set ≡ { aseq . is-SK-IPC aseq ∨ is-SK-EV-WAIT aseq ∨ is-SK-EV-SIGNAL aseq } ∪ {[]}

3.8.4 Control

When are actions aborting, and when are actions waiting. We do not currently use the set-error-code function yet.

**abbreviation** raborting
where raborting s ≡ aborting (↓s)

**abbreviation** rwaiting
where rwaiting s ≡ waiting (↓s)
definition rset-error-code :: rstate-t ⇒ raction-t ⇒ rstate-t
where rset-error-code s a ≡ s

Returns the set of threads that are involved in a certain action. For example, for an IPC call, the WAIT stage synchronizes with the partner. This partner is involved in that action.

definition rkinvolved :: int-point-t ⇒ rdom-t set
where rkinvolved a ≡
case a of SK-IPC dir WAIT partner page ⇒ {partner}
| SK-EV-SIGNAL EV-SIGNAL-FINISH partner ⇒ {partner}
| - ⇒ {}

abbreviation rinvolved :: int-point-t option ⇒ rdom-t set
where rinvolved ≡ Kernel.involved rkinvolved

3.8.5 Discharging the proof obligations

lemma inst-vpeq-rel:
  shows rvpeq-refl: rvpeq u s s
  and rvpeq-sym: rvpeq u s1 s2 ⇒ rvpeq u s2 s1
  and rvpeq-trans: [[ rvpeq u s1 s2 ; rvpeq u s2 s3 ]] ⇒ rvpeq u s1 s3

unfolding rvpeq-def using vpeq-rel by metis+

lemma inst-ifp-refl:
  shows ∀ u. rifp u u

unfolding rifp-def using Policy-properties.ifp-reflexive by fast

lemma inst-step-atomicity [simp]:
  shows ∀ s a. rcurrent (rstep s a) = rcurrent s

unfolding rstep-def rcurrent-def
by auto

lemma inst-weakly-step-consistent:
  assumes rvpeq u s t
    and rvpeq (rcurrent s) s t
    and rcurrent s = rcurrent t
    and rprecondition s (rcurrent s) a
    and rprecondition t (rcurrent t) a
  shows rvpeq u (rstep s a) (rstep t a)

using assms atomic-step-weakly-step-consistent rstate-invariant atomic-step-preserves-invariants

unfolding rcurrent-def rstep-def rvpeq-def rprecondition-def
by auto

lemma inst-local-respect:
  assumes not-ifp: ¬rifp (rcurrent s) u
    and prec: rprecondition s (rcurrent s) a
  shows rvpeq u s (rstep s a)

using assms atomic-step-respects-policy rstate-invariant atomic-step-preserves-invariants

unfolding rifp-def rprecondition-def rvpeq-def rstep-def rcurrent-def
by auto
lemma inst-output-consistency:
  assumes rvpeq: rvpeq (rcurrent s) s t
  and current-eq: rcurrent s = rcurrent t
  shows routput-f s a = routput-f t a
proof:
  have ∀ a s t. rvpeq (rcurrent s) s t ∧ rcurrent s = rcurrent t → routput-f s a = routput-f t a
  proof
    { fix a :: raction-t
      fix s t :: rstate-t
      fix p :: page-t
      assume 1: rvpeq (rcurrent s) s t
      and 2: rcurrent s = rcurrent t
      let ?part = partition (rcurrent s)
      have routput-f s a p = routput-f t a p
      proof
        (cases Policy; sp-spec-subj-obj ?part (PAGE p) READ
         rule: case-split [case-names Allowed Denied])
        case Allowed
        have 5: obj (↓s) (PAGE p) = obj (↓t) (PAGE p)
        using 1 Allowed unfolding rvpeq-def vpeq-def vpeq-obj-def by auto
        have 6: sp-impl-subj-obj (↓s) ?part (PAGE p) READ = sp-impl-subj-obj (↓t) ?part (PAGE p) READ
        using 2 Allowed unfolding rvpeq-def vpeq-def vpeq-subj-obj-def by auto
        show routput-f s a p = routput-f t a p
        unfolding routput-f-def using 2 5 6 by auto
        next case Denied
        hence sp-impl-subj-obj (↓s) ?part (PAGE p) READ = False
        and sp-impl-subj-obj (↓t) ?part (PAGE p) READ = False
        using rstate-invariant unfolding atomic-step-invariant-def sp-subset-def by auto
        thus routput-f s a p = routput-f t a p
        using 2 unfolding routput-f-def by simp
        qed
      } thus ∀ a s t. rvpeq (rcurrent s) s t ∧ rcurrent s = rcurrent t → routput-f s a = routput-f t a
      by auto
      qed
    thus ?thesis using assms by auto
  qed

lemma inst-cswitch-independent-of-state:
  assumes rcurrent s = rcurrent t
  shows rcurrent (rcswitch n s) = rcurrent (rcswitch n t)
  using rstate-invariant cswitch-preserves-invariants unfolding rcurrent-def rcswitch-def by simp

lemma inst-cswitch-consistency:
  assumes rvpeq u s t
  shows rvpeq u (rcswitch n s) (rcswitch n t)
proof
  have 1: vpeq (partition u) (↓s) ↓(rcswitch n s)
  using rstate-invariant cswitch-consistency-and-respect cswitch-preserves-invariants
  unfolding rcswitch-def by auto
  have 2: vpeq (partition u) (↓t) ↓(rcswitch n t)
  using rstate-invariant cswitch-consistency-and-respect cswitch-preserves-invariants
  unfolding rcswitch-def by auto
For the \texttt{PREP} stage (the first stage of the IPC action sequence) the precondition is True.

\begin{lstlisting}[language=Isabelle]
lemma prec-first-IPC-action:
assumes is-SK-IPC aseq
shows rprecondition s d (hd aseq)
using assms
unfolding is-SK-IPC-def rprecondition-def atomic-step-precondition-def
by auto
\end{lstlisting}

For the first stage of the \texttt{EV-WAIT} action sequence the precondition is True.

\begin{lstlisting}[language=Isabelle]
lemma prec-first-EV-WAIT-action:
assumes is-SK-EV-WAIT aseq
shows rprecondition s d (hd aseq)
using assms
unfolding is-SK-EV-WAIT-def rprecondition-def atomic-step-precondition-def
by auto
\end{lstlisting}

For the first stage of the \texttt{EV-SIGNAL} action sequence the precondition is True.

\begin{lstlisting}[language=Isabelle]
lemma prec-first-EV-SIGNAL-action:
assumes is-SK-EV-SIGNAL aseq
shows rprecondition s d (hd aseq)
using assms
unfolding is-SK-EV-SIGNAL-def rprecondition-def atomic-step-precondition-def
by auto
\end{lstlisting}

When not waiting or aborting, the precondition is “1-step inductive”, that is at all times the precondition holds initially (for the first step of an action sequence) and after doing one step.

\begin{lstlisting}[language=Isabelle]
lemma prec-after-IPC-step:
assumes prec: rprecondition s (rcurrent s) (aseq ! n)
and n-bound: Suc n < length aseq
and IPC: is-SK-IPC aseq
and not-aborting: ~raborting s (rcurrent s) (aseq ! n)
and not-waiting: ~rwaiting s (rcurrent s) (aseq ! n)
shows rprecondition (rstep s (aseq ! n)) (rcurrent s) (aseq ! Suc n)
proof
{  
  fix dir partner page
  let ?page' = (SOME page'. True)
  assume IPC: aseq = [SK-IPC dir PREP partner page,SK-IPC dir WAIT partner page,SK-IPC dir (BUF ?page') partner page]
  {    
    assume 0: n=0
    from 0 IPC prec not-aborting
    have ?thesis
    unfolding rprecondition-def atomic-step-precondition-def rstep-def rcurrent-def atomic-step-def atomic-step-ipc-def
    aborting-def
    by(auto)
  }
  moreover
  {    
    assume 1: n=1
    from 1 IPC prec not-waiting
    have ?thesis
    unfolding rprecondition-def atomic-step-precondition-def rstep-def rcurrent-def atomic-step-def atomic-step-ipc-def
    waiting-def
  }
}\end{lstlisting}
by(auto)
}
moreover
from IPC
have length aseq = 3
by auto
ultimately
have ?thesis
using n-bound
by arith
}
thus ?thesis
using IPC
unfolding is-SK-IPC-def
by(auto)
qed

When not waiting or aborting, the precondition is 1-step inductive.

lemma prec-after-EV-WAIT-step:
assumes prec: rprecondition s (rcurrent s) (aseq ! n)
and n-bound: Suc n < length aseq
and IPC: is-SK-EV-WAIT aseq
and not-aborting: ¬raborting s (rcurrent s) (aseq ! n)
and not-waiting: ¬rwaiting s (rcurrent s) (aseq ! n)
shows rprecondition (rstep s (aseq ! n)) (rcurrent s) (aseq ! Suc n)
proof{
  { fix consume

    assume WAIT: aseq = [SK-EV-WAIT EV-PREP consume,
      SK-EV-WAIT EV-WAIT consume,
      SK-EV-WAIT EV-FINISH consume]

    { assume 0: n=0
      from 0 WAIT prec not-aborting
      have ?thesis
      unfolding rprecondition-def atomic-step-precondition-def
      by(auto)
    }
    moreover
    {
      assume 1: n=1
      from 1 WAIT prec not-waiting
      have ?thesis
      unfolding rprecondition-def atomic-step-precondition-def
      by(auto)
    }
    moreover
    from WAIT
    have length aseq = 3
    by auto
    ultimately
    have ?thesis
    using n-bound
    by arith
  }
  thus ?thesis
  using assms
When not waiting or aborting, the precondition is 1-step inductive.

**lemma** prec-after-EV-SIGNAL-step:

**assumes**
\[
\begin{align*}
&\text{prec} : \text{rprecondition } s (\text{rcurrent } s) (\text{aseq } ! n) \\
&\text{and } n\text{-bound}: \text{Suc } n < \text{length } aseq \\
&\text{and } \text{SIGNAL}: \text{is-SK-EV-SIGNAL } aseq \\
&\text{and } \text{not-aborting}: \neg \text{raborting } s (\text{rcurrent } s) (\text{aseq } ! n) \\
&\text{and } \text{not-waiting}: \neg \text{rwaiting } s (\text{rcurrent } s) (\text{aseq } ! n)
\end{align*}
\]

**shows**
\[
\text{rprecondition} (\text{rstep } s (\text{aseq } ! n)) (\text{rcurrent } s) (\text{aseq } ! \text{Suc } n)
\]

**proof**

\[
\begin{align*}
&\text{fix } \text{partner} \\
&\text{assume } \text{SIGNAL1}: \text{aseq } = [\text{SK-EV-SIGNAL } \text{EV-SIGNAL-PREP } \text{partner}, \\
&\text{SK-EV-SIGNAL } \text{EV-SIGNAL-FINISH } \text{partner}]
\end{align*}
\]

\[
\begin{align*}
&\text{from } \text{SIGNAL1} \text{ have } \text{?thesis} \\
&\text{unfolding } \text{rprecondition-def } \text{atomic-step-precondition-def } \text{ev-signal-precondition-def} \\
&\text{aborting-def } \text{rstep-def } \text{atomic-step-def} \\
&\text{by } \text{auto}
\end{align*}
\]

**moreover**

\[
\begin{align*}
&\text{from } \text{SIGNAL1} \text{ have } \text{length } \text{aseq } = 2 \\
&\text{by } \text{auto}
\end{align*}
\]

**ultimately**

\[
\begin{align*}
&\text{have } \text{?thesis} \\
&\text{using } \text{n\text{-bound}} \\
&\text{by } \text{arith}
\end{align*}
\]

**thus** \text{?thesis}

**using** \text{assms}

**unfolding** \text{is-SK-EV-SIGNAL-def}

**by** \text{auto}

**qed**

**lemma** on-set-object-value:

**shows**
\[
\text{sp-impl-subj-subj (set-object-value ob val s)} = \text{sp-impl-subj-subj s}
\]

\[
\text{and } \text{sp-impl-subj-obj (set-object-value ob val s)} = \text{sp-impl-subj-obj s}
\]

**unfolding** \text{set-object-value-def} \text{ apply simp+ done}

**lemma** prec-IPC-dom-independent:

**assumes**
\[
\begin{align*}
&\text{current } s \neq d \\
&\text{and } \text{atomic-step-invariant } s \\
&\text{and } \text{atomic-step-precondition } s d a
\end{align*}
\]

**shows**
\[
\text{atomic-step-precondition (atomic-step-ipc (current ) dir stage partner page ) d a}
\]

**using** \text{assms on-set-object-value}

**unfolding** \text{atomic-step-precondition-def } \text{atomic-step-ipc-def } \text{ipc-precondition-def}

\[
\text{ev-signal-precondition-def set-object-value-def}
\]

**by** (\text{auto split add: int-point-t.splits ipc-stage-t.splits ipc-direction-t.splits}

\[
\text{ev-consume-t.splits ev-wait-stage-t.splits ev-signal-stage-t.splits})
\]

**lemma** prec-ev-signal-dom-independent:

**assumes**
\[
\begin{align*}
&\text{current } s \neq d \\
&\text{and } \text{atomic-step-invariant } s
\end{align*}
\]

**unfolding** \text{is-SK-EV-WAIT-def}

**by** \text{auto}

**qed**
and atomic-step-precondition s d a
shows atomic-step-precondition (atomic-step-ev-signal (current s) partner s) d a
using assms on-set-object-value
unfolding atomic-step-precondition-def atomic-step-ev-signal-def ipc-precondition-def
  ev-signal-precondition-def set-object-value-def
  by (auto split add: int-point-t.splits ipc-stage-t.splits ipc-direction-t.splits
  ev-consume-t.splits ev-wait-stage-t.splits ev-signal-stage-t.splits)

lemma prec-ev-wait-one-dom-independent:
  assumes current s /\ d
  and atomic-step-invariant s
  and atomic-step-precondition s d a
  shows atomic-step-precondition (atomic-step-ev-wait-one (current s) s) d a
  using assms on-set-object-value
  unfolding atomic-step-precondition-def atomic-step-ev-wait-one-def ipc-precondition-def
  ev-signal-precondition-def set-object-value-def
  by (auto split add: int-point-t.splits ipc-stage-t.splits ipc-direction-t.splits
  ev-consume-t.splits ev-wait-stage-t.splits ev-signal-stage-t.splits)

lemma prec-ev-wait-all-dom-independent:
  assumes current s /\ d
  and atomic-step-invariant s
  and atomic-step-precondition s d a
  shows atomic-step-precondition (atomic-step-ev-wait-all (current s) s) d a
  using assms on-set-object-value
  unfolding atomic-step-precondition-def atomic-step-ev-wait-all-def ipc-precondition-def
  ev-signal-precondition-def set-object-value-def
  by (auto split add: int-point-t.splits ipc-stage-t.splits ipc-direction-t.splits
  ev-consume-t.splits ev-wait-stage-t.splits ev-signal-stage-t.splits)

lemma prec-dom-independent:
  shows \( \forall s \ d \ a\ a'. rcurrent s /\ d \land rprecondition s d a \rightarrow rprecondition (rstep s a') d a \)
  using atomic-step-preserves-invariants
  rstate-invariant prec-IPC-dom-independent prec-ev-signal-dom-independent
  prec-ev-wait-all-dom-independent prec-ev-wait-one-dom-independent
  unfolding rcurrent-def rprecondition-def atomic-step-precondition-def
  by (auto split add: int-point-t.splits ev-consume-t.splits ev-wait-stage-t.splits ev-signal-stage-t.splits)

lemma ipc-precondition-after-cswitch [simp]:
  shows ipc-precondition d dir partner page ((( s)[(current := new-current)])
  = ipc-precondition d dir partner page ( s)
  using assms
  unfolding ipc-precondition-def
  by (auto split add: ipc-direction-t.splits)

lemma precondition-after-cswitch:
  shows \( \forall s \ d \ n \ a. rprecondition s d a \rightarrow rprecondition (rcswitch n s) d a \)
  using cswitch-preserves-invariants rstate-invariant
  unfolding rprecondition-def rcswitch-def atomic-step-precondition-def
  ev-signal-precondition-def
  by (auto split add: int-point-t.splits ipc-stage-t.splits ev-signal-stage-t.splits)

lemma aborting-switch-independent:
  shows \( \forall n \ s. raborting (rcswitch n s) = raborting s \)
  proof
  { fix n s
D31.1 – Formal specification of a generic MILS separation kernel

\{ 
  \text{fix } tid a 
  \hfill \text{have } raborting \ (rcswitch \ n \ s) \ tid \ a = raborting \ s \ tid \ a 
  \hfill \text{using rstate-invariant cswitch-preserves-invariants ev-signal-precondition-weakly-step-consistent cswitch-consistency-and-respect}
  \hfill \text{unfolding aborting-def rcswitch-def}
  \hfill \text{apply (auto split add: int-point-t.splits ipc-stage-t.splits ev-signal-stage-t.splits)}
  \hfill \text{ev-signal-stage-t.splits)}
  \hfill \text{apply (metis (full-types))}
  \hfill \text{by blast}
\} 

\hfill \text{hence } raborting \ (rcswitch \ n \ s) = raborting \ s \ \text{by auto}
\}

\text{thus } ?\text{thesis by auto}

\text{qed}

\text{lemma } \text{waiting-switch-independent:}
\text{shows } \forall \ n \ s. \ rwaiting \ (rcswitch \ n \ s) = rwaiting \ s
\text{proof --}
\{ 
  \text{fix } n \ s 
  \{ 
  \text{fix } tid a 
  \hfill \text{have } rwaiting \ (rcswitch \ n \ s) \ tid \ a = rwaiting \ s \ tid \ a 
  \hfill \text{using rstate-invariant cswitch-preserves-invariants}
  \hfill \text{unfolding waiting-def rcswitch-def}
  \hfill \text{by(auto split add: int-point-t.splits ipc-stage-t.splits ev-signal-stage-t.splits)}
  \} 

\hfill \text{hence } rwaiting \ (rcswitch \ n \ s) = rwaiting \ s \ \text{by auto}
\}

\text{thus } ?\text{thesis by auto}

\text{qed}

\text{lemma } \text{aborting-after-IPC-step:}
\text{assumes } d1 \neq d2
\text{shows } aborting \ \text{(atomic-step-ipc } d1 \ \text{dir stage partner page s)} \ d2 \ a = aborting \ s \ d2 \ a
\text{unfolding atomic-step-ipc-def aborting-def set-object-value-def ipc-precondition-def ev-signal-precondition-def}
\text{by(auto split add: int-point-t.splits ipc-stage-t.splits ipc-direction-t.splits ev-signal-stage-t.splits)}

\text{lemma } \text{waiting-after-IPC-step:}
\text{assumes } d1 \neq d2
\text{shows } waiting \ \text{(atomic-step-ipc } d1 \ \text{dir stage partner page s)} \ d2 \ a = waiting \ s \ d2 \ a
\text{unfolding atomic-step-ipc-def waiting-def set-object-value-def ipc-precondition-def}
\text{by(auto split add: int-point-t.splits ipc-stage-t.splits ipc-direction-t.splits ev-signal-stage-t.splits)}

\text{lemma } \text{raborting-consistent:}
\text{shows } \forall \ s \ t \ u. \ rvpeq \ u \ s \ t \ \rightarrow \ \text{raborting} \ u = \text{raborting} \ t \ u
\text{proof --}
\{ 
  \text{fix } s \ t \ u 
  \hfill \text{assume vpeq \ rvpeq \ u \ s \ t}
  \{ 
  \text{fix } a 
\}
from vpeq ipc-precondition-weakly-step-consistent rstate-invariant
have \( \land \text{tid dir partner page} \cdot \text{ipc-precondition} \land \text{dir partner page} (\downarrow s) = \text{ipc-precondition} \land \text{dir partner page} (\downarrow t) \)
unfolding rvpeq-def
by auto
with vpeq rstate-invariant have raborting \( s a = \text{raborting} \ t u a \)
apply (auto split add: int-point-t.splits ipc-stage-t.splits ev-signal-stage-t.splits)
by blast
}
hence raborting \( s u = \text{raborting} \ t u \) by auto
}
thus \(?\)thesis by auto
qed

lemma aborting-dom-independent:
assumes rcurrent \( s \notin d \)
shows raborting (rstep \( s a \)) \( d a' = \text{raborting} \ s d a' \)
proof –
have \( \land \text{tid dir partner page} \cdot \text{ipc-precondition} \land \text{dir partner page} \land \text{atomic-step} \ (s a) \)
\( \land \text{ev-signal-precondition} \land \text{tid partner} \ s = \text{ev-signal-precondition} \land \text{tid partner} \ (\text{atomic-step} \ s a) \)
proof –
fix \( \text{tid dir partner page} \ s \)
let \( ?s = \text{atomic-step} \ s a \)
have \( \forall \ p q \cdot \text{sp-impl-subj-subj} \ s p q = \text{sp-impl-subj-subj} \ ?s p q \)
\( \land (\forall \ p x m \cdot \text{sp-impl-subj-obj} \ s p x m = \text{sp-impl-subj-obj} \ ?s p x m) \)
unfolding atomic-step-def atomic-step-ipc-def
atomic-step-ev-wait-all-def atomic-step-ev-wait-one-def
atomic-step-ev-signal-def set-object-value-def
by (auto split add: int-point-t.splits ipc-stage-t.splits ipc-direction-t.splits ev-wait-stage-t.splits ev-consume-t.splits ev-signal-stage-t.splits)
thus ipc-precondition tid dir partner page \( s = \text{ipc-precondition tid dir partner page} \ (\text{atomic-step} \ s a) \)
\( \land \text{ev-signal-precondition tid partner} \ s = \text{ev-signal-precondition tid partner} \ (\text{atomic-step} \ s a) \)
unfolding ipc-precondition-def ev-signal-precondition-def by simp
qed
moreover have \( \land \ b \cdot (\uparrow (\text{atomic-step} (\downarrow s) b))) = \text{atomic-step} (\downarrow s) b \)
using rstate-invariant atomic-step-preserves-invariants rstate-up-down by auto
ultimately show \(?\)thesis
unfolding aborting-def rstep-def ev-signal-precondition-def
by (simp split add: int-point-t.splits ipc-stage-t.splits ev-wait-stage-t.splits ev-signal-stage-t.splits)
qed

lemma ipc-precondition-of-partner-consistent:
assumes vpeq \( \forall \ d \in \text{rkinvolved} (SK-IPC \ \text{dir WAIT} \ \text{partner page}) \cdot \text{rvpeq} \ d \ s t \)
shows ipc-precondition partner dir' page' (\downarrow s) = ipc-precondition partner dir' u page' (\downarrow t)
proof–
from assms ipc-precondition-weakly-step-consistent rstate-invariant
show \(?\)thesis
unfolding rvpeq-def rkinvolved-def
by auto
qed
lemma ev-signal-precondition-of-partner-consistent:
assumes vpeq \( \forall d \in rkinvolved \) (SK-EV-SIGNAL EV-SIGNAL-FINISH partner) . rvpeq d s t
shows ev-signal-precondition partner u (↓ s) = ev-signal-precondition partner u (↓ t)
proof –
from assms ev-signal-precondition-weakly-step-consistent rstate-invariant
show ?thesis
unfolding rvpeq-def rkinvolved-def
by auto
qed

lemma waiting-consistent:
shows \( \forall s t u a \). rvpeq (rcurrent s) s t \& (\forall d \in rkinvolved a . rvpeq d s t) \rightarrow rwaiting s u a = rwaiting t u a
proof –
{ fix s t u a
assume vpeq: rvpeq (rcurrent s) s t
assume vpeq-involved: \( \forall d \in rkinvolved a . rvpeq d s t \)
assume vpeq-u: rvpeq u s t
have rwaiting s u a = rwaiting t u a proof (cases a)
case SK-IPC
  thus rwaiting s u a = rwaiting t u a
  using ipc-precondition-of-partner-consistent vpeq-involved
unfolding waiting-def by (auto split add: ipc-stage-t.splits)
case SK-EV-WAIT
  thus rwaiting s u a = rwaiting t u a
  using ev-signal-precondition-of-partner-consistent vpeq-involved vpeq vpeq-u
unfolding waiting-def rkinvolved-def ev-signal-precondition-def
  rvpeq-def vpeq-def vpeq-local-def
  by (auto split add: ipc-stage-t.splits ev-wait-stage-t.splits ev-consume-t.splits)
  qed (simp add: waiting-def, simp add: waiting-def)
} thus ?thesis by auto
qed

lemma ipc-precondition-ensures-ifp:
assumes ipc-precondition (current s) dir partner page s and atomic-step-invariant s
shows rifp partner (current s)
proof –
let ?sp = \( \lambda t1 t2 \). Policy.sp-spec-subj-subj (partition t1) (partition t2)
have ?sp (current s) partner \( \lor ?sp partner (current s) \)
using assms unfolding ipc-precondition-def atomic-step-invariant-def sp-subset-def
by (cases dir, auto)
thus ?thesis
unfolding rifp-def using Policy-properties.ifp-compatible-with-sp-spec by auto
qed

lemma ev-signal-precondition-ensures-ifp:
assumes ev-signal-precondition (current s) partner s and atomic-step-invariant s
shows rifp partner (current s)
proof –
let ?sp = \( \lambda t1 t2 \). Policy.sp-spec-subj-subj (partition t1) (partition t2)
have ?sp (current s) partner \( \lor ?sp partner (current s) \)
using assms unfolding ev-signal-precondition-def atomic-step-invariant-def sp-subset-def
by (auto)
thus ?thesis
unfolding rifp-def using Policy-properties.ifp-compatible-with-sp-spec by auto
qed

lemma involved-ifp:
shows ∀ s a . ∀ d ∈ rkinvolved a . rprecondition s (rcurrent s) a → rifp d (rcurrent s)
proof–
{ fix s a d
 assume d-involved: d ∈ rkinvolved a
 assume prec: rprecondition s (rcurrent s) a
 from d-involved prec have rifp d (rcurrent s)
 using ipc-precondition-ensures-ifp ev-signal-precondition-ensures-ifp rstate-invariant
 unfolding rkinvolved-def rprecondition-def atomic-step-precondition-def rcurrent-def Kernel.involved-def
 by (cases a,simp,auto split add: int-point-t.splits ipc-stage-t.splits ev-signal-stage-t.splits)
 }
thus ?thesis by auto
qed

lemma spec-of-waiting-ev:
shows ∀ s a. rwaiting s (rcurrent s) (SK-EV-WAIT EV-FINISH EV-CONSUME-ALL) → rstep s a = s
unfolding waiting-def
by auto

lemma spec-of-waiting-ev-w:
shows ∀ s a. rwaiting s (rcurrent s) (SK-EV-WAIT EV-WAIT EV-CONSUME-ALL) → rstep s (SK-EV-WAIT EV-WAIT EV-CONSUME-ALL) = s
unfolding rstep-def atomic-step-def
by (auto split add: int-point-t.splits ipc-stage-t.splits ev-wait-stage-t.splits)

lemma spec-of-waiting:
shows ∀ s a. rwaiting s (rcurrent s) a → rstep s a = s
unfolding waiting-def rstep-def atomic-step-def atomic-step-ipc-def
atomic-step-ev-signal-def atomic-step-ev-wait-all-def
atomic-step-ev-wait-one-def
by (auto split add: int-point-t.splits ipc-stage-t.splits ev-wait-stage-t.splits)
end

3.9 Link implementation to CISK: the specific separation kernel is an interpretation of the generic model.

theory link-separation-kernel-model-to-CISK
imports separation-kernel-model
begin

We show that the separation kernel instantiation satisfies the specification of CISK.

theorem CISK-proof-obligations-satisfied:
shows Controllable-Interruptible-Separation-Kernel
rstep
routput-f
(↑sd)
proof (unfold-locales)
— show that rvpeq is equivalence relation
show ∀ a b c u. (rvpeq u a b ∧ rvpeq u b c) → rvpeq u a c
and ∀ a b u. rvpeq u a b → rvpeq u b a
and ∀ a u. rvpeq u a a
using inst-vpeq-rel by metis
— show output consistency
show ∀ a s t. rvpeq (rcurrent s) s t ∧ rcurrent s = rcurrent t → routput-f s a = routput-f t a
using inst-output-consistency by metis
— show reflexivity of ifp
show ∀ u. rifp u u
using inst-ifp-refl by metis
— show step consistency
show ∀ s t u a. rvpeq u s t ∧ rvpeq (rcurrent s) s t ∧ rcurrent s = rcurrent t → rvpeq u (rstep s a) (rstep t a)
using inst-weakly-step-consistent by blast
— show step atomicity
show ∀ a s t u. rifp (rcurrent s) u ∧ True ∧ rprecondition s (rcurrent s) a ∧ True ∧ rprecondition t (rcurrent t) a ∧ rcurrent s = rcurrent t → rvpeq u (rstep s a) (rstep t a)
using inst-step-atomicity by metis
— show cswitch is independent of state
show ∀ n s t. rcurrent s = rcurrent t → rcurrent (rcswitch n s) = rcurrent (rcswitch n t)
using inst-cswitch-independent-of-state by metis
— show cswitch consistency
show ∀ u s t n. rvpeq u s t → rvpeq u (rcswitch n s) (rcswitch n t)
using inst-cswitch-consistency by metis
— Show the empt action sequence is in AS-set
show [] ∈ rAS-set
unfolding rAS-set-def
by auto
— The invariant for the initial state, already encoded in rstate-t
show True
by auto
— Step function of the invariant, already encoded in rstate-t
show ∀ s n. True → True
by auto
— The precondition does not change with a context switch
show ∀ s d n a. rprecondition s d a → rprecondition (rcswitch n s) d a
using precondition-after-cswitch by blast
— The precondition holds for the first action of each action sequence
show ∀ s d aseq. True ∧ aseq ∈ rAS-set ∧ aseq ≠ [] → rprecondition s d (hd aseq)
using prec-first-IPC-action prec-first-EV-WAIT-action prec-first-EV-SIGNAL-action
unfolding rAS-set-def is-sub-seq-def
by auto
show \forall s a a'. (\exists a s e \text{ rAS-set. is-sub-seq a a' aseq}) \land \text{ True } \land \text{ rprecondition s (rcurrent s) a } \land \lnot \text{ raborting s (rcurrent s) a } \longrightarrow
\text{ rprecondition (rstep s a') (rcurrent s) a'}

using \text{ prec-after-IPC-step \ prec-after-EV-SIGNAL-step \ prec-after-EV-WAIT-step}

unfolding \text{ rAS-set-def is-sub-seq-def}

by auto

— Steps of other domains do not influence the precondition

show \forall s t a a'. rcurrent s \not\in d \land \text{ rprecondition s d a } \longrightarrow \text{ rprecondition (rstep s a') d a}

using \text{ pre-dom-independent by blast}

— The invariant

show \forall s a. \text{ True } \longrightarrow \text{ True}

by auto

— Aborting does not depend on a context switch

show \forall n s. \text{ raborting (rcswitch n s) = raborting s}

using \text{ aborting-switch-independent by auto}

— Aborting does not depend on actions of other domains

show \forall s t d a a'. rcurrent s \not\in d \land \text{ rprecondition s d a } \longrightarrow \text{ raborting (rstep s a') d = raborting s d}

using \text{ aborting-dom-independent by auto}

— Aborting is consistent

show \forall s t u. \text{ rvpeq u s t } \longrightarrow \text{ raborting s u = raborting t u}

using \text{ raborting-consistent by auto}

— Waiting does not depend on a context switch

show \forall n s. \text{ rwating (rcswitch n s) = rwaiting s}

using \text{ waiting-switch-independent by auto}

— Waiting is consistent

show \forall s t u a. \text{ rvpeq (rcurrent s) s t } \land (\forall d \in \text{ rkinvolved a } \text{. rvpeq d s t})

\land \text{ rvpeq u s t}

\longrightarrow \text{ rwaiting s u a = rwaiting t u a}

unfolding \text{ Kernel.involved-def}

using \text{ waiting-consistent by auto}

— Domains that are involved in an action may influence the domain of the action

show \forall s a. \forall d \in \text{ rkinvolved a } \text{. rprecondition s (rcurrent s) a } \longrightarrow \text{ rifp d (rcurrent s)}

using \text{ involved-ifp by blast}

— An action that is waiting does not change the state

show \forall s a. \text{ rwaiting s (rcurrent s) a } \longrightarrow \text{ rstep s a = s}

using \text{ spec-of-waiting by blast}

— Proof obligations for set-error-code. Right now, they are all trivial

show \forall s t d a a'. \text{ rcurrent s } \not\in d \land \text{ raborting s d a } \longrightarrow \text{ raborting (rset-error-code s a') d a}

unfolding \text{ rset-error-code-def}

by auto

show \forall s t u a. \text{ rvpeq u s t } \longrightarrow \text{ rvpeq u (rset-error-code s a) (rset-error-code t a)}

unfolding \text{ rset-error-code-def}

by auto

show \forall s u a. \text{ rifp (rcurrent s) u } \longrightarrow \text{ rvpeq u s (rset-error-code s a)}

unfolding \text{ rset-error-code-def}

by (metis (\forall a u. \text{ rvpeq u a a}))

show \forall s a. \text{ rcurrent (rset-error-code s a) = rcurrent s}

unfolding \text{ rset-error-code-def}

by auto

show \forall s t d a a'. \text{ rprecondition s d a } \land \text{ raborting s (rcurrent s) a' } \longrightarrow \text{ rprecondition (rset-error-code s a') d a}

unfolding \text{ rset-error-code-def}

by auto

show \forall s t d a a'. \text{ rcurrent s } \not\in d \land \text{ rwaiting s d a } \longrightarrow \text{ rwaiting (rset-error-code s a') d a}

unfolding \text{ rset-error-code-def}

by auto
Now we can instantiate CISK with some initial state, interrupt function, etc.

**interpretation Inst:**

- `Controllable-Interruptible-Separation-Kernel`
- `rstep` — step function, without program stack
- `routput-f` — output function
- `s0` — initial state
- `rcurrent` — returns the currently active domain
- `rcswitch` — switches the currently active domain
- `(op =) 42` — interrupt function (yet unspecified)
- `rinvolved` — returns a set of threads involved in the given action
- `rifp` — information flow policy
- `rvpeq` — view partitioning
- `rsAS-set` — the set of valid action sequences
- `rinvariant` — the state invariant
- `rprecondition` — the precondition for doing an action
- `raborting` — condition under which an action is aborted
- `rwating` — condition under which an action is delayed
- `rset-error-code` — updates the state. Has no meaning in the current model.

**using CISK-proof-obligations-satisfied by auto**

The main theorem: the instantiation implements the information flow policy `ifp`.

**theorem risecure:**

**using Inst.unwinding-implies-secure-CISK**

**by blast**

**end**

## 4 Conclusion

We have introduced a theory of intransitive non-interference for separation kernels with control. We have shown that it can be instantiated for a simple API consisting of IPC and events.

**References**

