Formal specification of a generic separation kernel

Citation for published version (APA):

Document status and date:
Published: 01/01/2014

Publisher Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:
• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.

Download date: 27. Jul. 2023
D31.1
Formal Specification of a Generic 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>
<tr>
<td>Deliverable type:</td>
<td>R</td>
</tr>
<tr>
<td>Deliverable reference number:</td>
<td>ICT-318353 / D31.1 / 0.0</td>
</tr>
<tr>
<td>Activity and Work package contributing to deliverable:</td>
<td>Activity 3 / WP 3.1</td>
</tr>
<tr>
<td>Due date:</td>
<td>September 2013 – M12</td>
</tr>
<tr>
<td>Actual submission date:</td>
<td>28th August, 2014</td>
</tr>
<tr>
<td>Responsible organisation:</td>
<td>Open University of The Netherlands</td>
</tr>
<tr>
<td>Editors:</td>
<td>Freek Verbeek, Julien Schmaltz</td>
</tr>
<tr>
<td>Dissemination level:</td>
<td>PU</td>
</tr>
<tr>
<td>Revision:</td>
<td>0.0 (r-2)</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
Freek Verbeek, Julien Schmaltz (Open University of The Netherlands)

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

Intransitive noninterference has been a widely studied topic in the last few decades. Several well-established methodologies apply interactive theorem proving to formulate a noninterference theorem over abstract academic models. In joint work with several industrial and academic partners throughout Europe, we are helping in the certification process of PikeOS, an industrial separation kernel developed at SYSGO. In this process, established theories could not be applied. We present a new generic model of separation kernels and a new theory of intransitive noninterference. The model is rich in detail, making it suitable for formal verification of realistic and industrial systems such as PikeOS. Using a refinement-based theorem proving approach, we ensure that proofs remain manageable.

This document corresponds to the deliverable D31.1 of the EURO-MILS Project http://www.euromils.eu.
Contents

1 Introduction 2

2 Preliminaries 3
   2.1 Binders for the option type ................................................. 3
   2.2 Theorems on lists ............................................................ 4

3 A generic model for separation kernels 6
   3.1 K (Kernel) ........................................................................... 7
      3.1.1 Execution semantics ......................................................... 8
   3.2 SK (Separation Kernel) ............................................................. 9
      3.2.1 Security for non-interfering domains .................................... 10
      3.2.2 Security for indirectly interfering domains ............................. 21
   3.3 ISK (Interruptible Separation Kernel) ....................................... 35
   3.4 CISK (Controlled Interruptible Separation Kernel) ...................... 48
      3.4.1 Execution semantics ......................................................... 50
      3.4.2 Formulations of security .................................................... 51
      3.4.3 Proofs ........................................................................... 51

4 Instantiation by a separation kernel with concrete actions 57
   4.1 Model of a separation kernel configuration ................................. 58
      4.1.1 Type definitions .............................................................. 58
      4.1.2 Configuration ................................................................. 58
   4.2 Formulation of a subject-subject communication policy and an information flow policy, and showing both can be derived from subject-object configuration data .......... 59
      4.2.1 Specification ................................................................. 59
      4.2.2 Derivation ................................................................. 59
   4.3 Separation kernel state and atomic step function ........................ 60
      4.3.1 Interrupt points .............................................................. 60
      4.3.2 System state ................................................................. 61
      4.3.3 Atomic step ................................................................. 61
   4.4 Preconditions and invariants for the atomic step .......................... 63
      4.4.1 Atomic steps of SK_IPC preserve invariants ....................... 64
      4.4.2 Summary theorems on atomic step invariants ..................... 65
   4.5 The view-partitioning equivalence relation .................................. 66
      4.5.1 Elementary properties ..................................................... 67
   4.6 Atomic step locally respects the information flow policy ................ 68
      4.6.1 Locally respects of atomic step functions ............................. 68
      4.6.2 Summary theorems on view-partitioning locally respects ......... 70
   4.7 Weak step consistency ............................................................. 71
      4.7.1 Weak step consistency of auxiliary functions ....................... 71
      4.7.2 Weak step consistency of atomic step functions ................... 73
      4.7.3 Summary theorems on view-partitioning weak step consistency ... 75
   4.8 Separation kernel model ........................................................... 75
      4.8.1 Initial state of separation kernel model ............................... 76
      4.8.2 Types for instantiation of the generic model ....................... 76
      4.8.3 Possible action sequences ............................................... 77
      4.8.4 Control ................................................................. 78
      4.8.5 Discharging the proof obligations .............................. 78
4.9 Link implementation to CISK: the specific separation kernel is an interpretation of the generic model. .......................................................... 88

5 Related Work .......................................................... 90

6 Conclusion .......................................................... 92
   6.0.1 Acknowledgement. ............................................. 92
1 Introduction

Separation kernels are at the heart of many modern security-critical systems [23]. With next generation technology in cars, aircrafts and medical devices becoming more and more interconnected, a platform that offers secure decomposition of embedded systems becomes crucial for safe and secure performance. PikeOS, a separation kernel developed at SYSGO, is an operating system providing such an environment [12, 2]. A consortium of several European partners from industry and academia works on the certification of PikeOS up to at least Common Criteria EAL5+, with ”+” being applying formal methods compliant to EAL7. Our aim is to derive a precise model of PikeOS and a precise formulation of the PikeOS security policy.

A crucial security property of separation kernels is intransitive noninterference. This property is typically required for systems with multiple independent levels of security (MILS) such as PikeOS. It ensures that a given security policy over different subjects of the system is obeyed. Such a security policy dictates which subjects may flow information to which other subjects.

Intransitive noninterference has been an active research field for the last three decades. Several papers have been published on defining intransitive noninterference and on unwinding methodologies that enable the proof of intransitive noninterference from local proof obligations. However, in the certification process of PikeOS these existing methodologies could not be directly applied. Generally, the methodologies are based on highly abstract generic models of computation. The gap between such an abstract model and the reality of PikeOS is large, making application of the methodologies tedious and cumbersome.

This paper presents a new generic model for separation kernels called CISK (for: Controlled Interruptible Separation Kernel). This model is richer in details and contains several facets present in many separation kernels, such as interrupts, context switches between domains and a notion of control. Regarding the latter, this concerns the fact that the kernel exercises control over the executions as performed by the domains. The kernel can, e.g., decide to skip actions of the domains, or abort them halfway. We prove that any instantiation of the model provides intransitive noninterference. The model and proofs have been formalized in Isabelle/HOL [21] which are included in the subsequent sections of this document.

We have adopted Rushby’s definition of intransitive noninterference [24]. We first present an overview of our approach and then discuss the relation between our approach and existing methodologies in the next section.

Overview

Generally, there are two conflicting interests when using a generic model. On the one hand the model must be sufficiently abstract to ensure that theorems and proofs remain manageable. On the other hand, the model must be rich enough and must contain sufficient domain-knowledge to allow easy instantiation. Rushby’s model, for example, is on one end of the spectrum: it is basically a Mealy machine, which is a highly abstract notion of computation, consisting only of state, inputs and outputs [24]. The model and its proofs are manageable, but making a realistic instantiation is tedious and requires complicated proofs.

We aim at the other side of the spectrum by having a generic model that is rich in detail. As a result, instantiating the model with, e.g., a model of PikeOS can be done easily. To ensure maintainability of the theorems and proofs, we have applied a highly modularized theorem proving technique.

Figure 1 shows an overview. The initial module “Kernel” is close to a Mealy machine, but has several facets added, including interrupts, context switches and control. New modules are added in such a way that each new module basically inserts an adjective before “Kernel”. The use of modules allows us to prove, e.g., a separation theorem in module “Separation Kernel” and subsequently to reuse this theorem later on when details on control or interrupts are added.

The second module adds a notion of separation, yielding a module of a Separation Kernel (SK). A security policy is added that dictates which domains may flow information to each other. Local proof
obligations are added from which a global theorem of noninterference is proven. This global theorem is the *unwinding* of the local proof obligations.

In the third module calls to the kernel are no longer considered atomic, yielding an Interruptible Separation Kernel (ISK). In this model, one call to the kernel is represented by an *action sequence*. Consider, for example, an IPC call (for: Inter Process Communication). From the point of view of the programmer this is one kernel call. From the point of view of the kernel it is an action sequence consisting of three stages IPC_PREP, IPC_WAIT, and IPC_SEND. During the PREP stage, it is checked whether the IPC is allowed by the security policy. The WAIT stage is entered if a thread needs to wait for its communication partner. The SEND stage is data transmission. After each stage, an interrupt may occur that switches the current context. A consequence of allowing interruptible action sequences is that it is no longer the case that any execution, i.e., any combination of atomic kernel actions, is realistic. We formulate a definition of *realistic execution* and weaken the proof obligations of the model to apply only to realistic executions.

The final module provides an interpretation of control that allows atomic kernel actions to be aborted or delayed. Additional proof obligations are required to ensure that noninterference is still provided. This yields a Controlled Interruptible Separation Kernel (CISK). When sequences of kernel actions are aborted, error codes can be transmitted to other domains. Revisiting our IPC example, after the PREP stage the kernel can decide to abort the action. The IPC action sequence will not be continued and error codes may be sent out. At the WAIT stage, the kernel can delay the action sequence until the communication partner of the IPC call is ready to receive.

In Section 3 we introduce a theory of intransitive non-interference for separation kernels with control, based on [31]. We show that it can be instantiated for a simple API consisting of IPC and events (Section 4). The rest of *this* section gives some auxiliary theories used for Section 3.

## 2 Preliminaries

### 2.1 Binders for the option type

```plaintext
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 “\(m \rightarrow \alpha\)” 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: “\(m_1||m_2 \rightarrow \alpha\)” represents “First compute \(m_1\) and \(m_2\), if one of them is None then return True, otherwise pass the result to \(\alpha\)”.

**Definition** B2 :: `a option ⇒ (\(a \Rightarrow \text{bool}\)) ⇒ \text{bool} (infixl → 65)

where \(B m \alpha \equiv \text{case } m \text{ of None } ⇒ \text{True} \mid \text{Some } a \Rightarrow \alpha\)

**Syntax** B2 :: `[`a option`, `a option`, (`a ⇒ `a ⇒ `bool`)] ⇒ `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 ⇒ True | (Some s1) ⇒ (case t of None ⇒ True | (Some t1) ⇒ f s1 t1))

using assms unfolding B2-def B-def by(cases s,cases t,simp+)

**Lemma** rewrite-B-None[simp]:

shows None ⇒ α ⇒ True

unfolding B-def by(auto)

**Lemma** rewrite-B-m-True[simp]:

shows m ⇒ (λ a . True) = True

unfolding B-def by(cases m,simp+)

**Lemma** rewrite-B2-cases:

shows (case a of None ⇒ True | (Some s) ⇒ (case b of None ⇒ True | (Some t) ⇒ f s t))

= (∀ s t . a = (Some s) ∧ b = (Some t) → f s t)

by(cases a,simp,cases b,simp+)

**Definition** strict-equal :: `a option ⇒ `a ⇒ `bool

where strict-equal m a ≡ case m of None ⇒ False | (Some a') ⇒ a' = a

end

### 2.2 Theorems on lists

theory List-Theorems

imports List

begin

**Definition** lastn :: `nat ⇒ `a list ⇒ `a list

where lastn n x = drop ((length x) − n) x

**Definition** is-sub-seq :: `a ⇒ `a ⇒ `a list ⇒ `bool

where is-sub-seq a b x ≡ \(\exists\ n . \text{Suc } n < \text{length } x \land x!n = a \land x!(\text{Suc } n) = b\)

**Definition** prefixes :: `a list set ⇒ `a list set

where prefixes s ≡ \{ x . \(\exists\ n \ y . n > 0 \land \ y \in x \land \text{take } n \ y = x\}\)

**Lemma** drop-one[simp]:

shows drop (Suc 0) x = tl x by(induct x,auto)

**Lemma** length-ge-one:

shows x \(\neq\) [] ⇒ length x ≥ 1 by(induct x,auto)

**Lemma** take-but-one[simp]:

shows x \(\neq\) [] ⇒ lastn ((length x) − 1) 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

---

EURO-MILS D31.1
lemma \text{lastn-one-less}:
shows \( n > 0 \land n \leq \text{length}\ x \land \text{lastn} n\ x = (a \# y) \longrightarrow \text{lastn} (n - 1)\ x = y \) unfolding \text{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 \text{list-sub-implies-member}:
shows \( \forall a\ x.\ \text{set}(a \# x) \subseteq Z \longrightarrow a \in Z \) by auto

lemma \text{subset-smaller-list}:
shows \( \forall a\ x.\ \text{set}(a \# x) \subseteq Z \longrightarrow \text{set}\ x \subseteq Z \) by auto

lemma \text{second-elt-is-hd-tl}:
shows \( \text{tl}\ x = (a \# x') \longrightarrow a = x!1 \) by (cases x,auto)

lemma \text{length-ge-2-implies-tl-not-empty}:
shows \( \text{length}\ x \geq 2 \longrightarrow \text{tl}\ x \neq [] \) by (cases x,auto)

lemma \text{length-lt-2-implies-tl-empty}:
shows \( \text{length}\ x < 2 \longrightarrow \text{tl}\ x = [] \) by (cases x,auto)

lemma \text{first-second-is-sub-seq}:
shows \( \text{length}\ x \geq 2 \Longleftrightarrow \text{is-sub-seq}(\hd\ x) (x!1)\ x \)
proof
assume \( \text{length}\ x \geq 2 \)

hence \( 1: (\text{Suc}\ 0) < \text{length}\ x \) by auto

hence \( x!0 = \hd\ x \) by(cases x,auto)

from this \( 1\) show \( \text{is-sub-seq}(\hd\ x) (x!1)\ x \) unfolding \text{is-sub-seq-def} by auto
qed

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

lemma \text{def-of-hd}:
shows \( y = a \# x \longrightarrow \hd\ y = a \) by simp

lemma \text{def-of-tl}:
shows \( y = a \# x \longrightarrow \text{tl}\ y = x \) by simp

lemma \text{drop-yields-results-implies-nbound}:
shows \( \text{drop}\ n\ x \neq [] \longrightarrow n < \text{length}\ x \)
by(induct x,auto)

lemma \text{hd-take[simp]}:
shows \( n > 0 \longrightarrow \hd(\text{take}\ n\ x) = \hd\ x \)
by(cases x,simp,cases n,auto)

lemma \text{consecutive-is-sub-seq}:
shows \( a \# (b \# x) = \text{lastn}\ n\ y \longrightarrow \text{is-sub-seq}\ a\ b\ y \)
proof
  assume 1: a ≠ (b ≠ x) = lastn n y
  from 1 drop-Suc[where n=(length y) − n and x=y]
  drop-tl[where n=(length y) − n and x=y]
  def-of-tl[where y=lastn n y and a=a and x=b≠x]
  drop-yields-results-implies-nbound[where n=Suc (length y − n) and x=y]
  have 3: Suc (length y − n) < length y unfolding lastn-def by auto
  from 3 1 hd-drop-is-nth[where n=(length y) − n and x=y] def-of-hd[where y=drop (Suc (length y − n)) y and x=x and a=b]
  drop-Suc[where n=(length y) − n and x=x]
  drop-tl[where n=(length y) − n and x=x]
  def-of-tl[where y=lastn n y and a=a and x=b≠x]
  have 4: y!=(length y − n) = a unfolding lastn-def by auto
  from 3 1 hd-drop-is-nth[where n=Suc ((length y) − n) and x=x] def-of-hd[where y=drop (Suc (length y − n))]
  and x=x and a=b]
  drop-Suc[where n=(length y) − n and x=x]
  drop-tl[where n=(length y) − n and x=x]
  def-of-tl[where y=lastn n y and a=a and x=b≠x]
  have 5: y!=(length y − n) = b unfolding lastn-def by auto
  from 3 4 5 show ?thesis
  unfolding is-sub-seq-def by auto
qed

lemma sub-seq-in-prefixes:
  assumes 3 y ∈ prefixes X. is-sub-seq a a’ y
  shows 3 y ∈ X. is-sub-seq a a’ y
proof
  from assms obtain y where y: y ∈ prefixes X ∧ is-sub-seq a a’ y by auto
  then obtain n x where x: n > 0 ∧ x ∈ X ∧ take n x = y
  unfolding prefixes-def by auto
  from y obtain i where sub-seq-index: Suc i < length y ∧ i = a ∧ y! Suc i = a’
  unfolding is-sub-seq-def by auto
  from sub-seq-index x have is-sub-seq a a’ x
  unfolding is-sub-seq-def using nth-take by auto
  from this x show ?thesis by metis
qed

lemma set-tl-is-subset:
  shows set (tl x) ⊆ set x by (induct x,auto)
lemma x-is-hd-snd-tl:
  shows length x ≥ 2 → x = (hd x) # x!1 # tl(tl x)
proof(induct x)
case Nil
  show ?case by auto
case (Cons a x)
  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-ll-x:
  shows x ≠ [] ∧ hd x ≠ [] → tl (hd x) ≠ tl x ≠ x using tl-x-not-x by (induct x, simp,auto)
end

3 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 [25] 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” [31].

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 [25]. 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.

3.1 K (Kernel)

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

The model makes use of the following types:

'state_t 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_t 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.

'action_t execution 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 take 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

EURO-MILS D31.1 Page 7 of 94
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 $\alpha$ 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 $\alpha$. It may also return None, indicating that no action is done. Secondly, it returns the updated execution. When executing action $a$, 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.

$$\text{locale Kernel} =$$

$$\begin{align*}
\text{fixes } kstep &:: 'state-t \Rightarrow 'action-t \Rightarrow 'state-t \\
\text{and } output-f &:: 'state-t \Rightarrow 'action-t \Rightarrow 'output-t \\
\text{and } s0 &:: 'state-t \\
\text{and } current &:: 'state-t \Rightarrow 'dom-t \\
\text{and } cswitch &:: 'time-t \Rightarrow 'state-t \Rightarrow 'state-t \\
\text{and } interrupt &:: 'time-t \Rightarrow \text{bool} \\
\text{and } kprecondition &:: 'state-t \Rightarrow 'action-t \Rightarrow \text{bool} \\
\text{and } realistic\_execution &:: 'action-t \Rightarrow \text{action-t execution} \Rightarrow \text{bool} \\
\text{and } control &:: 'state-t \Rightarrow 'dom-t \Rightarrow 'action-t \Rightarrow 'state-t \\
\text{and } involved &:: 'action-t \Rightarrow 'dom-t \Rightarrow 'state-t \\
\end{align*}$$

begin

3.1.1 Execution semantics

Short hand notations for using function control.

$$\begin{align*}
\text{definition } next-action &:: 'state-t \Rightarrow ('dom-t \Rightarrow 'action-t \Rightarrow 'state-t) \\
\text{where } next-action s execs &:: \text{fst (control s (current s) (execs (current s)))} \\
\text{definition } next-execs &:: 'state-t \Rightarrow ('dom-t \Rightarrow 'action-t \Rightarrow 'state-t) \\
\text{where } next-execs s execs &:: \text{fun-upd execs (current s) (fst (snd (control s (current s) (execs (current s))))}) \\
\text{definition } next-state &:: 'state-t \Rightarrow ('dom-t \Rightarrow 'action-t \Rightarrow 'state-t) \\
\text{where } next-state s execs &:: \text{snd (snd (control s (current s) (execs (current s))))} \\
\end{align*}$$

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.

$$\begin{align*}
\text{abbreviation } thread-empty &:: 'action-t \Rightarrow \text{bool} \\
\text{where } thread-empty\_exec &:: \text{exec} = [] \lor \text{exec} = [[]] \\
\end{align*}$$

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

$$\begin{align*}
\text{definition } step &:: \text{step s oa \equiv case oa of None } \Rightarrow s | (\text{Some } a) \Rightarrow kstep s a \\
\text{definition } kprecondition &:: 'state-t \Rightarrow 'action-t \Rightarrow \text{bool} \\
\text{where } kprecondition s a &:: a \Rightarrow kprecondition s \\
\text{definition } involved &:: \text{involved oa \equiv case oa of None } \Rightarrow \{\} | (\text{Some } a) \Rightarrow \text{involved a} \\
\end{align*}$$

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 \((\text{step } s')\). 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.

\[
\text{function } \text{run} \colon \text{time-t} \Rightarrow (\text{state-t} \Rightarrow (\text{action-t} \Rightarrow (\text{execution} \Rightarrow (\text{state-t} \Rightarrow \text{option})))
\]

\[
\text{where } \text{run } 0 \text{ s execs } = \text{ s/}\text{divides.alt0}
\]

\[
\text{run } (\text{Suc n}) \text{ None execs } = \text{ None/}\text{divides.alt0}
\]

\[
\text{interrupt } (\text{Suc n}) \Rightarrow \text{run } (\text{Suc n}) (\text{Some s}) \text{ execs } = \text{ run n (Some (cswitch (Suc n) s)) execs}
\]

\[
\text{~interrupt } (\text{Suc n}) \Rightarrow \text{thread-empty}(\text{execs (current s)}) \Rightarrow \text{run } (\text{Suc n}) (\text{Some s}) \text{ execs } = \text{ run n (Some s execs)}
\]

\[
\text{~interrupt } (\text{Suc n}) \Rightarrow \text{~thread-empty}(\text{execs (current s)}) \Rightarrow \text{~precondition (next-state s execs) (next-action s execs)} \Rightarrow \text{run } (\text{Suc n}) (\text{Some s}) \text{ execs } = \text{ None}
\]

\[
\text{~interrupt } (\text{Suc n}) \Rightarrow \text{~thread-empty}(\text{execs (current s)}) \Rightarrow \text{precondition (next-state s execs) (next-action s execs)} \Rightarrow \text{run } (\text{Suc n}) (\text{Some s}) \text{ execs } = \text{ run n (Some (step (next-state s execs) (next-action s execs))) (next-execs s execs)}
\]

\[
\text{using } \text{not0-implies-Suc by (metis option.exhaust prod-cases3,auto)}
\]

\text{termination by lexicographic-order}

end

end

3.2 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 \( i_a \). Function \( \text{vpeq} \) 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:

\textbf{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.

\textbf{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 (\text{cswitch} consistency).

Also, \text{cswitch} can only change which domain is currently active (\text{cswitch} consistency).

\textbf{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 (\text{next-action} consistent, \text{next-execs} consistent), the state as updated by the control function remains in \text{vpeq} (\text{next-state} consistent, \text{local-respects} \text{next-state}). Finally, function control cannot change which domain is active (\text{current} \text{next-state}).

\textbf{definition} \( \text{actions-in-execution: } \text{action-t} \text{ execution } \Rightarrow \text{action-t set} \)

\textbf{where} \( \text{actions-in-execution exec } = \{ a . \exists \text{ aseq } \in \text{ set exec} . a \in \text{ set aseq} \} \)

\textbf{locale} \( \text{Separation-Kernel } = \text{ Kernel kstep output-f s0 current cswitch interrupt kprecondition realistic-execution control kinvolved} \)

\textbf{for} \text{kstep } \Rightarrow \text{state-t } \Rightarrow \text{action-t } \Rightarrow \text{state-t}

\textbf{and} \text{output-f } \Rightarrow \text{state-t } \Rightarrow \text{action-t } \Rightarrow \text{output-t}
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 \( = 'dom-t \Rightarrow 'dom-t \Rightarrow bool \)

**where** unrelated \( d u \equiv \neg ifp'' '' d u \)**

### 3.2.1 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 \( u \).
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 :

\[
\begin{align*}
\text{\{'dom-t \Rightarrow \text{'action-t execution}\} \Rightarrow \text{'dom-t} \Rightarrow \text{\{'dom-t \Rightarrow \text{'action-t execution}\}} \\
\text{where} \quad \text{purge execs } u \equiv \lambda d . \text{. (if unrelated } d \text{ u then} \\
\quad \text{(SOME alpha . realistic-execution alpha)} \\
\quad \text{else execs } d)
\end{align*}
\]

A normal run from initial state s0 ending in state s\_f is equivalent to a run purged for domain (current\_s\_f).

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

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

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 \text{"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 \| t \rightarrow (\lambda s t . (\forall v . 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

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

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 . ifp^∗∗ d u \rightarrow 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

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

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-nt:

**assumes** prec-t: precondition (next-state t execs2) (next-action t execs2)

**assumes** not-ifp-curr-u: \neg ifp^∗∗ (current t) u

**assumes** vpeq-s-t: \forall v . ifp^∗∗ v u \rightarrow vpeq v s t

**shows** \forall v . ifp^∗∗ v u \rightarrow vpeq v s (step (next-state t execs2) (next-action t execs2))

**proof**

\{ fix v \}
assume ifp-v-uc ifp^*** v u

from ifp-v-uc not-ifp-curr-u have unrelated: ~ifp^*** (current t) v using rtranclp-trans by metis
from this current-next-state[THEN spec,THEN spec,where x1=t] locally-respects[THEN spec,THEN spec,THEN spec,where x1=next-state t execs2] vpeq-reflexive prec-s 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 unrelated this locally-respects-next-state vpeq-transitive have vpeq v t (step (next-state t execs2) (next-action t execs2)) by blast
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
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-ns-t:
assumes prec-s: precondition (next-state s execs) (next-action s execs)
assumes not-ifp-curr-u: ~ifp^*** (current s) u
assumes vpeq-s-t: ∀ v . ifp^*** v u → vpeq v s t
shows ∀ v . ifp^*** v u → vpeq v (step (next-state s execs) (next-action s execs)) t
proof--
{ fix v
assume ifp-v-uc ifp^*** v u

from ifp-v-uc not-ifp-curr-u have unrelated: ~ifp^*** (current s) v using rtranclp-trans by metis
from this current-next-state[THEN spec,THEN spec,where x1=s] vpeq-reflexive unrelated locally-respects[THEN spec,THEN spec,THEN spec,where x1=next-state s execs and x=v and x2=the (next-action s execs)] prec-s 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 unrelated this locally-respects-next-state vpeq-transitive have vpeq v s (step (next-state s execs) (next-action s execs)) by blast
from this and ifp-v-u and vpeq-s-t and vpeq-symmetric and vpeq-transitive have vpeq v (step (next-state s execs) (next-action s execs)) t by metis
} thus ?thesis by auto
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 can interact with u (the domain for which is purged).

lemma vpeq-ns-nt-ifp-w:
assumes vpeq-s-t: ∀ v . ifp^*** v u → vpeq v s t'
and current-s-t: current s = current t'
sshows precondition (next-state s execs) a ∧ precondition (next-state t' execs) a → (ifp^*** (current s) u → (∀ v . ifp^*** v u → vpeq v (step (next-state s execs) a) (step (next-state t' execs) a)))
proof--
{ fix a
assume prec-s: precondition (next-state s execs) a ∧ precondition (next-state t' execs) a
assume ifp-curr: ifp^*** (current s) u
from vpeq-s-t have vpeq-curr-s-t: ifp^*** (current s) u → vpeq (current s) s t' by auto
from ifp-curr precs
next-state-consistent[THEN spec,THEN spec,where xi=s and x=t'] vpeq-curr-s-t vpeq-s-t
current-next-state current-s-t weakly-step-consistent[THEN spec,THEN spec,THEN spec,THEN spec,where
x3=next-state s execs and x2=next-state t' execs and x=a]
show ∀ v . ifp'' v u → vpeq v (step (next-state s execs) a) (step (next-state t' execs) a)
unfolding step-def precondition-def B-def
by (cases a,auto)
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 purged-a-a2: purged-relation u execs execs2
and prec-s precondition (next-state s execs) (next-action s execs)
and current-s-t: current s = current t'
and vpeq-s-t: ∀ v . ifp'' v u → vpeq v s'
shows ¬ifp'' (current s) u ∧ precondition (next-state t' execs2) (next-action t' execs2) → (∀ v . ifp'' v u
→ vpeq v (step (next-state s execs) (next-action s execs)) (step (next-state t' execs2) (next-action t' execs2))

proof-
{
assume not-ifp: ¬ifp'' (current s) u
assume prec-t: precondition (next-state t' execs2) (next-action t' execs2)
fix a a' v
assume ifp-iff: ifp'' v u
from not-ifp and purged-a-a2 have ¬ifp'' (current s) u unfolding purged-relation-def by auto
from this and ifp-iff have ¬ifp'' (current s) v using rtranclp-trans by metis
from this current-nxt-state[THEN spec,THEN spec,THEN spec,where x1=s and x=execs] prec-s vpeq-reflexive
locally-respects[THEN spec,THEN spec,THEN spec,where x1=next-state s execs and x2=the (next-action s
eecs) and x=v]
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-n: vpeq v s (step (next-state s execs) (next-action s execs))
by blast
from not-ifp-curr-v current-s-t current-nxt-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] vpeq-reflexive
have vpeq v (next-state s execs2) (step (next-state t' execs2) (next-action t' execs2))
unfolding step-def precondition-def B-def
by (cases next-action s execs2,auto)
from not-ifp-curr-v current-s-t current-nxt-state have I: ¬ifp'' (current t') v
using rtranclp-trans by auto
from 0 I locally-respects-next-state vpeq-transitive
have vpeq-t-n: vpeq v t' (step (next-state t' execs2) (next-action t' execs2))
by blast
from vpeq-s-n and vpeq-t-n and vpeq-s-t and ifp-iff and vpeq-symmetric and vpeq-transitive
have vpeq-s-n: vpeq v (step (next-state s execs) (next-action s execs)) (step (next-state t' execs2) (next-action
t' execs2))
by blast
}
thus concludes 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 execs s t u execs2)
    assume interrupt-s : interrupt (Suc n)
    assume IH : (∀ 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-s-nt : ∀ v. ifp∗∗ v u → vpeq v (cswitch (Suc n) s) (cswitch (Suc n) t') by auto
        from current-ns-nt vpeq-s-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-ns-nt vpeq-ns-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(excs (current s))
}

EURO-MILS D31.1
assume \( IH: (\forall u \text{ execs}2, \text{equivalent-states} (\text{Some } s) t u \land \text{purged-relation} u \text{ execs} \text{ execs}2 \implies \text{equivalent-states} (\text{run } n (\text{Some } s) \text{ execs}) (\text{run } n t \text{ execs}2) u) \)

\[
\begin{align*}
&\text{fix } t' \\
&\text{assume } t: t = \text{Some } t' \\
&\text{fix } rs \\
&\text{assume } rs: \text{run } (\text{Suc } n) (\text{Some } s) \text{ execs} = \text{Some } rs \\
&\text{fix } rt \\
&\text{assume } rt: \text{run } (\text{Suc } n) (\text{Some } t') \text{ execs}2 = \text{Some } rt \\
&\text{assume } vpeq-s-t: \forall v. \text{ifp}^* v u \implies vpeq v s t' \\
&\text{assume } current-s-t: current s = current t' \\
&\text{assume } \text{purged-a-a2}: \text{purged-relation} u \text{ execs execs}2
\end{align*}
\]

— 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-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).

\[
\begin{align*}
&\text{from ifp-reflexive and vpeq-s-t have vpeq-s-t-u: vpeq u s t' by auto} \\
&\text{from thread-empty-s and purged-a-a2 and current-s-t have purged-a-na2: ~ifp^* (current t') u \implies} \\
&\text{purged-relation u execs (next-exec t' execs2)} \\
&\text{by (unfold next-exec-def, unfold purged-relation-def, auto)} \\
&\text{from step-atomicity current-next-state current-s-t have current-s-t: current s = current (step (next-state t' execs2))} \\
&\text{unfolding step-def} \\
&\text{by (cases next-action t' execs2, auto)}
\end{align*}
\]

— 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.

\[
\begin{align*}
&\text{have current-rs-rt: current rs = current rt} \\
&\text{proof (cases thread-empty(execs2 (current t')) rule : case-split [case-names t-empty t-not-empty, case t-empty]} \\
&\text{from purged-a-a2 and vpeq-s-t and current-s-t IH[ where t=Some t’ and u=u and ?execs2.0=execs2]} \\
&\text{have equivalent-states (run n (Some s) execs) (run n (Some t') execs2) u using rs rt by (auto)} \\
&\text{from this not-interrupt t-empty thread-empty-s} \\
&\text{show ?thesis using rs rt by (auto)}
\end{align*}
\]

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

\{ fix \( v \) assume \( ia : \text{ ifp}^{**} v u \) have \( vpeq \ v \text{ rs } rt \) proof (cases thread-empty(\( execs2 \) (current \( t \'))) rule :case-split[case-names \( t \)-empty \( t \)-not-empty])
case \( t \)-empty
\[ \text{ from purged-a-a2 and } vpeq \text{-s-t and current-s-t } \text{IH}[\text{where } t=\text{Some } t' \text{ and } u=u \text{ and } ?execs2.0=execs2] \]
\[ \text{ have equivalent-states } (\text{run } n (\text{Some } s ) \text{ execs}) (\text{run } n (\text{Some } t' ) \text{ execs2}) u \text{ using } rs \text{ rt } \text{by(auto)} \]
from \( ia \) this \( t \)-not-interrupt \( t \)-empty thread-empty-s
\[ \text{ show } \text{ ?thesis using } rt \text{ by(auto)} \]
next
case \( t \)-not-empty
\[ \text{ show } \text{ ?thesis} \]
proof (cases precondition (next-state \( t' \) execs2) (next-action \( t' \) execs2) rule :case-split[case-names \( t \)-prec \( t \)-not-prec])
case \( t \)-prec
\[ \text{ from } \text{t-prec current-next-state } \text{ and } vpeq \text{-s-t-u and thread-empty-s and purged-a-a2 and current-s-t} \]
\[ \text{ have not-ifp-curr-t : } \neg \text{ifp}^{*} (\text{current } (\text{next-state } t' \text{ execs2})) u \text{ unfolding purged-relation-def} \]
by auto
\[ \text{ from } \text{t-prec current-next-state locally-respects-next-state } \text{this } \text{and } vpeq \text{-s-t and locally-respects } \text{and} \]
\[ \text{vpeq-s-nt} \]
\[ \text{ have } vpeq \text{-s-nt : } (\forall v. \text{ifp}^{**} v u \rightarrow vpeq v s (\text{step } (\text{next-state } t' \text{ execs2}) (\text{next-action } t' \text{ execs2}))) \text{ by auto} \]
from \( \text{purged-a-na2 this current-s-nt not-ifp-curr-t current-next-state } \text{IH}[\text{where } t=\text{Some } (\text{step } (\text{next-state } t' \text{ execs2}) (\text{next-action } t' \text{ execs2})) (\text{next-execs } t' \text{ execs2})] \)
\[ \text{ have equivalent-states } (\text{run } n (\text{Some } s ) \text{ execs}) (\text{run } n (\text{Some } (\text{step } (\text{next-state } t' \text{ execs2}) (\text{next-action } t' \text{ execs2})))) u \text{ using } rs \text{ rt } \text{by(auto)} \]
next
case \( t \)-not-prec
\[ \text{ thus } \text{ ?thesis using } rt \text{ t-not-empty not-interrupt } \text{by(auto)} \]
qed
qed \}
\from \text{current-rs-rt and this have equivalent-states } (\text{Some } rs) (\text{Some } rt) u \text{ by auto} \}
\[ \text{thus } \text{?case by(simp add:option.splits.cases t.simp+)} \]
next
case \( (5 n \text{ execs } s t u \text{ execs2}) \)
assume \text{not-interrupt: } \neg \text{interrupt } (\text{Suc } n) \)
assume \text{thread-not-empty-s: } \neg \text{thread-empty}(\text{execs } (\text{current } s))
assume \text{not-prec-s: } \neg \text{-precondition } (\text{next-state } s \text{ execs}) (\text{next-action } s \text{ execs})
— Whenever the precondition does not hold, the entire theorem flattens to True and everything holds vacuously.

\[ \text{hence run } (\text{Suc } n) (\text{Some } s ) \text{ execs } = \text{None using } \text{not-interrupt thread-not-empty-s } \text{by simp} \]
— 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, 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 (for: next-s and next-t) are the states after one step.

— Some lemma’s used in the remainder of this case.

from ifp-reflexive and vpeq-s-t have vpeq-s-t-u vpeq u s t’ by auto
from step-atomicity and current-s-t current-next-state
have current-ns-nt: current (step (next-state s execs) (next-action s execs)) = current (step (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 vpeq-s-t have vpeq-curr-s-t: ifp++ (current s) u → vpeq (current s) s t’ by auto
from prec-s involved-ifp[THEN spec,THEN spec,where x1=next-state s execs and x=next-action s execs] vpeq-s-t have vpeq-involved: ifp++ (current s) u → (∀ d ∈ involved (next-action s execs) . vpeq d s t’)

unfolding 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) u → next-execs t’ execs = next-execs s 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) u → 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)

thus ?case by(simp add:option splits)
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.

```isar
have current-rs-rt: current rs = current rt
proof (cases precondition (next-state t' execs2) (next-action t' execs2) 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')) by auto
from current-ns-nt next-execs-t next-action-s-t purged-a-a2
curr-ifp-u prec-t prec-s vpeq-ns-nt-ifp-u
have equivalent-states (Some (step (next-state s execs) (next-action t' execs2)) (Some (step (next-state t' execs2) (next-action t' execs2))) u
unfolding purged-relation-def next-state-def
by auto
from this
\( \text{IH}[\text{where } u=\text{u and } ?\text{execs2}\.0=(\text{next-execs t' execs2}) \text{ and } t=\text{Some (step (next-state t' execs2) (next-action t' execs2))}] \)
current-ns-nt purged-na-na2
have equivalent-states (run n (Some (step (next-state s execs) (next-action t' execs2)) (next-exec s execs)) (next-exec s execs)) (run n (Some (step (next-state t' execs2) (next-action t' execs2)) (next-exec t' execs2))) u
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 curr-not-ifp-u
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 \( \text{IH}[\text{where } u=\text{u and } ?\text{execs2}\.0=(\text{next-execs t' execs2}) \text{ and } t=\text{Some (step (next-state t' execs2) (next-action t' execs2))}] \)
```

---
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' execs) (next-action t' execs))) (next-execs t' execs))

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 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 this not-interrupt thread-not-empty-s t-empty prec-s show ?thesis using rs rt by auto
qed

d { 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 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:**

from assms unwinding-implies-view-partitioned-ind have view-partitioned-inductive view-partitioned-ind by blast

have purged-relation $\forall u \text{ execs}. \text{purged-relation } u \text{ execs (purge } \text{ execs } u)$

by (unfold purged-relation-def, unfold purge-def, auto)

fix execs $s \in u$
assume \(1\) : equivalent-states \(s \leftrightarrow t \leftrightarrow u\)

from this view-partitioned-inductive purged-relation

have equivalent-states \((run n s \text{ execs}) (run n t (\text{purge execs } u)) u\)

unfolding view-partitioned-ind-def by auto

from this ifp-reflexive

have \(\text{run n s execs} \parallel \text{run n t (purge execs } u)\)

unfolding view-partitioned-ind-def by 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-

\{

fix execs a n

from asms unwinding-implies-view-partitioned

have vp : view-partitioned by blast

from vp and vpeq-reflexive

have \(I \vdash u . \text{run n (Some } s0) \text{ execs}

\parallel \text{run n (Some } s0) (\text{purge execs } u)

\rightarrow (\lambda rs rt. vpeq u rs rt \land \text{current } rs = \text{current } rt))

unfolding view-partitioned-def by auto

have \(\text{run n (Some } s0) \text{ execs} \rightarrow (\lambda s-f. \text{run n (Some } s0) (\text{purge execs (current s-f)}) \rightarrow (\lambda s-f2. \text{output-f s-f2 a} = \text{output-f s-f2 a} \land \text{current s-f} = \text{current s-f}))

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) (\text{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 I[THEN spec,where x=\text{current } rs]

have vpeq : vpeq (current rs) rs rt \land \text{current } rs = \text{current } rt

unfolding B-def by auto

from this output-consistent have output-f rs a = output-f rt a

by auto

from this vpeq \(\text{run n (Some } s0) \text{ execs} = \text{Some } rs \text{ } \text{Some}

show ?thesis unfolding B-def by auto

qed

qed

\}

thus ?thesis unfolding NI-unrelated-def by auto

qed

### 3.2.2 Security for indirectly interfering domains

Consider the following security policy over three domains \(A, B\) and \(C\): \(A \sim B \sim C\), but \(A \not\sim C\). 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
The aim of this subsection is to formalize the semantics where $A$ can write to $C$ via $B$ only. We define to 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.

**Definition: Intermediary**

```
definition intermediary :: 'dom-t ⇒ 'dom-t ⇒ bool
where intermediary d u ≡ v. ifp d u ∧ ¬ifp v u ∧ d ≠ u
```

**Primrec: Remove Gateway Communications**

```
primrec remove-gateway-communications :: 'dom-t ⇒ 'action-t execution ⇒ 'action-t execution
where remove-gateway-communications u [] = []
    | remove-gateway-communications u (aseq#exec) = (if ∃ a ∈ set aseq . ∃ v. intermediary v u ∧ v ∈ involved (Some a) then [] else aseq) # (remove-gateway-communications u exec)
```

**Definition: Ipurge-l**

```
definition ipurge-l ::
    ('dom-t ⇒ 'action-t execution) ⇒ 'dom-t ⇒ ('dom-t ⇒ 'action-t execution)
where
ipurge-l execs u ≡ λ d . if intermediary d u then
    []
    else if d = u then
        remove-gateway-communications u (execs u)
    else execs d
```

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.

**Abbreviation: Ind-SOURCE**

```
abbreviation ind-source :: 'dom-t ⇒ 'dom-t ⇒ bool
where
ind-source d u ≡ ifp d u ∧ ¬ifp u d
```

**Definition: Ipurge-r**

```
definition ipurge-r ::
    ('dom-t ⇒ 'action-t execution) ⇒ 'dom-t ⇒ ('dom-t ⇒ 'action-t execution)
where
ipurge-r execs u ≡ λ 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
```

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 $A \rightsquigarrow B \rightsquigarrow C$, but prohibits $A \rightsquigarrow C$.

**Definition: NI-Indirect-Sources**

```
definition NI-indirect-sources :: bool
where
NI-indirect-sources
    ≡ ∀ execs a n. run n (Some s0) execs →
    (λ s-f . (run n (Some s0) (ipurge-l execs (current s-f))) || run n (Some s0) (ipurge-r execs (current s-f))) →
    (λ s-l s-r . 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: ISecure**

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

**Abbreviation: IEquivalent-States**

```
abbreviation iequivalent-states :: 'state-t option ⇒ 'state-t option ⇒ 'dom-t ⇒ bool
where
iequivalent-states s t u ≡ s ∥ t → (λ s t . (∀ v . ifp v u ∧ ¬intermediary v u → vpeq v s t) ∧ current s = current t)
```
**definition** does-not-communicate-with-gateway

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

**definition** iview-partitioned : bool

**where** iview-partitioned \( \equiv \forall \ \text{execs ms mt n u} . \ \text{iequivalent-states ms mt u} \rightarrow (\forall \ v . \ \text{intermediary} v u \rightarrow v \notin \text{involved} (\text{Some} a)) \)

**definition** ipurged-relation1 : dom-t \( \Rightarrow \) (dom-t \( \Rightarrow \) action-t \( \Rightarrow \) (dom-t \( \Rightarrow \) action-t \( \Rightarrow \) bool)

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

Proof that if the current is not an intermediary for u, then all domains involved in the next action are vpeq.

**lemma** vpeq-involved-domains:

**assumes** ifp-curr : ifp (current s) u

and not-intermediary-curr : ~intermediary (current s) u

and no-gateway-comm : does-not-communicate-with-gateway u execs

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

and prec-s : precondition (next-state s execs) (next-action s execs)

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

**proof**

- { fix v
  assume involved : v \in \text{involved} (\text{next-action s execs})
  from this prec-s involved-ifp[THEN spec,THEN spec,where x1=next-state s execs and x=next-action s execs]
  have ifp-v-curr : ifp v (current s)
  using current-next-state
  unfolding involved-def precondition-def B-def
  by (cases next-action s execs,auto)
  have vpeq v s t'
  } proof

  - { assume ifp v u \^ ~intermediary v u
    from this vpeq-s-t
    have vpeq v s t' by (auto)
    }

  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 \sim ifp v u
from ifp-curr not-intermediary-curr ifp-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 \notin involved (Some a)
proof-
{
fix a∶ 'action-t
fix aseq u exec v
have aseq ∈ set (remove-gateway-communications u exec) ∧ a ∈ set aseq ∧ intermediary v u \rightarrow v \notin 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 ∧ iequivalent-states s t u ∧ ipurged-relation1 u execs execs2
\rightarrow 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: (\forall u execs2. does-not-communicate-with-gateway u execs ∧
\[
\text{iequivalent-states (Some (\text{cswitch} (\text{Suc} n) s)) t u \land \text{ipurged-relation1 u execs execs2} \implies \\
\text{iequivalent-states (run n (Some (\text{cswitch} (\text{Suc} n) s)) execs) (run n t execs2) u)}
\]

\[
\{ \text{fix } t' = '\text{state-t} \\
\text{assume } t = \text{Some } t' \\
\text{fix } rs \\
\text{assume } rs: \text{run (Suc n) (Some s) execs = Some rs} \\
\text{fix } rt \\
\text{assume } rt: \text{run (Suc n) (Some t') execs2 = Some rt} \\
\text{assume } \text{no-gateway-comm: does-not-communicate-with-gateway u execs} \\
\text{assume } \text{vpeq-s-t: } \forall v. \text{ifp v u} \land \neg \text{intermediary v u} \implies \text{vpeq v s t'} \\
\text{assume } \text{current-s-t: current s = current t'} \\
\text{assume } \text{purged-a-a2: ipurged-relation1 u execs execs2} \\
\text{from } \text{current-s-t cswitch-independent-of-state} \\
\text{have } \text{current-ns-nt: current (\text{cswitch} (\text{Suc} n) s) = current (\text{cswitch} (\text{Suc} n) t')} \\
\text{by blast} \\
\text{from } \text{cswitch-consistency vpeq-s-t} \\
\text{have } \text{vpeq-ns-nt: } \forall v. \text{ifp v u} \land \neg \text{intermediary v u} \implies \text{vpeq v (\text{cswitch} (\text{Suc} n) s) (\text{cswitch} (\text{Suc} n) t')} \\
\text{by auto} \\
\text{from } \text{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 (\text{cswitch} (\text{Suc} n) t') and ?execs2.0=execs2]} \\
\text{have } \text{current-ss-rt: current rs = current rt using rs rt by(auto)} \\
\{ \text{fix } v \\
\text{assume } ia: \text{ifp v u} \land \neg \text{intermediary v u} \\
\text{from } \text{no-gateway-comm interrupt-s current-ns-nt vpeq-ns-nt vpeq-reflexive ia current-s-t purged-a-a2 IH[where u=u and t=Some (\text{cswitch} (\text{Suc} n) t') and ?execs2.0=execs2]} \\
\text{have } \text{vpeq v rs rt using rs rt by(auto)} \\
\} \\
\text{from } \text{current-ss-rt and this have iequivalent-states (Some rs) (Some rt) u by auto} \\
\} \\
\text{thus } \text{?case by(simp add:option.splits,cases t,simp+)} \\
\text{next} \\
\text{case (4 n execs s t u execs2)} \\
\text{assume } \text{not-interrupt: } \neg \text{interrupt (Suc n)} \\
\text{assume } \text{thread-empty-s: thread-empty(execs (current s))} \\
\text{assume IH: } (\forall t u execs2. \text{does-not-communicate-with-gateway u execs} \land \text{iequivalent-states (Some s) t u} \land \\
\text{ipurged-relation1 u execs execs2} \implies \text{iequivalent-states (run n (Some s) execs) (run n t execs2) u)} \\
\{ \text{fix } t' \\
\text{assume } t: t = \text{Some } t' \\
\text{fix } rs \\
\text{assume } rs: \text{run (Suc n) (Some s) execs = Some rs} \\
\text{fix } rt \\
\text{assume } rt: \text{run (Suc n) (Some t') execs2 = Some rt} \\
\text{assume } \text{no-gateway-comm: does-not-communicate-with-gateway u execs} \\
\text{assume } \text{vpeq-s-t: } \forall v. \text{ifp v u} \land \neg \text{intermediary v u} \implies \text{vpeq v s t'} \\
\text{assume } \text{current-s-t: current s = current t'} \\
\text{assume } \text{purged-a-a2: ipurged-relation1 u execs execs2} \\
\text{from } \text{ifp-reflexive vpeq-s-t have vpeq-u-s-t: vpeq u s t' unfolding intermediary-def by auto} \\
\text{from } \text{step-atomicity current-next-state current-s-t have current-s-nt: current s = current (step (next-state t')}}
\texttt{execs2} \ (\textit{next-action \textit{t'} execs2})

\textbf{unfolding step-def}
\begin{itemize}
  \item by (cases \textit{next-action \textit{s execs}}, cases \textit{next-action \textit{t'} execs2}, \textit{simp simp cases next-action \textit{t'} execs2, simp simp})
  \item from \texttt{vpeq-s-t have vpeq-curr-s-t: \textit{ifp (current \textit{s}) \textit{u and \sim intermediaiy (current \textit{s}) \textit{u \rightarrow vpeq (current \textit{s}) \textit{t'}} by auto}}
  \item have iequivalent-states \ (\textit{run (Suc \textit{n}) (Some \textit{s}) execs}) \ (run (Suc \textit{n}) (Some \textit{t'}) execs2) \textit{u}
  \item proof (cases \textit{thread-empty(execs2 (current \textit{t'}))})
  \item case True
  \begin{itemize}
    \item from \texttt{purged-a-a2 and vpeq-s-t and current-s-t IH[where \textit{t=Some \textit{t'} and \textit{u=u and \?execs2.0=execs2}] no-gateway-comm}
    \item have iequivalent-states \ (\textit{run \textit{n} (Some \textit{s}) execs}) \ (run \textit{n} (Some \textit{t'}) execs2) \textit{u using \textit{rt \textit{by(auto)}}}
    \item from this \texttt{have False using \textit{rs \textit{by(auto)}}}
    \item show \texttt{?thesis using \textit{rs \textit{by(auto)}}}
  \end{itemize}
  \item qed
\end{itemize}

\texttt{from False purged-a-a2 \textit{thread-empty-s current-s-t have I: ind-source (current \textit{t'}) \textit{u or unrelated (current \textit{t'}) \textit{u unfolding ipurged-relation1-def intermediaiy-def by auto}}}
\begin{itemize}
  \item fix \textit{v}
  \item assume \texttt{ifp-v: ifp \textit{v \textit{u}}}
  \item assume \texttt{v-not-intermediaiy: \sim intermediaiy \textit{v \textit{u}}}
\end{itemize}

\texttt{from I ifp-v v-not-intermediaiy have not-ifp-curr-v: \sim ifp (current \textit{t'}) \textit{v unfolding intermediaiy-def by auto}}
\texttt{from not-ifp-curr-v prec-t locally-respects[THEN spec,THEN spec,THEN spec,where \textit{x1=next-state \textit{t'}} execs2 and \textit{x2=the (next-action \textit{t'} execs2)] current-next-state vpeq-reflexive}
\begin{itemize}
  \item have \texttt{vpeq v (next-state \textit{t'} execs2)} \ (\textit{step (next-state \textit{t'} execs2)} (next-action \textit{t'} execs2))
  \item unfolding step-def precondtion-def B-def
  \item by (cases next-action \textit{t'} execs2,auto)
  \item from this vpeq-transitive not-ifp-curr-v locally-respects-next-state
  \item have \texttt{vpeq-tnt: vpeq v \textit{t'} (step (next-state \textit{t'} execs2)} (next-action \textit{t'} execs2))
  \item by blast
  \item from vpeq-s-t ifp-v v-not-intermediaiy vpeq-tnt vpeq-transitive vpeq-symmetric vpeq-reflexive
  \item have \texttt{vpeq v s (step (next-state \textit{t'} execs2)} (next-action \textit{t'} execs2))
  \item by (meets)
\end{itemize}

\texttt{hence \texttt{vpeq-ns-nt: \forall \textit{v \textit{ifp v \textit{u \sim intermediaiy v \textit{u \rightarrow vpeq v \textit{s (step (next-state \textit{t'} execs2)} (next-action \textit{t'} execs2)) by auto}}}}
\texttt{from False purged-a-a2 current-s-t thread-empty-s have purged-a-a2: ipurged-relation1 \textit{u execs (next-execs \textit{t'} execs2)}}
\texttt{unfolding ipurged-relation1-def next-execs-def by(auto)}
\texttt{from vpeq-ns-nt no-gateway-comm}
\texttt{and IH[where \textit{t=Some (step (next-state \textit{t'} execs2) (next-action \textit{t'} execs2)) and \?execs2.0=(next-execs \textit{t'} execs2) and \textit{u=u}]
  \texttt{and current-s-t purged-a-a2}
  \texttt{have eq-ns-nt: iequivalent-states (run \textit{n} (Some \textit{s}) execs)}}
\[
\begin{align*}
\text{(run } n \text{ (Some (step (next-state } t' \text{ execs2) (next-action } t' \text{ execs2))) (next-execs } t')
\end{align*}
\]

\text{execs2)) \text{ by auto}

\text{from prec-t eq-ns-nt not-interrupt False thread-empty-s}

\text{show ?thesis using } t \text{ rs } rt \text{ by(auto)}

\text{qed}

\thus \text{?case by(simp add:option.splits,cases } t,\text{simp+)}

\text{next}

\text{case } (5 \text{ n execs } s \text{ t } u \text{ execs2)}

\text{assume not-interrupt: ¬interrupt (Suc } n)\n
\text{assume thread-not-empty-s: ¬thread-empty(execs (current } s))\n
\text{assume not-prec-s: ¬precondition (next-state } s \text{ execs) (next-action } s \text{ execs)}\n
\text{hence run (Suc } n) (Some } s) \text{ execs } = \text{ None using not-interrupt thread-not-empty-s by simp}

\thus \text{?case by(simp add:option.splits)}

\text{next}

\text{case } (6 \text{ n execs } s \text{ t } u \text{ execs2)}

\text{assume not-interrupt: ¬interrupt (Suc } n)\n
\text{assume thread-not-empty-s: ¬thread-empty(execs (current } s))\n
\text{assume prec-s: precondition (next-state } s \text{ execs) (next-action } s \text{ execs)}\n
\text{assume IH: (∀ } u \text{ execs2. does-not-communicate-with-gateway } u \text{ (next-execs } s \text{ execs) } \land \text{i-equivalent-states (Some (step (next-state } s \text{ execs) (next-action } s \text{ execs))) } t \text{ } u \land \text{i-equivalent-states (run } n \text{ (Some (step (next-state } s \text{ execs) (next-action } s \text{ execs))) (next-execs } s \text{ execs)) (run } n \text{ t execs2) } u)\n
\{\n\text{fix } t'\n\text{assume } t': t = \text{Some } t'\n\text{fix } rs\n\text{assume } rs: \text{run (Suc } n) (\text{Some } s) \text{ execs } = \text{Some } rs\n\text{fix } rt\n\text{assume } rt: \text{run (Suc } n) (\text{Some } t') \text{ execs2 } = \text{Some } rt\n\text{assume no-gateway-comm: does-not-communicate-with-gateway } u \text{ execs}\n\text{assume vpeq-s-t: } \forall \nu. \text{ifp v u } \land \text{¬intermediary } v u \longrightarrow \text{vpeq v s t'}\n\text{assume current-s-t: current } s = \text{current } t'\n\text{assume purged-a-a2: i-equivalent-states (Some (step (next-state } s \text{ execs) (next-action } s \text{ execs))) (next-execs } s \text{ execs)) (run } n \text{ (Some (step (next-state } s \text{ execs) (next-action } s \text{ execs))) (next-execs } s \text{ execs)} (run } n \text{ t execs2) } u)\n
\from \text{ifp-reflexive vpeq-s-t have vpeq-u-s-t: vpeq u s t' unfolding intermediary-def by auto}
\from \text{step-atomicity and current-s-t current-next-state}
\text{have current-ns-nt: current (step (next-state } s \text{ execs) (next-action } s \text{ execs)) = current (step (next-state } t' \text{ execs2) (next-action } t' \text{ execs2)) unfolding step-def}
\text{by (cases next-action s execs,cases next-action } t' \text{ execs2,simp,simp,cases next-action } t' \text{ execs2,simp,simp)}
\from \text{step-atomicity current-next-state current-s-t have current-ns-t: current (step (next-state } s \text{ execs) (next-action } s \text{ execs)) = current } t'\n\text{unfolding step-def}
\text{by (cases next-action s execs,auto)}
\from \text{vpeq-s-t have vpeq-curr-s-t: ifp (current } s) u \land \text{¬intermediary (current } s) u \longrightarrow \text{vpeq (current } s) s t'\n\text{unfolding intermediary-def by auto}
\from \text{from current-s-t purged-a-a2 have eq-execs ifp (current } s) u \land \text{¬intermediary (current } s) u \longrightarrow \text{execs (current } s) = \text{execs2 (current } s)\n\text{by(auto simp add: i-equivalent-states1-def)}
\from \text{from vpeq-involved-domains no-gateway-comm vpeq-s-t vpeq-involved-domains prec-s}
\text{have vpeq-involved: ifp (current } s) u \land \text{¬intermediary (current } s) u \longrightarrow (\forall d \in \text{ involved (next-action } s \text{ execs) . vpeq d s t'})
by blast
from current-s-t next-execs-consistent\[THEN \text{spec}, THEN \text{spec}, THEN \text{spec}, where s2 = s and x1 = t' and x = execs\]
vpeq-curr-s-t vpeq-involved
have next-execs-t \[\text{ifp} (\text{current} s) u \land \neg \text{intermediary} (\text{current} s) u \rightarrow next-execs t' execs = next-execs s execs\]
by (auto simp add: next-execs-def)
from current-s-t and purged-a-a2 and thread-not-empty-s next-action-consistent\[THEN \text{spec}, THEN \text{spec}, where x1 = s and x = t'\]
vpeq-curr-s-t vpeq-involved
have next-action-s-t \[\text{ifp} (\text{current} s) u \land \neg \text{intermediary} (\text{current} s) u \rightarrow next-action t' execs2 = next-action s execs\]
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 \[\text{ifp} (\text{current} s) u \land \neg \text{intermediary} (\text{current} s) u \rightarrow \neg \text{thread-empty}(execs2 (\text{current} t'))\]
unfolding ipurged-relation1-def by auto
have vpeq-ns-nt-1 \[\forall \text{precs} \text{precondition} (\text{next-state s execs}) a \land \neg \text{precondition} (\text{next-state t' execs}) a \rightarrow \text{ifp} (\text{current} s) u \land \neg \text{intermediary} (\text{current} s) u \rightarrow (\forall v . \text{ifp} v u \land \neg \text{intermediary} v u \rightarrow vpeq v (\text{step} (\text{next-state s execs}) a)) \rightarrow \text{step} (\text{next-state t' execs} a)\]
proof-
fix a
assume precs precondition (next-state s execs) a \land \neg precondiion (next-state t' execs) a
assume ifp-curr \[\text{ifp} (\text{current} s) u \land \neg \text{intermediary} (\text{current} s) u\]
from ifp-curr precs
next-state-consistent\[THEN \text{spec}, THEN \text{spec}, where x1 = s and x = t'\]
vpeq-curr-s-t vpeq-s-t
next-state-current-s-t weakly-step-consistent\[THEN \text{spec}, THEN \text{spec}, THEN \text{spec}, THEN \text{spec}, where x3 = next-state s execs and x2 = next-state t' execs and x = \text{the a}\]
show \(\forall v . \text{ifp} v u \land \neg \text{intermediary} v u \rightarrow vpeq v (\text{step} (\text{next-state s execs}) a) \rightarrow (\text{step} (\text{next-state t' execs}) a)\)
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 \in \text{actions-in-execution} (next-execs s execs u)
from this no-gateway-comm [unfolded does-not-communicate-with-gateway-def, THEN \text{spec}, where x = a]
next-execs-subset \[THEN \text{spec}, THEN \text{spec}, THEN \text{spec}, where x2 = s and x1 = execs and x0 = u\]
have \(\forall v . \text{intermediary} v u \rightarrow v \notin \text{involved} \text{ (Some} a)\)
unfolding actions-in-execution-def
by (auto)
}
thus \(\text{thesis}\) unfolding does-not-communicate-with-gateway-def by auto
qed
have inequivalent-states \[\text{run} (\text{Suc} n) \text{ (Some s) execs} \text{ (run} (\text{Suc} n) \text{ (Some} t') \text{ execs2}) u\]
proof \(\text{cases ifp} (\text{current} s) u \land \neg \text{intermediary} (\text{current} s) u \text{ rule :case-split[case-names} T F]\)
case T
show \(\text{thesis}\)
proof \(\text{cases} \text{thread-empty}(execs2 \text{ (current} t')) \text{ rule :case-split[case-names} T2 F2]\)
case F2
show \(\text{thesis}\)
proof \(\text{cases} \text{precondition} \text{ (next-state} t' \text{ execs2} \text{) (next-action} t' \text{ execs2) rule :case-split[case-names} T3 F3]\)
case T3
from T purged-a-a2 current-s-t
next-execs-consistent\[THEN \text{spec}, THEN \text{spec}, where x1 = s and x = t'\]
vpeq-curr-s-t vpeq-involved
have purged-na-na: ipurged-relation1 u (next-execs s execs) (next-execs t' execs2)
unfolding ipurged-relation1-def next-execs-def
by auto
D31.1 – Formal Specification of a Generic Separation Kernel

from IH[where \( t = \text{Some} \text{ (step (next-state } t' \text{ execs2)} \text{ (next-action } t' \text{ execs2)}) \text{ and } \?\text{execs2.0=next-execs } t' \text{ execs2 and } u = u] \]
purged-na-na2 current-ns-nt vpeq-ns-nt-1[where \( a = \text{(next-action } s \text{ execs)} \)] T T3 prec-s

next-action-s-t eq-execs current-s-t no-gateway-comm-na
have eq-ns-nt iequivalent-states \( \text{run } n \text{ (Some (step (next-state } s \text{ execs}) \text{ (next-action } s \text{ execs)})) (next-execs } s \text{ execs}) \)

\( \text{run } n \text{ (Some (step (next-state } t' \text{ execs2)} \text{ (next-action } t' \text{ execs2)))) (next-execs } t' \text{ execs2)} \) \)

\( \text{unfolding next-state-def} \)
by (auto,metis)
from this not-interrupt thread-not-empty-s prec-s F2 T3
have current-rs-rt \( \text{using } rs \text{ rt by auto} \)
{
  fix v
  assume ia: \( \text{ifp } v \text{ u } \wedge \neg \text{intermediary } v \text{ u} \)
  from this eq-ns-nt not-interrupt thread-not-empty-s prec-s F2 T3
  have vpeq v rs rt \( \text{using } rs \text{ rt by auto} \)
}
from this and current-rs-rt show ?thesis using rs rt by auto
next
case F3
from F3 F2 not-interrupt show ?thesis using rt by simp
qed
next

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

case F
hence 1: ind-source (current s) u \( \lor \) unrelated (current s) u \( \lor \) intermediary (current s) u

unfolding intermediary-def
by auto
from purged-a-a2 and thread-not-empty-s
have 2: \( \neg \) intermediary (current s) u unfolding ipurged-relation1-def by auto

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

have prec-t: \( \neg \)thread-empty(execs2 (current t')) \( \Longrightarrow \) precondition (next-state t' execs2) (next-action t' execs2)

proof-
assume thread-not-empty-t: \( \neg \)thread-empty(execs2 (current t'))
{
  assume not-prec-t: \( \neg \)precondition (next-state t' execs2) (next-action t' execs2)
  hence run (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: \( \text{ifp } v \text{ u} \)
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
tus 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)) unfolding step-def precondition-def B-def 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-nt 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 IIF[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-exec 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-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 (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 I: 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 I this iview-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-rtranclp 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 → (λ 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

\{
    \text{fix} \ n \ s \ execs \ t \ execs2
    \text{have} \ mcurrents \ s \ t \minus-\rightarrow \ mcurrents \ (run \ n \ s \ execs) \ (run \ n \ t \ execs2) \\
    \text{proof} \ (\text{induct} \ n \ s \ execs \ \text{arbitrary:} \ t \ execs2 \ \text{rule:} \ \text{run.induct}) \\
    \text{case} \ (1 \ s \ execs \ t \ execs2) \\
    \text{from this} \ \text{show} \ ? \text{case using} \ current-s-t \ \text{unfolding} \ B-def \ \text{by} \ \text{auto} \\
    \text{next} \\
    \text{case} \ (2 \ n \ execs \ t \ execs2) \\
    \text{show} \ ? \text{case unfolding} \ mcurrents-def \ \text{by}(\text{auto}) \\
    \text{next} \\
    \text{case} \ (3 \ n \ s \ execs \ t \ execs2) \\
    \ \text{assume} \ interrupt: \ interrupt \ (Suc \ n) \\
    \ \text{assume} \ \text{IH}: \ (\lambda t \ execs2. \ mcurrents \ (Some \ (cswitch \ (Suc \ n) \ s)) \ t \minus-\rightarrow \ mcurrents \ (run \ n \ (Some \ (cswitch \ (Suc \ n) \ s)) \ execs) \ (run \ n \ t \ execs2)) \\
    \{ \\
    \ \text{fix} \ t' \\
    \ \text{assume} \ t. \ t = (Some \ t') \\
    \ \text{assume} \ curr: \ mcurrents \ (Some \ s) \ t \\
    \ \text{from} \ t \ \text{curr} \ \text{cswitch-independent-of-state}[THEN \ spec,THEN \ spec,THEN \ spec,where \ x1=s] \ \text{have} \ \text{current-ns-nt:} \\
    \ \text{current} \ (cswitch \ (Suc \ n) \ s) = \text{current} \ (cswitch \ (Suc \ n) \ t') \\
    \ \text{unfolding} \ \text{mcurrents-def by simp} \\
    \ \text{from} \ \text{current-ns-nt \ \text{IH}[where} \ t=Some \ (cswitch \ (Suc \ n) \ t') \ \text{and} \ ?\text{execs2.0=execs2]} \\
    \ \text{have} \ \text{mcurrents-ns-nt:} \ \text{mcurrents} \ (run \ n \ (Some \ (cswitch \ (Suc \ n) \ s)) \ execs) \ (run \ n \ (Some \ (cswitch \ (Suc \ n) \ t')) \ execs2) \\
    \ \text{unfolding} \ \text{mcurrents-def by(auto)} \\
    \ \text{from} \ \text{mcurrents-ns-nt \ interrupt \ t} \\
    \ \text{have} \ \text{mcurrents} \ (run \ (Suc \ n) \ (Some \ s) \ execs) \ (run \ (Suc \ n) \ t \ execs2) \\
    \ \text{unfolding} \ \text{mcurrents-def B2-def B-def by(cases \ run \ n \ (Some \ (cswitch \ (Suc \ n) \ s)) \ execs, cases \ run \ (Suc \ n) \ t \ execs2,auto)} \\
    \} \\
    \text{thus} \ ? \text{case unfolding} \ \text{mcurrents-def B2-def by(cases \ t,auto)} \\
    \text{next} \\
    \text{case} \ (4 \ n \ execs \ s \ t \ execs2) \\
    \ \text{assume} \ not-interrupt: \ ¬\text{interrupt} \ (Suc \ n) \\
    \ \text{assume} \ \text{thread-empty-s:} \ \text{thread-empty}(\text{execs} \ (\text{current} \ s)) \\
    \ \text{assume} \ \text{IH}: \ (\\lambda t \ execs2. \ mcurrents \ (Some \ s) \ t \minus-\rightarrow \ mcurrents \ (run \ n \ (Some \ s) \ execs) \ (run \ n \ t \ execs2)) \\
    \{ \\
    \ \text{fix} \ t' \\
    \ \text{assume} \ t. \ t = (Some \ t') \\
    \ \text{assume} \ curr: \ mcurrents \ (Some \ s) \ t \\
    \{ \\
    \ \text{assume} \ \text{thread-empty-t:} \ \text{thread-empty}(\text{execs2} \ (\text{current} \ t')) \\
    \ \text{from} \ t \ \text{curr} \ \text{not-interrupt \ thread-empty-s \ \text{this} \ \text{IH}[where} \ ?\text{execs2.0=execs2 \ and} \ t=Some \ t'] \\
    \ \text{have} \ \text{mcurrents} \ (run \ (Suc \ n) \ (Some \ s) \ execs) \ (run \ (Suc \ n) \ t \ execs2) \\
    \ \text{by} \ \text{auto} \\
    \} \\
    \ \text{moreover} \\
    \{ \\
    \ \text{assume} \ \text{not-prec-t:} \ ¬\text{thread-empty}(\text{execs2} \ (\text{current} \ t')) \ \land \ ¬\text{precondition} \ (\text{next-state} \ t' \ \text{execs2}) \ (\text{next-action} \ t' \ \text{execs2}) \\
    \ \text{from} \ t \ \text{this not-interrupt} \\
    \ \text{have} \ \text{mcurrents} \ (run \ (Suc \ n) \ (Some \ s) \ execs) \ (run \ (Suc \ n) \ t \ execs2) \\
    \ \text{unfolding} \ \text{mcurrents-def by} \ (\text{simp add: rewrite-B2-cases}) \\
    \} \\
    \ \text{moreover}
\[
\begin{align*}
\{ & \quad \textbf{assume step-t:} \:\neg\text{-thread-empty}(execs2 (current t')) \wedge \text{precondition} (\text{next-state t' execs2}) \quad (\text{next-action t' execs2}) \\
& \quad \text{have mcurrents (Some s) (Some (step (next-state t' execs2) (next-action t' execs2)))} \\
& \quad \text{using step-atomicity curr t current-next-state unfolding mcurrents-def} \\
& \quad \text{unfolding step-def by (cases next-action t' execs2.auto)} \\
& \quad \text{from t step-t curr not-interrupt thread-empty-s this IH[where ?execs2.0=next-execs t' execs2 and t=Some (step (next-state t' execs2) (next-action t' execs2))]} \\
& \quad \text{have mcurrents (run (Suc n) (Some s) execs) (run (Suc n) t execs2)} \\
& \quad \text{by auto} \\
& \} \\
& \quad \textbf{ultimately have mcurrents (run (Suc n) (Some s) execs) (run (Suc n) t execs2) by blast} \\
& \} \\
& \quad \textbf{thus ?case unfolding mcurrents-def B2-def by (cases t.auto)} \\
& \} \\
& \textbf{next case (5 n execs s t execs2)} \\
& \quad \textbf{assume not-interrupt-s:} \:\neg\text{-interrupt (Suc n)} \\
& \quad \textbf{assume thread-not-empty-s:} \:\neg\text{-thread-empty (execs (current s))} \\
& \quad \textbf{assume not-prec-s:} \: \neg \text{precondition (next-state s execs) (next-action s execs)} \\
& \quad \textbf{hence run (Suc n) (Some s) execs = None using not-interrupt-s thread-not-empty-s by simp} \\
& \} \\
& \quad \textbf{thus ?case unfolding mcurrents-def by (simp add-option splits)} \\
& \} \\
& \textbf{next case (6 n execs s t execs2)} \\
& \quad \textbf{assume not-interrupt:} \: \neg \text{-interrupt (Suc n)} \\
& \quad \textbf{assume thread-not-empty:} \: \neg \text{-thread-empty (execs (current s))} \\
& \quad \textbf{assume prec-s:} \: \text{precondition (next-state s execs) (next-action s execs)} \\
& \quad \textbf{assume IH:} \: (\\neg execs2.} \\
& \quad \text{mcurrents (Some (step (next-state s execs) (next-action s execs))) t \rightarrow} \\
& \quad \text{mcurrents (run n (Some (step (next-state s execs) (next-action s execs))) (next-execs s execs)) (run n t execs2))} \\
& \{ \\
& \quad \textbf{fix t} \\
& \quad \textbf{assume t:} \quad t = (\text{Some t'}) \\
& \quad \textbf{assume curr:} \quad mcurrents (Some s) t \\
& \{ \\
& \quad \textbf{assume thread-empty-t:} \: \text{-thread-empty (execs2 (current t'))} \\
& \quad \text{have mcurrents (Some (step (next-state s execs) (next-action s execs))) (Some t')} \\
& \quad \text{using step-atomicity curr t current-next-state unfolding mcurrents-def} \\
& \quad \text{unfolding step-def by (cases next-action s execs.auto)} \\
& \quad \text{from t curr not-interrupt thread-not-empty-s prec-s thread-empty-t this IH[where ?execs2.0=execs2 and t=Some t']} \\
& \quad \text{have mcurrents (run (Suc n) (Some s) execs) (run (Suc n) t execs2)} \\
& \quad \text{by auto} \\
& \} \\
& \quad \textbf{moreover} \\
& \{ \\
& \quad \textbf{assume not-prec-t:} \: \neg \text{-thread-empty (execs2 (current t'))} \wedge \neg \text{precondition (next-state t' execs2) (next-action t' execs2)} \\
& \quad \text{from t this not-interrupt} \\
& \quad \text{have mcurrents (run (Suc n) (Some s) execs) (run (Suc n) t execs2)} \\
& \quad \text{unfolding mcurrents-def B2-def by (auto)} \\
& \} \\
& \quad \textbf{moreover} \\
& \{ \\
& \quad \textbf{assume step-t:} \: \neg \text{-thread-empty (execs2 (current t'))} \wedge \text{precondition (next-state t' execs2) (next-action t'}
execs2)

  have mcurrents (Some (step (next-state s execs) (next-action s execs))) (Some (step (next-state t’ execs2) (next-action t’ execs2)))
  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’ execs2, simp, simp, cases next-action t’ execs2, simp, simp)
  from current-next-state t step-t curr not-interrupt thread-not-empty-s prec-s this IH
  [where ?execs2.0=next-execs t’ execs2 and t=Some (step (next-state t’ execs2) (next-action t’ execs2))]
  have mcurrents (run (Suc n) (Some s) execs) (run (Suc n) t execs2)
  by auto

ultimately have mcurrents (run (Suc n) (Some s) execs) (run (Suc n) t execs2) by blast

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 1: ∀ 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-l 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-l)
Some l \ (THEN spec where x=current s-f)
have s I pref s-ipurge-l s-ipurge-r ∧ current s-ipurge-l = current s-ipurge-r
unfolding B-def by auto
from this 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-l)
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-secure:
shows secure
using unwinding-implies-NI-indirect-sources unwinding-implies-NI-unrelated assms unfolding secure-def by(auto)
end
end

3.3 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 \( \gamma \in \text{AS\_set} \) 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.

locale Interruptible_Separation_Kernel = Separation_Kernel kstep output-f s0 current cswitch interrupt kprecondition realistic-execution-control kinvolved ifp vpeq

for kstep :: 'state-t ⇒ 'action-t ⇒ 'state-t
and output-f :: 'state-t ⇒ 'action-t ⇒ 'output-t
and s0 :: 'state-t
and current :: '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 t if an interrupt occurs in the given state at the given time
and kprecondition :: 'state-t ⇒ 'action-t ⇒ bool — Returns t 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
and ifp :: 'dom-t ⇒ 'dom-t ⇒ bool
and vpeq :: 'dom-t ⇒ 'state-t ⇒ 'state-t ⇒ bool

+ fixes AS-set :: ('action-t list) set — Returns a set of valid action sequences, i.e., the attack surface
and invariant :: 'state-t ⇒ bool
and AS-precondition :: 'state-t ⇒ 'dom-t ⇒ 'action-t ⇒ bool
and aborting :: 'state-t ⇒ 'dom-t ⇒ 'action-t ⇒ bool
and waiting :: 'state-t ⇒ 'dom-t ⇒ 'action-t ⇒ bool

assumes empty-in-AS-set ([ ] ∈ AS-set
and invariant-s0: invariant s0
and invariant-after-cswitch ∀ s n . invariant s → invariant (cswitch n s)
and preconditional-after-cswitch ∀ s d n a . AS-precondition s d a → AS-precondition (cswitch n s) d a
and AS-prec-first-action: ∀ s d aseq . invariant s ∧ aseq ∈ AS-set ∧ aseq ≠ [] → AS-precondition s d (hd aseq)
and AS-prec-after-step: ∀ 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'
and AS-prec-dom-independent: ∀ s d a a' . current s ≠ d ∧ AS-precondition s d a → AS-precondition (kstep s a a') d a
and spec-of-invariant: ∀ s a . invariant s → invariant (kstep s a)

and kprecondition-def: kprecondition s a ≡ invariant s ∧ AS-precondition s (current s) a
and realistic-execution-def: realistic-execution aseq ≡ set aseq ⊆ AS-set
and control-spec ∀ s d aseqs . case control s d aseqs of (a,aseqs',s') ⇒ (thread-empty aseqs ∧ (a,aseqs') = (None,[ ])) ∨ (∗ Nothing happens ∗) (aseqs ≠ [] ∧ hd aseqs ≠ [] ∧ ~aborting s' d (the a) ∧ ~waiting s' d (the a) ∧ (a,aseqs') = (Some (hd (hd aseqs)), (tl (hd aseqs))#(tl aseqs))) ∨ (∗ Execute the first action of the current action sequence ∗)
We can now formulate a total run function, since based on the new assumptions the case where the precondition does not hold, will never occur.

**function** run-total :: time-t ⇒ 'state-t ⇒ ('dom-t ⇒ 'action-t execution) ⇒ 'state-t
**where** run-total 0 s execs = s
| interrupt (Suc n) === run-total (Suc n) s execs = run-total n (cswitch (Suc n) s) execs
| ~interrupt (Suc n) === thread-empty(execs (current s)) === run-total (Suc n) s execs = run-total n s execs
| ~interrupt (Suc n) === ~thread-empty(execs (current s)) === run-total (Suc n) s execs = run-total n (step (next-state s execs) (next-action s execs)) (next-execs s 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 :: ('dom-t ⇒ 'action-t execution) ⇒ bool
**where** realistic-executions execs ∀ d . realistic-execution (execs d)

Lemma run_total_equal_run is proven by doing induction. It is however not inductive and can therefore not be proven directly: a realistic execution is not necessarily realistic after performing one action. 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 AS_set). The first action sequence, however, is being executed and is therefore not necessarily an action sequence from AS_set, but it is the last part of some action sequence from AS_set.

**definition** realistic-AS-partial :: 'action-t list ⇒ bool
**where** realistic-AS-partial aseq ≡ ∃ n aseq’ . n ≤ length aseq’ ∧ aseq’ ∈ AS-set ∧ aseq = lastn n aseq’

**definition** realistic-executions-ind :: ('dom-t ⇒ 'action-t execution) ⇒ bool
**where** realistic-executions-ind execs ∀ d . (case execs d of [] ⇒ True | (aseq#aseq) ⇒ 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-execs 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-exec-def by (auto)
}
moreover
{
  let ?ct= execs (current s)
  let ?execs' = (tl (hd ?ct)) # (il ?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-exec-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 ?ct= execs (current s)
  let ?execs' = ?ct
  let ?a' = Some (hd (hd ?ct))
  assume c-waiting: let (a,aseqs',s') = ?c in
  (a,aseqs') = (?a', ?execs')
from na-def c-waiting d-is-curr
  have as-defs: aseq = hd ?execs' ∧ aseqs = tl ?execs'
  unfolding next-exec-def by (auto)
from realistic[unfolded realistic-executions-ind-def,THEN spec,where x=d] d-is-curr set-tl-is-subset[where x=?execs']
  have subset: set (tl ?execs') ⊆ AS-set
  unfolding Let-def realistic-AS-partial-def
  by (cases execs d,auto)
from na-def c-waiting d-is-curr
  have ?execs' ≠ [] unfolding next-exec-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 ≤ length aseq' ∧ aseq' ∈ AS-set ∧ hd(execs d) = lastn n aseq'
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 \land realistic-executions-ind execs \land precondition-ind s execs \implies 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)
}

qed
\begin{verbatim}
assume interrupt: interrupt (Suc n)
assume IH: (\forall s. \text{strict-equal} \ (\text{Some} \ (\text{cswitch} \ (\text{Suc} \ n) \ s)) \ sa \land \ \text{realistic-executions-ind} \ execs \land \ \text{precondition-ind} \ sa \ execs \implies
  \text{strict-equal} \ (\text{run} \ n \ (\text{Some} \ (\text{cswitch} \ (\text{Suc} \ n) \ s)) \ execs) \ (\text{run-total} \ n \ sa \ execs))

\{ 
  assume equal-s-sa: \text{strict-equal} \ (\text{Some} \ s) \ sa 
  assume realistic: \text{realistic-executions-ind} \ execs 
  assume inv-sa: \text{precondition-ind} \ sa \ execs 
  have inv-nsa: \text{precondition-ind} \ (\text{cswitch} \ (\text{Suc} \ n) \ sa) \ execs 

proof
{ 
  fix d 
  have fst (control (cswitch (Suc n) sa) d (execs d)) \implies \text{AS-precondition} \ (\text{cswitch} \ (\text{Suc} \ n) \ sa) \ d 
  using next-action-after-cswitch inv-sa \unfolded \text{precondition-ind-def}.\implies \text{THEN conjunct2}.\implies \text{spec where x=d]}

  \text{precondition-after-cswitch}
  unfolding \text{Let-def B-def \text{precondition-ind-def}}
  by(cases fst (control (cswitch (Suc n) sa) d (execs d)),auto)
}
thus \text{thesis} using inv-sa \text{invariant-after-cswitch} unfolding \text{precondition-ind-def} by auto

qed

from equal-s-sa \text{realistic} inv-nsa IH[where sa=\text{cswitch} \ (\text{Suc} \ n) \ sa]

have equal-ns-nt: \text{strict-equal} \ (\text{run} \ n \ (\text{Some} \ (\text{cswitch} \ (\text{Suc} \ n) \ s)) \ execs) \ (\text{run-total} \ n \ (\text{cswitch} \ (\text{Suc} \ n) \ sa) \ execs)

unfolding \text{strict-equal-def} by(auto)

\}

from this interrupt \text{show \?case by auto}

next case (4 \text{n execs s sa})
  assume not-interrupt \implies \text{~interrupt} \ (\text{Suc} \ n)
  assume thread-empty: \text{thread-empty} \ (\text{execs} \ (\text{current} \ s))
  assume IH: (\forall s. \text{strict-equal} \ (\text{Some} \ s) \ sa \land \ \text{realistic-executions-ind} \ execs \land \ \text{precondition-ind} \ sa \ execs \implies
  \text{strict-equal} \ (\text{run} \ n \ (\text{Some} \ s) \ execs) \ (\text{run-total} \ n \ sa \ execs))
  have current-s-sa: \text{strict-equal} \ (\text{Some} \ s) \ sa \implies \text{current} \ s = \text{current} \ sa \ unfolding \text{strict-equal-def by auto}

\{
  assume equal-s-sa: \text{strict-equal} \ (\text{Some} \ s) \ sa 
  assume realistic: \text{realistic-executions-ind} \ execs 
  assume inv-sa: \text{precondition-ind} \ sa \ execs 
  from equal-s-sa \text{realistic} \text{inv-sa IH[where sa=sa]}
  
  have equal-ns-nt: \text{strict-equal} \ (\text{run} \ n \ (\text{Some} \ s) \ execs) \ (\text{run-total} \ n \ sa \ execs)
  unfolding \text{strict-equal-def by(auto)}

  \}

from this current-s-sa \text{thread-empty not-interrupt \text{show \?case by auto}

next case (5 \text{n execs s sa})
  assume not-interrupt \implies \text{~interrupt} \ (\text{Suc} \ n)
  assume thread-not-empty: \text{~thread-empty} \ (\text{execs} \ (\text{current} \ s))
  assume not-prec \text{~precondition} \ (\text{next-state} \ s \ execs) \ (\text{next-action} \ s \ execs)
  \section{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: \text{strict-equal} \ (\text{Some} \ s) \ sa 
  assume realistic: \text{realistic-executions-ind} \ execs 
  assume inv-sa: \text{precondition-ind} \ sa \ execs 
  from equal-s-sa \text{have s-sa s = sa unfolding \text{strict-equal-def by auto}
  from inv-sa have}
  \text{next-action} \ s \ execs \implies \text{AS-precondition} \ sa \ (\text{current} \ sa)
\end{verbatim}
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 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 have s-sa : s = sa unfolding 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) ?nsa unfolding strict-equal-def by auto
from inv-sa equal-s-sa have inv-s : invariant s unfolding 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 ?a 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 (hd (execs (current s))) ∧ ?na d = (tl (tl (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 ∧ hds(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 consecutive-is-sub-seq[where a=a and b=the ?a' and n=n and y=aseq' and x=tl (tl (hd (execs (current s))))]
This holds, since the control mechanism will ensure that action a’ is the start of a new action sequence in AS-set.

-is None (statement snd-action-not-none), we prove that the precondition is inductive, i.e., it will hold for a’.

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

next-state-precondition not-aborting not-waiting
show ?thesis
by auto

next

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

from 1 2 inv-s
sub-seq-in-prefixes[ where X=AS-set ] Some next-state-invariant
next-state-invariant[ THEN spec,THEN spec,where x=s and x1=d and x=execs d ]

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)

from spec-of-waiting a-def True

have no-step: step s ?a = s unfolding step-def by (cases next-action s execs,auto)

from no-step Some True a-def

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

have 2: AS-precondition s (current s) (the ?a’)
unfolding next-action-def B-def
by (auto)

from a-def na-def this True Some no-step

show ?thesis
unfolding step-def
by (auto)

qed

next

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.
\begin{verbatim}
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]
  control-spec[THEN spec, THEN spec, THEN spec, where x2=s and x1=d and x=execs d]
  have na-def: the '?a' = hd (tl (execs (current s))) ∧ ?na d = tl (execs (current s))
  unfolding next-action-def next-execs-def
  by (auto)
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]
  this
  have l: tl (execs (current s)) \neq [] ∧ hd (tl (execs (current s))) \neq []
  by auto
from this realistic[unfolded realistic-executions-ind-def, THEN spec, where x=d] True thread-not-empty
  have hd (tl (execs (current s))) \in AS-set
  by (cases execs d, auto)
from True snd-action-not-none this
  inv-sa this na-def 1
  AS-prec-first-action[THEN spec, THEN spec, THEN spec, where x2=\ns and x=hd (tl (execs (current s))) and x1=d]
  show ?thesis by auto
  qed
\end{verbatim}
have equal-ns-nt : strict-equal (run n (Some ?ns) ?na) (run-total n (step (next-state sa execs) (next-action sa execs)) (next-execs sa execs)) by(auto)
}
from this current-s-sa thread-not-empty not-interrupt prec show ?case by auto
qed
}
hence thm-inductive : ∀ m s execs n . strict-equal m s ∧ realistic-executions-ind execs ∧ precondition-ind s execs → strict-equal (run n m execs) (run-total n s execs) by blast
have 1: strict-equal (Some s) 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-execution-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-execution-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-execution-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 x2=s and x1=d and x=hd (execs d)]
have AS-precondition s d (the (fst (control s d (execs d))))
using control-spec[THEN spec,THEN spec,THEN spec,where x=execs d and x1=d and x2=s]
by auto
}
hence fst (control s d (execs d)) ↦ AS-precondition s d
unfolding B-def
by (cases fst (control s d (execs d)),auto)
}
from this invariant show ?thesis unfolding precondition-ind-def by auto
qed
from thm-inductive 1 2 3 show ?thesis by auto
qed

Theorem unwinding_implies_isecure gives security for all realistic executions. For unrealistic execu-
tions, it holds vacuously and therefore does not tell us anything. In order to prove security for this
refinement (i.e., for function run_total), we have to prove that purging yields realistic runs.
lemma realistic-purge:
  shows ∀ execs d. realistic-executions execs → realistic-executions (purge execs d)
proof -
  { 
    fix execs d
    assume realistic-executions execs
    hence realistic-executions (purge execs d)
    using someI[where P=realistic-execution and x=execs d]
    unfolding realistic-executions-def purge-def by(simp)
  }
thus ?thesis by auto
qed

lemma remove-gateway-comm-subset:
shows set (remove-gateway-communications d exec) ⊆ set exec ∪ {[]}
by (induct exec, auto)

lemma realistic-ipurge-l:
  shows ∀ execs d. realistic-executions execs → realistic-executions (ipurge-l execs d)
proof -
  { 
    fix execs d
    assume 1: realistic-executions execs
    from empty-in-AS-set remove-gateway-comm-subset[where d=d and exec=execs d] 1 have realistic-executions
    (ipurge-l execs d)
    unfolding realistic-executions-def ipurge-l-def by (auto)
  }
thus ?thesis by auto
qed

lemma realistic-ipurge-r:
  shows ∀ execs d. realistic-executions execs → realistic-executions (ipurge-r execs d)
proof -
  { 
    fix execs d
    assume 1: realistic-executions execs
    from empty-in-AS-set remove-gateway-comm-subset[where d=d and exec=execs d] 1 have realistic-executions
    (ipurge-r execs d)
    unfolding realistic-executions-def ipurge-r-def by (auto)
  }
thus ?thesis by auto
qed

We now have sufficient lemma’s to prove security for run_total. The definition of security is similar
to that in Section 3.2. It now assumes that the executions are realistic and concerns function run_total
instead of function run.

definition NI-unrelated-total::bool
where NI-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 NI-indirect-sources-total::bool
where NI-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-iversec-total:
shows isecure-total
proof−
from assms unwinding-implies-iversec have secure-partial: NI-unrelated unfolding isecure-def by blast
from assms unwinding-implies-iversec have iversec1-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)))
    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-f1 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-total-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-b
Some I 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
  qed
  qed
  } thus ?thesis unfolding NI-indirect-sources-total-def by auto
qed
from NI-unrelated-total NI-indirect-sources-total show ?thesis unfolding isecure-total-def by auto
qed
end
end

3.4 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 [31].

  First, a locale is defined that defines all generic functions and all proof obligations (see Section 2.3 of [31]).
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 if 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 AS-set :: ('action-list' t) 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 action can be executed.

and aborting :: 'state-t ⇒ 'dom-t ⇒ 'action-t ⇒ bool — Returns true if the action is aborted.

and waiting :: 'state-t ⇒ 'action-t ⇒ bool — Returns true if 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: ∀ a. ifp u u

and weakly-step-consistent: ∀ s t u. 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: ∀ s a u. ¬ifp (current s) u ∧ invariant s ∧ AS-precondition s (current s) a → vpeq u s

(kstep s a)

and output-consistent: ∀ a t u. 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: ∀ a 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 preconditions-after-cswitch: ∀ s d n a. AS-precondition s d a → AS-precondition (cswitch n s) d a

and AS-prec-first-action: ∀ s d 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) (current s) a′ d a

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

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

and precondition-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)
3.4.1 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 [31] 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 [31].

```
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::'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 ⇒ '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-exec 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
```
3.4.2 Formulations of security

The definitions of security as presented in Section 2.2 of [31].

```
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 ⇒ '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

abbreviation purge where purge ≡ Separation-Kernel.purge

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

abbreviation ipurge-r where ipurge-r ≡ Separation-Kernel.ipurge-r

definition NI-unrelated :: bool
  where NI-unrelated ≡ \forall execs a n. realistic-executions execs →
    \{(let s-f = run n s0 execs in
      output-f s-f a = output-f (run n s0 (purge execs (current s-f))) a)\}

definition NI-indirect-sources :: bool
  where NI-indirect-sources ≡ \forall 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
```

3.4.3 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 difference 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 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

proof−
{ fix s t execs
  assume vpeq . vpeq (current s) s t
  assume vpeq-involved: \forall d ∈ involved (next-action s execs) . vpeq d s t
  assume current-s-t: current s = current t
  from aborting-consistent current-s-t vpeq
  have aborting t (current s) = aborting s (current s) by auto
  from current-s-t this waiting-consistent vpeq-involved
  have next-action s execs = next-action t execs
  unfolding Kernel.next-action-def
    by (cases s,(current s),execs (current s)) rule: CISK-control.cases,auto
}{ thus ?thesis by auto
qed
```
lemma next-exec-consistent:
shows \( \forall s \ t \ \text{execs} . \ vpeq \ (\text{current} \ s) \ s \ t \land (\forall d \in \text{involved} \ (\text{next-action} \ s \ \text{execs}) . \ vpeq \ d \ s \ t) \land \text{current} \ s = \text{current} \ t \rightarrow \text{fst} \ (\text{snd} \ (\text{CISK-control} \ s \ (\text{current} \ s) \ (\text{execs} \ (\text{current} \ s)))) = \text{fst} \ (\text{snd} \ (\text{CISK-control} \ t \ (\text{current} \ s) \ (\text{execs} \ (\text{current} \ s)))) \)
proof
{ 
fix s t \ \text{execs} 
asume vpeq: vpeq \ (\text{current} \ s) \ s \ t 
asume vpeq-involved: \forall d \in \text{involved} \ (\text{next-action} \ s \ \text{execs}) . \ vpeq \ d \ s \ t 
asume current-s-t: \text{current} \ s = \text{current} \ t 
from aborting-consistent current-s-t vpeq
have 1: aborting \ t \ (\text{current} \ s) = aborting \ s \ (\text{current} \ s) by auto 
from 1 vpeq current-s-t vpeq-involved waiting-consistent [THEN spec,THEN spec,THEN spec,THEN spec,where x3=s and x2=t and x1=current s and x=the (next-action s execs)]
have \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)))) 
unfolding Kernel.next-action-def Kernel.involved-def
by (cases (s,(current s)\text{execs} (current s)) rule: CISK-control.cases,auto split add: split-if-asm)
}
thus \ ?\text{thesis} by auto 
qed

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

lemma current-next-state:
shows \( \forall s \ \text{execs} . \ \text{current} \ (\text{next-state} \ s \ \text{execs}) = \text{current} \ s \)
proof
{ 
fix s \ \text{execs} 
have current \ (\text{next-state} \ s \ \text{execs}) = \text{current} \ s 
unfolding Kernel.next-state-def 
using current-set-error-code 
by (cases (s,(current s)\text{execs} (current s)) rule: CISK-control.cases,auto)
}
thus \ ?\text{thesis} by auto 
qed

lemma locally-respects-next-state:
shows \( \forall s \ u \ \text{execs} . \ \sim\text{ifp} \ (\text{current} \ s) \ u \rightarrow vpeq \ u \ s \ (\text{next-state} \ s \ \text{execs}) \)
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 fst (CISK-control (cswitch n s) d aseqs) = fst (CISK-control s d aseqs)
    using aborting-switch-independent waiting-switch-independent
    by (cases (s, d, aseqs) rule: CISK-control.cases, auto)
  }
  thus ?thesis by auto

qed

lemma next-action-after-next-state:
shows ∀ 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)) = fst (CISK-control s d (execs d))

proof–
{
  fix s execs d aseqs
  assume I: current s ≠ d
  have fst (CISK-control (next-state s execs) d aseqs) = None ∨ fst (CISK-control (next-state s execs) d aseqs) = fst (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 ?thesis
  unfolding Kernel.next-state-def
  using aborting-error-update waiting-error-update
  by (cases (sa, current sa, execs (current sa)) rule: CISK-control.cases, auto split: split-if-asm)
qed

thus ?thesis by auto
qed

lemma next-action-after-step:
shows ∀ s a d aseqs . current s # d → fst (CISK-control (step s a) d aseqs) = fst (CISK-control s d aseqs)
proof−
{ fix s a d aseqs
  assume 1: current s # d
  from this aborting-after-step
  have fst (CISK-control (step s a) d aseqs) = fst (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 ?thesis by auto
qed

lemma next-state-precondition:
shows ∀ s d a execs . AS-precondition s d a → AS-precondition (next-state s execs) d a
proof−
{ fix s d a execs
  assume AS-precondition s d a
  hence 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 ?thesis by auto
qed

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

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

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

\textbf{theorem} \ unwinding-implies-isecure-CISK: \\
\text{shows} \ \text{isecure} \\
\text{proof}− \\
\text{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} \\
\text{proof (unfold-locales)} \\
\text{show} \ \forall \ a \ b \ c \ u. \ \text{vpeq} \ u \ a \ b \ \land \ \text{vpeq} \ u \ b \ c \ \Rightarrow \ \text{vpeq} \ u \ a \ c \\
  \text{using} \ \text{vpeq-transitive by blast} \\
\text{show} \ \forall \ a \ b \ u. \ \text{vpeq} \ u \ a \ b \ \Rightarrow \ \text{vpeq} \ u \ b \ a \\
  \text{using} \ \text{vpeq-symmetric by blast} \\
\text{show} \ \forall \ a \ u. \ \text{vpeq} \ u \ a \ a \\
  \text{using} \ \text{vpeq-reflexive by blast} \\
\text{show} \ \forall \ a \ u. \ \text{vpeq} \ u \ s \ t \\
  \text{using} \ \text{vpeq-reflexive by blast} \\
\text{show} \ \forall \ s \ t \ u. \ \text{vpeq} \ u \ s \ t \ \land \ \text{vpeq} \ (\text{current} \ s) \ s \ t \ \land \ \text{kprecondition} \ s \ a \ \land \ \text{kprecondition} \ t \ a \ \land \ \text{current} \ s = \ \text{current} \ t \\
  \Rightarrow \ \text{vpeq} \ u \ (\text{kstep} \ s \ a) \ (\text{kstep} \ t \ a) \\
  \text{using} \ \text{weakly-step-consistent by blast} \\
\text{show} \ \forall \ a \ s \ u. \ \neg \text{ifp} \ (\text{current} \ s) \ u \ \land \ \text{kprecondition} \ s \ a \ \Rightarrow \ \text{vpeq} \ u \ s \ (\text{kstep} \ s \ a) \\
  \text{using} \ \text{locally-respects by blast} \\
\text{show} \ \forall \ a \ s \ t. \ \text{vpeq} \ (\text{current} \ s) \ s \ t \ \land \ \text{current} \ s = \ \text{current} \ t \ \Rightarrow \ (\text{output-f} \ s \ a) = (\text{output-f} \ t \ a) \\
  \text{using} \ \text{output-consistent by blast} \\
\text{show} \ \forall \ s \ t. \ \text{current} \ (\text{kstep} \ s \ a) = \ \text{current} \ s \\
  \text{using} \ \text{step-atomicity by blast} \\
\text{show} \ \forall \ n \ s \ t. \ \text{current} \ s = \ \text{current} \ t \ \Rightarrow \ \text{current} \ (\text{cswitch} \ n \ s) = \ \text{current} \ (\text{cswitch} \ n \ t) \\
  \text{using} \ \text{cswitch-independent-of-state by blast} \\
\text{show} \ \forall \ u \ s \ n \ s. \ \text{vpeq} \ u \ s \ t \ \Rightarrow \ \text{vpeq} \ u \ (\text{cswitch} \ n \ s) \ (\text{cswitch} \ n \ t) \\
  \text{using} \ \text{cswitch-consistency by blast} \\
\text{show} \ \forall \ s \ t. \ \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} \\
  \Rightarrow \ \text{next-action} \ s \ \text{execs} = \ \text{next-action} \ t \ \text{execs} \\
  \text{using} \ \text{next-action-consistent by blast} \\

\[
\begin{align*}
\text{show } & \forall \ s \ t \ \text{execs.} \quad \text{vpeq (current s) s t }\land (\forall \ d \in \text{involved} (\text{next-action s execs}) \quad \text{vpeq d s t} ) \land \text{current s }= \text{current t }\rightarrow \\
 & \text{fst (snd (CISK-control s (current s) (execs (current s)))) }= \text{fst (snd (CISK-control t (current s) (execs (current s))))}
\end{align*}
\]

\text{using next-execs-consistent by blast}

\text{show } \forall i s t \ u \ \text{execs.} \quad \text{vpeq (current s) s t }\land \text{vpeq u s t }\land \text{current s }= \text{current t }\rightarrow \text{vpeq u (next-state s execs)} \\
(\text{next-state t execs})

\text{using next-state-consistent by auto}

\text{show } \forall i s \execs. \quad \text{current (next-state s execs) }= \text{current s}

\text{using current-next-state by auto}

\text{show } \forall i s u \execs. \quad \text{~ifp (current s) u }\rightarrow \text{vpeq u s (next-state s execs)}

\text{using locally-respects-next-state by auto}

\text{show } [ ] \in \text{AS-set}

\text{using empty-in-AS-set by blast}

\text{show } \forall i s n \ . \quad \text{invariant s }\rightarrow \text{invariant (cswitch n s)}

\text{using invariant-after-cswitch by blast}

\text{show } \forall i s d n a. \quad \text{AS-precondition s d a }\rightarrow \text{AS-precondition (cswitch n s) d a}

\text{using pre-condition-after-cswitch by blast}

\text{show invariant s0}

\text{using invariant-s0 by blast}

\text{show } \forall i s d a aseq. \quad \text{invariant s }\land \text{aseq }\in \text{AS-set }\land \text{aseq }\notin [ ] \rightarrow \text{AS-precondition s d (hd aseq)}

\text{using AS-prec-first-action by blast}

\text{show } \forall i s a a'. \quad (\exists \text{aseq}\in\text{AS-set}. \text{is-sub-seq a a' aseq}) \land \text{invariant s }\land \text{AS-precondition s (current s) a }\land \neg \\
\text{aborting s (current s) a }\land \neg \text{waiting s (current s) a }\rightarrow \\
\text{AS-precondition (kstep s a) (current s) a'}

\text{using AS-prec-after-step by blast}

\text{show } \forall i s d a a'. \quad \text{current s }\notin d \land \text{AS-precondition s d a }\rightarrow \text{AS-precondition (kstep s a') d a}

\text{using AS-prec-dom-independent by blast}

\text{show } \forall i s a \ . \quad \text{invariant s }\rightarrow \text{invariant (kstep s a)}

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

\text{show } \forall s a. \quad \text{kprecondition s a }\equiv \text{kprecondition s a}

\text{by auto}

\text{show } \forall s a. \quad \text{realistic-execution aseq }\equiv \text{set aseq }\subseteq \text{AS-set}

\text{unfolding realistic-execution-def by auto}

\text{show } \forall s a. \quad \forall i d \in \text{involved a. kprecondition s (the a) }\rightarrow \text{ifp d (current s)}

\text{using involved-if unfolding Kernel.involved-def by (auto split: option.splits)}

\text{show } \forall i s \execs. \quad \text{next-action s execs }\rightarrow (\lambda a. \ a \in \text{actions-in-execution (execs (current s))})

\text{using next-action-from-execs by blast}

\text{show } \forall s \execs u. \quad \text{actions-in-execution (next-execs s execs u) }\subseteq \text{actions-in-execution (execs u)}

\text{using next-execs-subset by blast}

\text{show } \forall s d \aseqs.

\text{case CISK-control s d \aseqs of}

\begin{align*}
(a, \aseqs', s') &\Rightarrow \\
\text{thread-empty \aseqs }\land (a, \aseqs') &\Rightarrow (\text{None, [ ]}) \lor \\
\aseqs \notin [ ] \land \text{hd aseqs }\notin [ ] \land \neg \text{aborting s' d (the a) }\land \neg \text{waiting s' d (the a) }\land (a, \aseqs') &\Rightarrow (\text{Some (hd (hd aseqs))}, \text{tl (hd aseqs)} }\notin \text{tl aseqs}) \lor \\
\aseqs \notin [ ] \land \text{hd aseqs }\notin [ ] \land \text{waiting s' d (the a) }\land (a, \aseqs', s') &\Rightarrow (\text{Some (hd (hd aseqs))}, \text{aseqs, s }) \lor (a, \aseqs') = (\text{None, tl aseqs})
\end{align*}

\text{using CISK-control-spec by blast}

\text{show } \forall s n d \aseqs. \quad \text{fst (CISK-control (cswitch n s) d \aseqs) }= \text{fst (CISK-control s d \aseqs)}

\text{using next-action-after-cswitch by auto}

\text{show } \forall s \execs d.

\begin{align*}
\text{current s }\notin d &\rightarrow \\
\text{fst (CISK-control (next-state s execs) d (execs d)) }= \text{None }\lor \text{fst (CISK-control (next-state s execs) d (execs d)) }= \text{fst (CISK-control s d (execs d))}
\end{align*}

\text{using next-action-after-next-state by auto}
show ∀ s d a seqs. current s ≠ d → fst (CISK-control (step s a) d a seqs) = fst (CISK-control s d a seqs)
using next-action-after-step by auto

show ∀ s d a seqs. AS-precondition s d a → AS-precondition (next-state s execs) d a
using next-state-precondition by auto

show ∀ s d a seqs. 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

have run-equals-run-total:
  \(\forall n s execs . \ run n s execs \equiv Interruptible-Separation-Kernel.run-total kstep current cswitch interrupt\)
CISK-control n s execs
proof
  fix n s execs
  show run n s execs \equiv 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 0: Interruptible-Separation-Kernel.isecure-total kstep output-f s0 current cswitch interrupt realistic-execution
CISK-control kinvolved ifp
by (metis int.unwinding-implies-isecure-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
qed

end

4 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 it at a first glance, for example the L4
microkernel API only provided IPC as communication primitive [16]. 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 $sp\subsetset$. 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.

### 4.1 Model of a separation kernel configuration

**theory** Step-configuration  
**imports** Main  
begin

#### 4.1.1 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.

**typedef** partition-id-t
**typedef** thread-id-t

**typedef** page-t — physical address of a memory page
**typedef** 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.

#### 4.1.2 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 $\Rightarrow$ obj-id-t $\Rightarrow$ mode-t $\Rightarrow$ 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 $\Rightarrow$ partition-id-t

end
4.2 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

4.2.1 Specification

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

locale policy-axioms =
fixes \( \text{sp-spec-subj-obj} : \tau \rightarrow \text{obj-id-t} \rightarrow \text{mode-t} \rightarrow \text{bool} \)
and \( \text{sp-spec-subj-subj} : \tau \rightarrow \tau \rightarrow \text{bool} \)
and \( \text{ifp} : \tau \rightarrow \tau \rightarrow \text{bool} \)
assumes \( \text{sp-spec-file-provider} : \forall \ p1 \ p2 \ f \ m1 \ m2 \cdot \)
\( \text{sp-spec-subj-obj} \ p1 \ (\text{FILEP} \ f) \ m1 \land \)
\( \text{sp-spec-subj-obj} \ p2 \ (\text{FILEP} \ f) \ m2 \rightarrow \text{sp-spec-subj-subj} \ p1 \ p2 \)
and \( \text{sp-spec-no-wronly-pages} : \forall \ p \ x \cdot \text{sp-spec-subj-obj} \ p \ (\text{PAGE} \ x) \ \text{WRITE} \rightarrow \text{sp-spec-subj-obj} \ p \ (\text{PAGE} \ x) \ \text{READ} \)
and \( \text{ifp-reflexive} : \forall \ p \cdot \text{ifp} \ p \ p \)
and \( \text{ifp-compatible-with-sp-spec} : \forall \ a \ b \cdot \text{sp-spec-subj-subj} \ a \ b \rightarrow \text{ifp} \ a \ b \land \text{ifp} \ b \ a \)
and \( \text{ifp-compatible-with-ipc} : \forall \ a \ b \ c \ x \cdot \text{sp-spec-subj-subj} \ a \ b \land \text{sp-spec-subj-obj} \ b \ (\text{PAGE} \ x) \ \text{WRITE} \land \text{sp-spec-subj-obj} \ c \ (\text{PAGE} \ x) \ \text{READ} \)
\rightarrow \text{ifp} \ a \ c \)
begin end

4.2.2 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 4.2.1 are satisfied.

locale abstract-policy-derivation =
fixes \( \text{configuration-subj-obj} : \tau \rightarrow \text{obj-id-t} \rightarrow \text{mode-t} \rightarrow \text{bool} \)
begin

definition \( \text{sp-spec-subj-obj} \ a \ x \ m \equiv \)
\( \text{configuration-subj-obj} \ a \ x \ m \lor (\exists \ y \cdot x = \text{PAGE} \ y \land m = \text{READ} \land \text{configuration-subj-obj} \ a \ x \ \text{WRITE}) \)

definition \( \text{sp-spec-subj-subj} \ a \ b \equiv \)
\( \exists \ f \ m1 \ m2 \cdot \text{sp-spec-subj-obj} \ a \ (\text{FILEP} \ f) \ m1 \land \text{sp-spec-subj-obj} \ b \ (\text{FILEP} \ f) \ m2 \)

definition \( \text{ifp} \ a \ b \equiv \)
\( \text{sp-spec-subj-subj} \ a \ b \lor \text{sp-spec-subj-subj} \ b \ a \lor (\exists \ c \ y \cdot \text{sp-spec-subj-subj} \ a \ c \land \text{sp-spec-subj-obj} \ c \ (\text{PAGE} \ y) \ \text{WRITE} \)

Show that the policies specified in Section 4.2.1 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:

\[ p \lor p1 p2 f m1 m2 \]

\[ sp-spec-subj-obj p1 (FILEP f) m1 \land \\
sp-spec-subj-obj p2 (FILEP f) m2 \rightarrow sp-spec-subj-subj p1 p2 \]

unfolding sp-spec-subj-subj-def by auto

show sp-spec-no-wronly-pages:

\[ \forall p x. sp-spec-subj-obj p (PAGE x) WRITE \rightarrow sp-spec-subj-obj p (PAGE x) READ \]

unfolding sp-spec-subj-obj-def by auto

show ifp-reflexive:

\[ \forall p. ifp p p \]

unfolding ifp-def by auto

show ifp-compatible-with-sp-spec:

\[ \forall a b. sp-spec-subj-subj a b \rightarrow ifp a b \land ifp b a \]

unfolding ifp-def by auto

show ifp-compatible-with-ipc:

\[ \forall a b x. (sp-spec-subj-subj a b \land \\
sp-spec-subj-obj b (PAGE x) WRITE \land sp-spec-subj-obj c (PAGE x) READ) \rightarrow ifp a c \]

unfolding ifp-def by auto

qed

end

**Type-synonym**

sp-subj-subj-t = partition-id-t \rightarrow partition-id-t \Rightarrow bool

sp-subj-obj-t = partition-id-t \Rightarrow obj-id-t \Rightarrow mode-t \Rightarrow bool

**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 \[ \forall p. Policy.ifp p p \]

using Policy-properties.ifp-reflexive by auto

end

4.3 Separation kernel state and atomic step function

**Theory**

Step

imports Step-policies

begin

4.3.1 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 \mid RECV

ipc-stage-t = PREP \mid WAIT \mid BUF page-t
**4.3.2 System state**

typedec obj — 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 — values of all objects
thread :: thread-id-t ⇒ thread-t — internal state of threads

Later (Section 4.4), 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_...).

**4.3.3 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 ( obj := 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 := λ q q’ q” . ( part-id = q ∧ obj-id = q’ ∧ m = q” )

∨ sp-impl-subj-obj s 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 := λ q q’ . ( p = q ∧ p’ = q’ ) ∨ sp-impl-subj-subj s 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 ev-signal-precondition = thread-id-t => thread-id-t => state-t => bool where
ev-signal-precondition tid partner s ≡
(sp-impl-subj-subj s (partition tid) (partition partner))

definition atomic-step-ev-signal :: thread-id-t => thread-id-t => state-t => state-t where
atomic-step-ev-signal tid s =
  s (|| thread := fun-upd (thread s) partner (thread s partner (|| ev-counter := Suc (ev-counter (thread s partner)) ) ) ) )

definition atomic-step-ev-wait-one :: thread-id-t => state-t => state-t where
atomic-step-ev-wait-one tid s =
  s (|| thread := fun-upd (thread s) tid (thread s tid (|| ev-counter := (ev-counter (thread s tid) − 1)) ) ) )

definition atomic-step-ev-wait-all :: thread-id-t => state-t => state-t where
atomic-step-ev-wait-all tid s =
  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 aborting :: state-t => thread-id-t => int-point-t => bool where
aborting s tid a ≡ case a of SK-IPC dir PREP partner page ⇒
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 waiting :: state-t ⇒ thread-id-t ⇒ int-point-t ⇒ bool
where waiting s tid a ≡
case a of
  SK-IPC dir WAIT partner page ⇒
    ~ipc-precondition partner (opposite-ipc-direction dir) tid (SOME page′ . True) s
  SK-EV-WAIT EV-PREP - ⇒ False
  SK-EV-WAIT EV-WAIT - ⇒ ev-counter (thread s tid) = 0
  SK-EV-WAIT EV-FINISH - ⇒ False
| - ⇒ 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 4.8). 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 atomic-step :: state-t ⇒ int-point-t ⇒ state-t where
atomic-step s ipt ≡
case ipt of
  SK-IPC dir stage partner page ⇒
    atomic-step-ipc (current s) dir stage partner page s
  SK-EV-WAIT EV-PREP consume ⇒ s
  SK-EV-WAIT EV-WAIT consume ⇒ s
  SK-EV-WAIT EV-FINISH consume ⇒
    case consume of
      EV-CONSUME-ONE ⇒ atomic-step-ev-wait-one (current s) s
      EV-CONSUME-ALL ⇒ atomic-step-ev-wait-all (current s) s
  SK-EV-SIGNAL EV-SIGNAL-PREP partner ⇒ s
  SK-EV-SIGNAL EV-SIGNAL-FINISH partner ⇒
    atomic-step-ev-signal (current s) partner s
  NONE ⇒ s

end

4.4 Preconditions and invariants for the atomic step

definition sp-subset s ≡
(∀ p1 p2 . sp-impl-subj-subj s p1 p2 ⇒ Policy.sp-spec-subj-subj p1 p2)
∧ (∀ p1 p2 m . sp-impl-subj-obj s p1 p2 m ⇒ Policy.sp-spec-subj-obj p1 p2 m)

The following predicate expresses the precondition for the atomic step. The precondition depends on the type of the atomic action.
**definition** atomic-step-precondition :: state-t ⇒ thread-id-t ⇒ int-point-t ⇒ bool

where

atomic-step-precondition s tid ipt ≡

case ipt of

- SK-IPC dir WAIT partner page ⇒
  (* the thread managed it past PREP stage *)
  ipc-precondition tid dir partner page s

- SK-IPC dir (BUF page') partner page ⇒
  (* both the calling thread and its communication partner
    managed it past PREP and WAIT stages *)
  ipc-precondition tid dir partner page s
  ∧ ipc-precondition partner (opposite-ipc-direction dir) tid page' s

- SK-EV-SIGNAL EV-SIGNAL-FINISH partner ⇒
  ev-signal-precondition tid partner s

| - ⇒
  (* No precondition for other interrupt points. *)
  True

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

**definition** atomic-step-invariant :: state-t ⇒ bool

where

atomic-step-invariant s ≡

sp-subset s

### 4.4.1 Atomic steps of SK_IPC preserve invariants

**lemma** set-object-value-invariant:

shows atomic-step-invariant s = atomic-step-invariant (set-object-value ob va s)

proof −

show ?thesis using assms

unfolding atomic-step-precondition-def ipc-precondition-def

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** set-thread-value-invariant:

shows atomic-step-invariant s = atomic-step-invariant (s | thread := thrst |)

proof −

show ?thesis using assms

unfolding atomic-step-precondition-def ipc-precondition-def

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

\begin{verbatim}
case BUF
  show \( ?\text{thesis} \)
  using assms BUF set-object-value-invariant
  unfolding atomic-step-ipc-def
  by (simp split add: ipc-direction-t.splits)
qed
qed

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

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

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

4.4.2 \textbf{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.

\begin{verbatim}
theorem atomic-step-preserves-invariants:
  fixes \( s :: \text{state-t} \)
  and tid :: \text{thread-id-t}
  assumes atomic-step-invariant \( s \)
  shows atomic-step-invariant (atomic-step \( s a \))
proof (cases a)
  case SK-IPC
    then show \( ?\text{thesis} \) unfolding atomic-ipc-preserves-invariants
    using assms atomic-ipc-preserves-invariants
    by simp
  next case (SK-EV-WAIT ev-wait-stage consume)
\end{verbatim}
then show ?thesis
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

4.5 The view-partitioning equivalence relation

theory Step-ypeq
  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.

\[
\text{definition} \quad \text{vpeq-obj} :: \text{partition-id-t} \Rightarrow \text{state-t} \Rightarrow \text{state-t} \Rightarrow \text{bool} \quad \text{where} \\
\text{vpeq-obj} \equiv \forall \text{obj-id}. (\text{Policy.sp-spec-subj-obj u obj-id READ} \implies (\text{obj s}) \text{obj-id} = (\text{obj t}) \text{obj-id})
\]

\[
\text{definition} \quad \text{vpeq-subj-subj} :: \text{partition-id-t} \Rightarrow \text{state-t} \Rightarrow \text{state-t} \Rightarrow \text{bool} \quad \text{where} \\
\text{vpeq-subj-subj} \equiv \forall \text{v}. (\text{Policy.sp-spec-subj-subj u v \implies sp-impl-subj-subj s u v = sp-impl-subj-subj t u v}) \land (\text{Policy.sp-spec-subj-subj v u \implies sp-impl-subj-subj s v u = sp-impl-subj-subj t v u})
\]

\[
\text{definition} \quad \text{vpeq-subj-obj} :: \text{partition-id-t} \Rightarrow \text{state-t} \Rightarrow \text{state-t} \Rightarrow \text{bool} \quad \text{where} \\
\text{vpeq-subj-obj} \equiv \forall \text{ob m}. (\text{Policy.sp-spec-subj-obj u ob m \implies sp-impl-subj-obj s u ob m = sp-impl-subj-obj t u ob m}) \land (\text{Policy.sp-spec-subj-obj p1 ob \ PROVIDE} \land (\text{Policy.sp-spec-subj-obj u ob READ} \lor \text{Policy.sp-spec-subj-obj u ob WRITE} )) \implies (\text{sp-impl-subj-obj s p1 ob \ PROVIDE} = \text{sp-impl-subj-obj t p1 ob \ PROVIDE})
\]

\[
\text{definition} \quad \text{vpeq-local} :: \text{partition-id-t} \Rightarrow \text{state-t} \Rightarrow \text{state-t} \Rightarrow \text{bool} \quad \text{where} \\
\text{vpeq-local} \equiv \forall \text{tid}. (\text{partition tid} = u \implies (\text{thread s tid}) = (\text{thread t tid}))
\]

\[
\text{definition} \quad \text{vpeq} \equiv \text{vpeq-obj} \land \text{vpeq-subj-subj} \land \text{vpeq-subj-obj} \land \text{vpeq-local}
\]

4.5.1 Elementary properties

\[
\text{lemma} \quad \text{vpeq-rel:} \\
\text{shows} \quad \text{vpeq-refl} :: \text{vpeq u s s} \land \text{vpeq-sym} :: \text{vpeq u s t} \implies \text{vpeq u t s} \land \text{vpeq-trans} :: \text{[trans]} : [\text{vpeq u s1 s2 ; vpeq u s2 s3}] \implies \text{vpeq u s1 s3}
\]

\[
\text{unfolding} \quad \text{vpeq-def vpeq-obj-def vpeq-subj-subj-def vpeq-subj-obj-def vpeq-local-def} \quad \text{by auto}
\]

Auxiliary equivalence relation.

\[
\text{lemma} \quad \text{set-object-value-ign:} \\
\text{assumes} \quad \text{eq-obs} : \sim (\text{Policy.sp-spec-subj-obj u x READ}) \land (\text{set-object-value x y s}) \quad \text{shows} \quad \text{vpeq u s}
\]

\[
\text{proof} \quad \text{from} \quad \text{assms} \quad \text{show} \quad ?\text{thesis} \quad \text{unfolding} \quad \text{vpeq-def vpeq-obj-def vpeq-subj-subj-def vpeq-subj-obj-def vpeq-local-def} \quad \text{by auto}
\quad \text{qed}
\]

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

\[
\text{theorem} \quad \text{cswitch-consistency-and-respect:} \\
\text{fixes} \quad u :: \text{partition-id-t} \land s :: \text{state-t} \land \text{new-current :: thread-id-t}
\]
assumes atomic-step-invariant s
shows vpeq u s (s \{ current := 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

4.6 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 [31]).

4.6.1 Locally respects of atomic step functions

lemma ipc-respects-policy:
  assumes nos \sim 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)
case PREP
  thus ?thesis
  unfolding atomic-step-ipc-def
  using vpeq-refl by simp
next
case WAIT
  thus ?thesis
  unfolding atomic-step-ipc-def
  using vpeq-refl by simp
next case (BUF mypage)
  show ?thesis
  proof(cases dir)
case RECV
  thus ?thesis
  unfolding atomic-step-ipc-def
  using vpeq-refl BUF by simp
next
case SEND
  have Policy.sp-spec-subj-subj (partition tid) (partition partner)
  and Policy.sp-spec-subj-obj (partition partner) (PAGE mypage) WRITE
  using BUF SEND inv prec ipt-case
  unfolding atomic-step-invariant-def sp-subset-def
  unfolding atomic-step-precondition-def ipc-precondition-def opposite-ipc-direction-def
  by auto
  hence nos \sim Policy.sp-spec-subj-obj u (PAGE mypage) READ
  using no Policy-properties.ifp-compatible-with-ipc
  by auto
thus \( ?\text{thesis} \)

using BUF SEND assms

unfolding atomic-step-ipc-def set-object-value-def

unfolding vpeq-def vpeq-obj-def vpeq-subj-obj-def vpeq-subj-subj-def vpeq-local-def

by auto

qed

qed

lemma ev-signal-respects-policy:

assumes no \( \sim \) 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 –

from assms have \( \sim \) sp-impl-subj-subj \( s \) (partition tid) \( u \)

unfolding Policy.ifp-def atomic-step-invariant-def sp-subset-def

by auto

with prec have \( \sim \) \( \{\text{partition partner}\} \neq u \)

unfolding atomic-step-precondition-def ev-signal-precondition-def

by (auto simp add: ev-signal-stage-t.splits)

then have \( 2\):vpeq-local \( u \) \( s \) (atomic-step-ev-signal tid partner \( s \))

unfolding vpeq-local-def atomic-step-ev-signal-def

by simp

have \( 3\):vpeq-obj \( u \) \( s \) (atomic-step-ev-signal tid partner \( s \))

unfolding vpeq-obj-def atomic-step-ev-signal-def

by simp

have \( 4\):vpeq-subj-obj \( u \) \( s \) (atomic-step-ev-signal tid partner \( s \))

unfolding vpeq-subj-obj-def atomic-step-ev-signal-def

by simp

have \( 5\):vpeq-subj-subj \( u \) \( s \) (atomic-step-ev-signal tid partner \( s \))

unfolding vpeq-subj-subj-def atomic-step-ev-signal-def

by simp

with \( 2\) \( 3\) \( 4\) \( 5\) show \( ?\text{thesis} \)

unfolding vpeq-def

by simp

qed

lemma ev-wait-all-respects-policy:

assumes no \( \sim \) Policy.ifp (partition tid) \( u \)

and inv: atomic-step-invariant \( s \)

and prec: atomic-step-precondition \( s \) tid ipt

and ipt-case: \( ipt = \) SK-EV-WAIT ev-wait-stage EV-CONSUME-ALL

shows vpeq \( u \) \( s \) (atomic-step-ev-wait-all tid \( s \))

proof –

from assms have \( \sim \) \( \{\text{partition tid}\} \neq u \)

unfolding Policy.ifp-def

by simp

then have \( 2\):vpeq-local \( u \) \( s \) (atomic-step-ev-wait-all tid \( s \))

unfolding vpeq-local-def atomic-step-ev-wait-all-def

by simp

have \( 3\):vpeq-obj \( u \) \( s \) (atomic-step-ev-wait-all tid \( s \))

unfolding vpeq-obj-def atomic-step-ev-wait-all-def

by simp

have \( 4\):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 5: 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 no: ~ Policy.ifp (partition tid) u
and inv: atomic-step-invariant s
and prec: atomic-step-precondition s tid ipt
and ipt-case: 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 1:(partition tid) # u
unfolding Policy.ifp-def
by simp
then have 2: vpeq-local u s (atomic-step-ev-wait-one tid s)
unfolding vpeq-local-def atomic-step-ev-wait-one-def
by simp
have 3: 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

4.6.2 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: ~ Policy.ifp (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
4.7 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 \textit{atomic-step-invariant} holds (see [31]).

4.7.1 Weak step consistency of auxiliary functions

lemma ipc-precondition-weakly-step-consistent:
assumes eq-tid :: vpeq \((\text{partition tid})\) \(s1\) \(s2\)
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 \(\Rightarrow\) tid \(\divides\) alt0
  let \(?receiver\) = case dir of SEND \(\Rightarrow\) partner \(\divides\) alt0
  let \(?local-access-mode\) = case dir of SEND \(\Rightarrow\) READ \(\divides\) alt0
  hence \(?A\) = sp-impl-subj-subj \(s1\) (partition \(?sender\)) (partition \(?receiver\)) = sp-impl-subj-subj \(s2\) (partition \(?sender\)) (partition \(?receiver\))
  and \(?B\) = sp-impl-subj-obj \(s1\) (partition tid) (PAGE page) ?local-access-mode
      = sp-impl-subj-obj \(s2\) (partition tid) (PAGE page) ?local-access-mode
  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 \(?subset\) \(s1\) and \(?subset\) \(s2\)
        unfolding atomic-step-invariant-def sp-subset-def by auto
      hence \(?A\) = sp-impl-subj-subj \(s1\) (partition \(?sender\)) (partition \(?receiver\))
      and \(?B\) = sp-impl-subj-obj \(s1\) (partition tid) (PAGE page) ?local-access-mode
      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
      hence \(?subset\) \(s1\) and \(?subset\) \(s2\)
        using inv1 inv2 unfolding atomic-step-invariant-def sp-subset-def by auto
      hence \(?A\) = sp-impl-subj-subj \(s1\) (partition tid) (PAGE page) ?local-access-mode
      and \(?B\) = sp-impl-subj-obj \(s1\) (partition tid) (PAGE page) ?local-access-mode
      unfolding sp-subset-def by auto
      thus \(?B\) by auto
    qed
  qed

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

lemma ev-signal-precondition-weakly-step-consistent:
assumes eq-tid: vpeq \((\text{partition tid})\) \(s1\) \(s2\)
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)
and ?A = sp-impl-subj-subj s2 (partition tid) (partition partner)

have A: ?A

proof (cases Policy.sp-spec-subj-subj (partition tid) (partition partner))

next 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 (cases x = x')

next case False

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-obj u ?s1' ?s2' unfolding vpeq-obj-def by auto

qed

have E4: vpeq-obj u ?s1' ?s2'

proof –

from 2 3 4 show obj ?s1' x' = obj ?s2' x'
by simp

thus vpeq-obj u ?s1' ?s2' unfolding vpeq-obj-def by auto

qed

have E5: vpeq-subj-subj u ?s1' ?s2'

proof –

EURO-MILS D31.1 Page 72 of 94
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

4.7.2 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 dir stage partner page s1)  
(atomic-step-ipc tid dir stage partner page s2)  
proof –  
  have \( \forall \) mypage . \{ \text{dir} = \text{SEND}; \text{stage} = \text{BUF mypage} \} \implies \text{thesis}  
proof –  
  fix mypage  
  assume dir-send: dir = \text{SEND}  
  assume stage-buf: stage = \text{BUF mypage}  
  have Policy.sp-spec-subj-obj (partition tid) (PAGE page) READ  
  using inv1 prec1 dir-send stage-buf ipt-case  
  unfolding atomic-step-invariant-def sp-subset-def  
  unfolding 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 dir stage partner page s1)  
  (atomic-step-ipc tid dir stage partner page s2)  
  using dir-send stage-buf eq-obs set-object-value-consistent  
  unfolding atomic-step-ipc-def  
  by auto  
  qed  
  thus \text{thesis}  
  using eq-obs unfolding atomic-step-ipc-def  
  by \{ \text{cases stage, auto, cases dir, auto} \}  
  qed

lemma ev-wait-one-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
D31.1 – Formal Specification of a Generic Separation Kernel

and prec1: atomic-step-precondition s1 (current s1) ipt
and prec2: atomic-step-precondition s1 (current s1) ipt
shows vpeq u
  (atomic-step-ev-one tid s1)
  (atomic-step-ev-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-one-def
by simp

lemma ev-wait-all-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 (current s1) ipt
and prec2: atomic-step-precondition s1 (current s1) ipt
shows vpeq u
  (atomic-step-ev-wait-all tid s1)
  (atomic-step-ev-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-wait-all-def
by simp

lemma ev-signal-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 (current s1) ipt
and 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
atomic-step-ev-signal-def
by simp

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

definition extend-f :: (partition-id-t ⇒ partition-id-t ⇒ bool)⇒ (partition-id-t ⇒ partition-id-t ⇒ bool)⇒ (partition-id-t ⇒ partition-id-t ⇒ bool) where
  extend-f f g ≡ λ p1 p2 . f p1 p2 ∨ g p1 p2

definition extend-subj-subj :: (partition-id-t ⇒ partition-id-t ⇒ bool)⇒ state-t ⇒ state-t where
  extend-subj-subj s f s ≡ s (sp-impl-subj-subj −> extend-f (sp-impl-subj-subj) s)

lemma extend-subj-subj-consistent:
  fixes f :: partition-id-t ⇒ partition-id-t ⇒ bool
  assumes vpeq u s1 s2
  shows vpeq u (extend-subj-subj f s1) (extend-subj-subj f s2)
  proof
    let ?g1 = sp-impl-subj-subj s1 and ?g2 = sp-impl-subj-subj s2
    have ∀ v . Policy.sp-spec-subj-subj u v −> ?g1 u v = ?g2 u v
    and ∀ v . Policy.sp-spec-subj-subj v u −> ?g1 v u = ?g2 v u
    using assms unfolding vpeq-def vpeq-subj-subj-def by auto
4.7.3 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).

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

4.8 Separation kernel model

\[ \text{theory Separation-kernel-model} \]
\[ \text{imports [...]/step/Step} \]
\[ .../.../step/Step-invariants \]
\[ .../.../step/Step-vpeq \]
\[ .../.../step/Step-vpeq-locally-respects \]
\[ .../.../step/Step-vpeq-weakly-step-consistent \]
\[ \text{CISK} \]
\[ \text{begin} \]

First (Section 4.8.1) 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 [31]. For example, ‘rifp’ is the instantiation of the generic ‘ifp’.

Later (Section 4.8.5) all CISK proof obligations are discharged, e.g., weak step consistency, output consistency, etc. These will be used in Section 4.9.

4.8.1 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.

\[\begin{align*}
\text{consts} & \\
initial-current & :: \text{thread-id-t} \\
initial-obj & :: \text{obj-id-t} \Rightarrow \text{obj-t}
\end{align*}\]

\[\text{definition } s_0 :: \text{state-t where}\]
\[s_0 \equiv (\text{sp-impl-subj-subj} = \text{Policy.sp-spec-subj-subj,}\]
\[\text{sp-impl-subj-obj} = \text{Policy.sp-spec-subj-obj,}\]
\[\text{current} = \text{initial-current,}\]
\[\text{obj} = \text{initial-obj,}\]
\[\text{thread} = \lambda - . (| \text{ev-counter} = 0 |)\]

\[\text{lemma } \text{initial-invariant:}\]
\[\text{shows } \text{atomic-step-invariant } s_0\]
\[\text{proof -}\]
\[\text{have } \text{sp-subset } s_0\]
\[\text{unfolding } \text{sp-subset-def } s_0\text{-def by auto}\]
\[\text{thus } ?\text{thesis}\]
\[\text{unfolding } \text{atomic-step-invariant-def by auto}\]
\[\text{qed}\]

4.8.2 Types for instantiation of the generic model

To simplify formulations, we include the state invariant \(\text{atomic-step-invariant}\) in the state data type. The initial state \(s_0\) serves at witness that \(rstate-t\) is non-empty.

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

\[\text{definition } abs :: \text{state-t} \Rightarrow rstate-t (\uparrow -) \text{ where } abs = \text{Abs-rstate-t}\]
\[\text{definition } rep :: rstate-t \Rightarrow \text{state-t} (\downarrow -) \text{ where } rep = \text{Rep-rstate-t}\]

\[\text{lemma } rstate-invariant:\]
\[\text{shows } \text{atomic-step-invariant } (\downarrow s)\]
\[\text{unfolding } \text{rep-def by (metis Rep-rstate-t mem-Collect-eq)}\]

\[\text{lemma } rstate-down-up[simp]:\]
\[\text{shows } (\uparrow s) = s\]
\[\text{unfolding } \text{rep-def abs-def using Rep-rstate-t-inverse by auto}\]

\[\text{lemma } rstate-up-down[simp]:\]
\[\text{assumes } \text{atomic-step-invariant } s\]
\[\text{shows } (\downarrow s) = s\]
\[\text{using } \text{assms Abs-rstate-t-inverse unfolding rep-def abs-def by auto}\]

A CISK action is identified with an interrupt point.
type-synonym raction-t = int-point-t

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

definition rstep :: rstate-t \(\Rightarrow\) raction-t \(\Rightarrow\) rstate-t where
rstep s a = \(\uparrow\) (atomic-step ↓ s) a

Each CISK domain is identified with a thread id.

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 \(\Rightarrow\) visible-obj-t
definition routput-f :: rstate-t \(\Rightarrow\) raction-t \(\Rightarrow\) 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 \(\Rightarrow\) rdom-t \(\Rightarrow\) raction-t \(\Rightarrow\) 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 \(\Rightarrow\) rstate-t \(\Rightarrow\) rstate-t \(\Rightarrow\) bool where
rvpeq u s1 s2 = vpeq (partition u) ↓ s1 ↓ s2

definition rifp :: rdom-t \(\Rightarrow\) rdom-t \(\Rightarrow\) bool where
rifp u v = Policy.dfp (partition u) (partition v)

Context Switches
definition rcswitch :: nat \(\Rightarrow\) rstate-t \(\Rightarrow\) rstate-t where
rcswitch n s = \(\uparrow\) (↓ s ((\(\downarrow s\)) (current := (SOME t . True)) )

4.8.3 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 \(\Rightarrow\) 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 \(\Rightarrow\) 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**

$$\text{is-SK-EV-SIGNAL} : \text{raction-t list} \Rightarrow \text{bool}$$

**Where**

$$\text{is-SK-EV-SIGNAL aseq} \equiv \exists \text{ partner} .$$

$$aseq = [\text{SK-EV-SIGNAL EV-SIGNAL-PREP partner},$$

$$\text{SK-EV-SIGNAL EV-SIGNAL-FINISH partner}]$$

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

**Definition**

$$\text{rAS-set} : \text{raction-t list set}$$

**Where**

$$\text{rAS-set} \equiv \{ aseq . \text{is-SK-IPC aseq} \lor \text{is-SK-EV-WAIT aseq} \lor \text{is-SK-EV-SIGNAL aseq} \} \cup \{\}$$

### 4.8.4 Control

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

**Abbreviation**

$$\text{raborting}$$

**Where**

$$\text{raborting s} \equiv \text{aborting}(\downarrow s)$$

**Abbreviation**

$$\text{rwaiting}$$

**Where**

$$\text{rwaiting s} \equiv \text{waiting}(\downarrow s)$$

**Definition**

$$\text{rset-error-code} : \text{rstate-t} \Rightarrow \text{raction-t} \Rightarrow \text{rstate-t}$$

**Where**

$$\text{rset-error-code s a} \equiv s$$

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**

$$\text{rkinvolved} : \text{int-point-t} \Rightarrow \text{rdom-t set}$$

**Where**

$$\text{rkinvolved a} \equiv$$

$$\text{case a of } \text{SK-IPC dir WAIT partner page} \Rightarrow \{ \text{partner} \}$$

$$\text{SK-EV-SIGNAL EV-SIGNAL-FINISH partner} \Rightarrow \{ \text{partner} \}$$

$$\text{-} \Rightarrow \{ \}$$

**Abbreviation**

$$\text{rinvolved}$$

**Where**

$$\text{rinvolved} \equiv \text{Kernel.involved rkinvolved}$$

### 4.8.5 Discharging the proof obligations

**Lemma**

$$\text{inst-vpeq-rel}$$

**Shows**

$$\text{rvpeq-refl}: \text{rvpeq u s s}$$

**And**

$$\text{rvpeq-sym}: \text{rvpeq u s1 s2} \implies \text{rvpeq u s2 s1}$$

**And**

$$\text{rvpeq-trans}: [[ \text{rvpeq u s1 s2}; \text{rvpeq u s2 s3} ]] \implies \text{rvpeq u s1 s3}$$

**Unfolding**

$$\text{rvpeq-def using vpeq-rel by metis} +$$

**Lemma**

$$\text{inst-ifp-refl}$$

**Shows**

$$\forall u . \text{rifp u u}$$

**Unfolding**

$$\text{rifp-def using Policy-properties.ifp-reflexive by fast}$$

**Lemma**

$$\text{inst-step-atomicity \ [simp]}$$

**Shows**

$$\forall s a . \text{rcurrent (rstep s a) = rcurrent s}$$

**Unfolding**

$$\text{rstep-def rcurrent-def using atomic-step-does-not-change-current-thread rstate-up-down rstate-invariant atomic-step-preserves-invariants by auto}$$

**Lemma**

$$\text{inst-weakly-step-consistent}$$

**Assumes**

$$\text{rvpeq u s t}$$
and \( \text{rvpeq } (\text{rcurrent } s) \ s \ t \)
and \( \text{rcurrent } s = \text{rcurrent } t \)
and \( \text{rprecondition } s (\text{rcurrent } s) \ a \)
and \( \text{rprecondition } t (\text{rcurrent } t) \ a \)
shows \( \text{rvpeq } u (\text{rstep } s a) (\text{rstep } t a) \)

using \( \text{assms atomic-step-weakly-step-consistent rstate-invariant atomic-step-preserves-invariants} \)

unfolding \( \text{rcurrent-def rstep-def rvpeq-def rprecondition-def} \)
by \text{auto}

**Lemma inst-local-respect:**

assumes \( \neg \text{rifp} (\text{rcurrent } s) \ u \)
and \( \text{rprecondition } s (\text{rcurrent } s) \ a \)
shows \( \text{rvpeq } u s t = \text{rvpeq } u s \)

using \( \text{assms atomic-step-respects-policy rstate-invariant atomic-step-preserves-invariants} \)

unfolding \( \text{rifp-def rprecondition-def rvpeq-def rstep-def rcurrent-def} \)
by \text{auto}

**Lemma inst-output-consistency:**

assumes \( \text{rvpeq } (\text{rcurrent } s) \ s t \)
and \( \text{current-eq } \text{rcurrent } s = \text{rcurrent } t \)
shows \( \text{routput-f } s a = \text{routput-f } t a \)

unfolding \( \text{cases Policy,sp-spec-subj-obj ?part (PAGE p) READ} \)
rule: \text{case-split [case-names Allowed Denied]}

\begin{cases}
\text{case Allowed}
\begin{align*}
\text{have 5}: \text{obj } (\downarrow s) (\text{PAGE } p) &= \text{obj } (\uparrow t) (\text{PAGE } p) \\
& \text{using 1 Allowed unfolding rvpeq-def vpeq-def vpeq-obj-def by auto} \\
\text{have 6}: \text{sp-impl-subj-obj } (\downarrow s) ?\text{part } (\text{PAGE } p) \text{ READ} = \text{sp-impl-subj-obj } (\uparrow t) ?\text{part } (\text{PAGE } p) \text{ READ} \\
& \text{using 1 2 Allowed unfolding rvpeq-def vpeq-def vpeq-subj-obj-def by auto} \\
& \text{show routput-f } s a p = \text{routput-f } t a p \\
& \text{unfolding routput-f-def using 2 5 6 by auto} \\
\text{next case Denied}
\begin{align*}
& \text{hence sp-impl-subj-obj } (\downarrow s) ?\text{part } (\text{PAGE } p) \text{ READ} = \text{False} \\
& \text{and sp-impl-subj-obj } (\uparrow t) ?\text{part } (\text{PAGE } p) \text{ READ} = \text{False} \\
& \text{using rstate-invariant unfolding atomic-step-invariant-def sp-subset-def by auto} \\
& \text{thus routput-f } s a p = \text{routput-f } t a p \\
& \text{using 2 unfolding routput-f-def by simp} \\
\text{qed}
\end{align*}
\end{cases}
\end{cases}
\)

\begin{cases}
\text{thus } s t \ \text{rvpeq } (\text{rcurrent } s) \ s t \land \text{rcurrent } s = \text{rcurrent } t \implies \text{routput-f } s a = \text{routput-f } t a \\
& \text{by auto} \\
\text{qed}
\end{cases}
\)

\begin{cases}
\text{thus } ?\text{thesis using assms by auto} \\
\end{cases}
\)
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
  from 1 2 assms show ?thesis unfolding rvpeq-def using vpeq-rel by metis
qed

For the PREP stage (the first stage of the IPC action sequence) the precondition is True.

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

For the first stage of the EV-WAIT action sequence the precondition is True.

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

For the first stage of the EV-SIGNAL action sequence the precondition is True.

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
ev-signal-precondition-def
by auto

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.

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 \( \text{rprecondition} \ (\text{rstep} \ s \ (\text{aseq} \ ! \ n)) \ (\text{rcurrent} \ s) \ (\text{aseq} \ ! \ Suc \ n) \)

\text{proof—}

\{ 
  \text{fix} \ dir \ \text{partner page} \\
  \text{let} \ ?\text{page}' = (\text{SOME} \ \text{page'}). \text{True} \\
  \text{assume IPC: aseq = [SK-IPC dir PREP partner page,SK-IPC dir WAIT partner page,SK-IPC dir (BUF ?page') partner page]}
\}

\text{assume} 0: n=0 \\
\text{from} 0 \text{ IPC prec not-aborting} \\
\text{have} \ ?\text{thesis} \\
\text{unfolding} \text{rprecondition-def atomic-step-precondition-def rstep-def rcurrent-def atomic-step-def atomic-step-ipc-def aborting-def} \\
\text{by(auto)}

moreover 

\{ 
  \text{assume} 1: n=1 \\
  \text{from} 1 \text{ IPC prec not-waiting} \\
  \text{have} \ ?\text{thesis} \\
  \text{unfolding} \text{rprecondition-def atomic-step-precondition-def rstep-def rcurrent-def atomic-step-def atomic-step-ipc-def waiting-def} \\
  \text{by(auto)}
\}

moreover 

from IPC 

\text{have} length aseq = 3 \\
\text{by} auto

ultimately 

\text{have} \ ?\text{thesis} \\
\text{using} n-bound \\
\text{by} arith

\}

\text{thus} \ ?\text{thesis} \\
\text{using IPC} \\
\text{unfolding} \text{is-SK-IPC-def} \\
\text{by(auto)}

\text{qed}

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

\text{lemma} \text{prec-after-EV-WAIT-step:}

\text{assumes} \text{prec: rprecondition s (rcurrent s) (aseq ! n)} \\
\text{and} n-bound: Suc n < length aseq \\
\text{and IPC: is-SK-EV-WAIT aseq} \\
\text{and not-aborting: \neg raborting s (rcurrent s) (aseq ! n)} \\
\text{and not-waiting: \neg rwaiting s (rcurrent s) (aseq ! n)}

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

\text{proof—}

\{ 
  \text{fix} \ consume \\
  \text{assume WAIT: aseq = [SK-EV-WAIT EV-PREP consume,} \\
  \text{SK-EV-WAIT EV-WAIT consume,} \\
  \text{SK-EV-WAIT EV-FINISH consume]}
\}

\text{assume} 0: n=0 \\
\text{from} 0 \text{ WAIT prec not-aborting} \\
\text{have} \ ?\text{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
unfolding is-SK-EV-WAIT-def
by auto
qed

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

lemma prec-after-EV-SIGNAL-step:
assumes prec: rprecondition s (rcurrent s) (aseq ! n)
  and n-bound: Suc n < length aseq
  and SIGNAL: is-SK-EV-SIGNAL 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−
  {  
    assume SIGNAL1: aseq = [SK-EV-SIGNAL EV-SIGNAL-PREP partner,
                             SK-EV-SIGNAL EV-SIGNAL-FINISH partner]
    
    assume 0: n=0
    from 0 SIGNAL1 prec not-aborting
    have ?thesis
unfolding rprecondition-def atomic-step-precondition-def ev-signal-precondition-def
  aborting-def rstep-def atomic-step-def
  by auto
}

moreover
from SIGNAL1
  have length aseq = 2
  by auto
ultimately
  have ?thesis
  using n-bound
  by arith
}

thus ?thesis
using assms
unfolding is-SK-EV-SIGNAL-def
by auto

qed

lemma on-set-object-value:
shows sp-impl-subj-subj (set-object-value ob val s) = sp-impl-subj-subj s
and sp-impl-subj-obj (set-object-value ob val s) = sp-impl-subj-obj s
unfolding set-object-value-def apply simp+ done

lemma prec-IPC-dom-independent:
assumes current s \not= d
and atomic-step-invariant s
and atomic-step-precondition s d a
shows atomic-step-precondition (atomic-step-ipc (current s) dir stage partner page s) d a
using assms on-set-object-value
unfolding atomic-step-precondition-def atomic-step-ipc-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-signal-dom-independent:
assumes current s \not= d
and atomic-step-invariant s
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 \not= 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 \not= 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 \( \forall \ s \ d \ a \ a'. \ rcurrent s \not= d \ \\& \ \ \ rprecondition s d a \ \rightarrow \ \ rprecondition (rstep s a') d a \)
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
lemma ipc-precondition-after-cswitch\[simp\]:
shows ipc-precondition s d dir partner page ((↓ s)([current := new-current]))
= ipc-precondition s d dir partner page (↓ s)
using assms
unfolding ipc-precondition-def
by (auto split add: ipc-direction-t.splits)

lemma precondition-after-cswitch:
shows ∀ s d n a. rprecondition s d a → 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 ∀ n s. raborting (rcswitch n s) = raborting s
proof−
{ fix n s
{ fix tid a
  have raborting (rcswitch n s) tid a = raborting s tid a
  using rstate-invariant cswitch-preserves-invariants ev-signal-precondition-weakly-step-consistent
cswitch-consistency-and-respect
  unfolding aborting-def rcswitch-def
  apply (auto split add: int-point-t.splits ipc-stage-t.splits
  ev-wait-stage-t.splits ev-signal-stage-t.splits)
  apply (metis (full-types))
  by blast
}
hence raborting (rcswitch n s) = raborting s by auto
}
thus ?thesis by auto
qed

lemma waiting-switch-independent:
shows ∀ n s. rwaiting (rcswitch n s) = rwaiting s
proof−
{ fix n s
{ fix tid a
  have rwaiting (rcswitch n s) tid a = rwaiting s tid a
  using rstate-invariant cswitch-preserves-invariants
  unfolding waiting-def rcswitch-def
  by (auto split add: int-point-t.splits ipc-stage-t.splits ev-wait-stage-t.splits)
}
hence rwaiting (rcswitch n s) = rwaiting s by auto
}
thus ?thesis by auto
qed

lemma aborting-after-IPC-step:
assumes d1 ≠ d2
shows aborting (atomic-step-ipc d1 dir stage partner page s) d2 a = aborting s d2 a
lemma waiting-after-IPC-step:
assumes d1 ≠ d2
shows waiting (atomic-step-ipc d1 dir stage partner page s) d2 a = waiting s d2 a

lemma raborting-consistent:
shows ∀ s t u. rvpeq u s t → raborting s u = raborting t u

proof -
{  
fix s t u
assume vpeq: rvpeq u s t  
{    
fix a
from vpeq ipc-precondition-weakly-step-consistent rstate-invariant
have ∃ tid dir partner page . ipc-precondition u dir partner page (↓s) = ipc-precondition u dir partner page (↓t)
unfolding rvpeq-def
by auto
with vpeq rstate-invariant have raborting s u a = raborting t u a
unfolding aborting-def rvpeq-def vpeq-def vpeq-local-def ev-signal-precondition-def
vpeq-subj-subj-def atomic-step-invariant-def sp-subset-def rep-def
apply (auto split add: int-point-t.splits ipc-stage-t.splits ev-signal-stage-t.splits)
by blast
}
hence raborting s u = raborting t u by auto
}
thus ?thesis by auto
qed

lemma aborting-dom-independent:
assumes rcurrent s ≠ d
shows raborting (rstep s a) d a' = raborting s d a'

proof -
have ∀ tid dir partner page s . ipc-precondition tid dir partner page s = ipc-precondition tid dir partner page (atomic-step s a)∧ ev-signal-precondition tid partner s = ev-signal-precondition tid partner (atomic-step s a)

proof -  
fix tid dir partner page s
let ?s = atomic-step s a
have (∀ p q . sp-impl-subj-subj s p q = sp-impl-subj-subj ?s p q)∧ (∀ p x m . sp-impl-subj-obj s p x m = 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 = ipc-precondition tid dir partner page (atomic-step s a)
\[ \text{ev-signal-precondition tid partner } s = \text{ev-signal-precondition tid partner } (\text{atomic-step } s a) \]

\text{unfolding ipc-precondition-def ev-signal-precondition-def by simp}

\text{qed}

\text{moreover have } \land b . (\langle (\langle\langle \text{atomic-step } (\downarrow s) b \rangle) \rangle) = \text{atomic-step } (\downarrow s) b

\text{using rstate-invariant atomic-step-preserves-invariants rstate-up-down by auto}

\text{ultimately show } \text{thesis}

\text{unfolding aborting-def rstep-def ev-signal-precondition-def}

\text{by (simp split add: int-point-t.splits ipc-stage-t.splits ev-wait-stage-t.splits ev-signal-stage-t.splits)}

\text{qed}

\text{lemma ipc-precondition-of-partner-consistent:}

\text{assumes vpeq : } \forall d \in \text{rkinsonvolved } (SK-IPC \text{ dir WAIT partner page} ) . \text{rvpeq } d s t

\text{shows ipc-precondition partner dir'} u page' (\downarrow s) = \text{ipc-precondition partner dir'} u page' (\downarrow t)

\text{proof--}

\text{from assms ipc-precondition-weakly-step-consistent rstate-invariant}

\text{show } \text{thesis}

\text{unfolding rvpeq-def rkinvolved-def}

\text{by auto}

\text{qed}

\text{lemma ev-signal-precondition-of-partner-consistent:}

\text{assumes vpeq : } \forall d \in \text{rkinsonvolved } (SK-EV-SIGNAL EV-SIGNAL-FINISH partner) . \text{rvpeq } d s t

\text{shows ev-signal-precondition partner u (\downarrow s) = ev-signal-precondition partner u (\downarrow t)}

\text{proof--}

\text{from assms ev-signal-precondition-weakly-step-consistent rstate-invariant}

\text{show } \text{thesis}

\text{unfolding rvpeq-def rkinvolved-def}

\text{by auto}

\text{qed}

\text{lemma waiting-consistent:}

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

\land \text{rvpeq } u s t

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

\text{proof--}

\{ 

\text{fix } s t u a

\text{assume vpeq: rvpeq } (\text{rcurrent } s) s t

\text{assume vpeq-involved: } \forall d \in \text{rkinsonvolved } a . \text{rvpeq } d s t

\text{assume vpeq-u: rvpeq } u s t

\text{have rwaiting } s u a = \text{rwaiting } t u a \text{ proof (cases a)}

\text{case } SK-IPC

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

\text{using ipc-precondition-of-partner-consistent vpeq-involved}

\text{unfolding waiting-def by (auto split add: ipc-stage-t.splits)}

\text{next case } SK-EV-WAIT

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

\text{using ev-signal-precondition-of-partner-consistent}

\text{vpeq-involved vpeq-v}

\text{unfolding waiting-def rkinvolved-def ev-signal-precondition-def}

\text{vpeq-def vpeq-local-def}

\text{by (auto split add: ipc-stage-t.splits ev-wait-stage-t.splits ev-consume-t.splits)}

\text{qed (simp add: waiting-def, simp add: waiting-def)}

\}

\text{thus } \text{thesis by auto}
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 = λ t1 t2 . Policy.sp-spec-subj-subj (partition t1) (partition t2)
  have ?sp (current s) partner ∨ ?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 = λ t1 t2 . Policy.sp-spec-subj-subj (partition t1) (partition t2)
  have ?sp (current s) partner ∨ ?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 . ∀ 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
4.9 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
(↑s0)
rcurrent
rcswitch
rkinvolved
rifp
rvpeq
rAS-set
rinvariant
rprecondition
raborting
rwaiting
rset-error-code
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 ∧ 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-weakly-step-consistent by blast
— show step atomicity
show ∀ s a . rcurrent (rstep s a) = rcurrent s
using inst-step-atomicity by metis
show ∀ a s t . ¬ rifp (rcurrent s) u ∧ True ∧ rprecondition s (rcurrent s) a ⟹ rvpeq u s (rstep s a)
using inst-local-respect by blast
— show csswitch 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 csswitch consistency
show \( \forall u s t n. \ \text{rvpeq } u s t \rightarrow \text{rvpeq } u (\text{rcswitch } n s) (\text{rcswitch } n t) \)
using \text{inst-cswitch-consistency} by \text{metis}
— Show the empt action sequence is in \text{AS-set}

show \([\ ] \in \text{rAS-set}\)
unfolding \text{rAS-set-def} by \text{auto}
— The invariant for the initial state, already encoded in \text{rstate-t}

show \(\text{True}\)
by \text{auto}
— Step function of the invariant, already encoded in \text{rstate-t}

show \(\forall s n. \text{True} \rightarrow \text{True}\)
by \text{auto}
— The precondition does not change with a context switch

show \(\forall s d n a. \ \text{rprecondition } s d a \rightarrow \text{rprecondition } (\text{rcswitch } n s) d a\)
using \text{precondition-after-cswitch} by \text{blast}
— The precondition holds for the first action of each action sequence

show \(\forall s d a\text{seq}. \text{True} \land a\text{seq} \in \text{rAS-set} \land a\text{seq} \notin [\ ] \rightarrow \text{rprecondition } s d (\text{hd aseq})\)
using \text{prec-first-IPC-action} \text{prec-first-EV-WAIT-action} \text{prec-first-EV-SIGNAL-action}
unfolding \text{rAS-set-def} \text{is-sub-seq-def} by \text{auto}
— The precondition holds for the next action in an action sequence, assuming the sequence is not aborted or delayed

show \(\forall s a a'. (\exists a\text{seq} \in \text{rAS-set}. \ lsub-seq a a' a\text{seq}) \land \text{True} \land \text{rprecondition } s (\text{rcurrent } s) a \land \neg \text{raborting } s (\text{rcurrent } s) a \land \neg \text{rwaiting } s (\text{rcurrent } s) a \rightarrow \text{rprecondition } (\text{rstep } s a') (\text{rcurrent } s) a'\)
using \text{prec-after-IPC-step} \text{prec-after-EV-SIGNAL-step} \text{prec-after-EV-WAIT-step}
unfolding \text{rAS-set-def} \text{is-sub-seq-def} by \text{auto}
— Steps of other domains do not influence the precondition

show \(\forall s d a a'. \text{rcurrent } s d a \land \text{rprecondition } s d a' \rightarrow \text{rprecondition } (\text{rstep } s a') d a\)
using \text{prec-dom-independent} by \text{blast}
— The invariant

show \(\forall s a. \text{True} \rightarrow \text{True}\)
by \text{auto}
— Aborting does not depend on a context switch

show \(\forall n s. \text{raborting } (\text{rcswitch } n s) = \text{raborting } s\)
using \text{aborting-switch-independent} by \text{auto}
— Aborting does not depend on actions of other domains

show \(\forall s d a. \text{rcurrent } s d a \land \text{rprecondition } s d a \rightarrow \text{rprecondition } (\text{rstep } s a) d a\)
using \text{aborting-dom-independent} by \text{auto}
— Aborting is consistent

show \(\forall s t u. \ \text{rvpeq } u s t \rightarrow \text{raborting } s u = \text{raborting } t u\)
using \text{raborting-consistent} by \text{auto}
— Waiting does not depend on a context switch

show \(\forall n s. \text{rwaiting } (\text{rcswitch } n s) = \text{rwaiting } s\)
using \text{waiting-switch-independent} by \text{auto}
— Waiting is consistent

show \(\forall s t u a. \ \text{rvpeq } (\text{rcurrent } s) s t \land (\forall d \in \text{rinvolved } a. \ \text{rvpeq } d s t) \land \text{rvpeq } u s t \rightarrow \text{rwaiting } s u a = \text{rwaiting } t u a\)
unfolding \text{Kernel.involved-def}
using \text{waiting-consistent} by \text{auto}
— Domains that are involved in an action may influence the domain of the action

show \(\forall s a. \forall d \in \text{rinvolved } a. \ \text{rprecondition } s (\text{rcurrent } s) a \rightarrow \text{rifp } d (\text{rcurrent } s)\)
using \text{involved-ifp} by \text{blast}
— An action that is waiting does not change the state
D31.1 – Formal Specification of a Generic Separation Kernel

Using spec-of-waiting by blast
— Proof obligations for set-error-code. Right now, they are all trivial

Show ∀ s d a a′. rcurrent s ↛ d ∧ raborting s d a → raborting (rset-error-code s a′) d a

Unfolding rset-error-code-def by auto

Show ∀ s t u a. rvpeq u s t → rvpeq u (rset-error-code s a) (rset-error-code t a)

Unfolding rset-error-code-def by auto

Show ∀ s u a. ¬ rifp (rcurrent s) u → rvpeq u s (rset-error-code s a)

By (metis ∀ a u. rvpeq u a a)

Show ∀ s a. rcurrent (rset-error-code s a) = rcurrent s

Unfolding rset-error-code-def by auto

Qed

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 give action
rifp — information flow policy
rvpeq — view partitioning
rAS-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
rwaiting — 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:
Inst.isecure
Using Inst.unwinding-implies-isecure-CISK
By blast

End

5 Related Work

We consider various definitions of intransitive (I) noninterference (NI). This overview is by no means intended to be complete. We first prune the field by focusing on INI with as granularity the domains: if the security policy states the act “v ~ u”, this means domain v is permitted to flow any information it has at its disposal to u. We do not consider language-based approaches to noninterference [26], which allow
finer granularity mechanisms (i.e., flowing just a subset of the available information). Secondly, several formal verification efforts have been conducted concerning properties similar and related to INI such as no-exfiltration and no-infiltration [9]. Heitmeyer et al. prove these properties for a separation kernel in a Common Criteria certification process [11] (which kernel and which EAL is not clear). Martin et al. proved separation properties over the MASK kernel [18] and Shapiro and Weber verified correctness of the EROS confinement mechanism [28]. Klein provides an excellent overview of OSs for which such properties have been verified [13]. Thirdly, INI definitions can be built upon either state-based automata, trace-based models, or process algebraic models [30]. We do not focus on the latter, as our approach is not based on process algebra.

Transitive NI was first introduced by Goguen and Meseguer in 1982 [7] and has been the topic of heavy research since. Goguen and Meseguer tried to extend their definition with an unless construct to allow such policies [8]. This construct, however, did not capture the notion of INI [17]. The first commonly accepted definition of INI is Rushby’s purging-based definition IP-secure [24]. IP-secure has been applied to, e.g., smartcards [27] and OS kernel extensions [7]. To the best of our knowledge, Rushby’s definition has not been applied in a certification context. Rushby’s definition has been subject to heavy scrutiny [22], [29] and a vast array of modifications have been proposed.

Roscoe and Goldsmith provide CSP-based definitions of NI for the transitive and the intransitive case, here dubbed as lazy and mixed independence. The latter one is more restrictive than Rushby’s IP-secure. Their critique on IP-secure, however, is not universally accepted [7]. Greve at al. provided the GWV framework developed in ACL2 [9]. Their definition is a non-inductive version of noninterference similar to Rushby’s step consistency. GWV has been used on various industrial systems. The exact relation between GWV and (I)P-secure, i.e., whether they are of equal strength, is still open. The second property, Declassification, means whether the definition allows assignments in the form of $l := \text{declassify}(h)$ (where we use Sabelfeld’s notation for high and low variables). Information flows from $h$ to $l$, but only after it has been declassified. In general, NI is coarser than declassification. It allows where downgrading can occur, but now what may be downgraded [17]. Mantel provides a definition of transitive NI where exceptions can be added to allow de-classification by adding intransitive exceptions to the security policy [17].

To deal with concurrency, definitions of NI have been proposed for Non-Deterministic automata. Von Oheimb defined noninfluence for such systems. His definition can be regarded as a “non-deterministic version” of IP-secure. Engelhardt et al. defined nTA-secure, the non-deterministic version of TA-security. Finally, some notions of INI consider models that are in a sense richer than similar counterparts. Leslie extends Rushby’s notion of IP-secure for a model in which the security policy is Dynamic. Eggert et al. defined i-secure, an extension of IP-secure. Their model extends Rushby’s model (Mealy machines) with Local security policies. Murray et al. extends Von Oheimb definition of noninfluence to apply to a model that does not assume a static mapping of actions to domains. This makes it applicable to OSs, as in such a setting such a mapping does not exist [20]. NI-OS has been applied to the seL4 separation kernel [20], [14].

Most definitions have an associated methodology. Various methodologies are based on unwinding [8]. This breaks down the proof of NI into smaller proof obligations (POs). These POs can be checked by some manual proof [24], [10], model checking [32] or dedicated algorithms [5]. The methodology of Murray et al. is a combination of unwinding, automated deduction and manual proofs. Some definitions are undecidable and have no suitable unwinding.

We are aiming to provide a methodology for INI based on a model that is richer in detail than Mealy machines. This places our contribution next to other works that aim to extend IP-secure [15], [4] in Figure 2. Similar to those approaches, we take IP-secure as a starting point. We add kernel control mechanisms, interrupts and context switches. Ideally, we would simply prove IP-secure over CISK. We argue that this is impossible and that a rephrasing is necessary.

Our ultimate goal — certification of PikeOS — is very similar to the work done on seL4 [20][19]. There are two reasons why their approach is not directly applicable to PikeOS. First, seL4 has been developed from scratch. A Haskell specification serves as the medium for the implementation as well
as the system model for the kernel [6]. C code is derived from a high level specification. PikeOS, in contrast, is an established industrial OS. Secondly, interrupts are mostly disabled in seL4. Klein et al. side-step dealing with the verification complexity of interrupts by using a mostly atomic API [14]. In contrast, we aim to fully address interrupts.

With respect to attempts to formal operating system verifications, notable works are also the Verisoft I project [1] where also a weak form of a separation property, namely fairness of execution was addressed [3].

6 Conclusion

We have introduced a generic theory of intransitive non-interference for separation kernels with control as a series of locales and extensible record definitions in order to achieve a modular organization. Moreover, we have shown that it can be instantiated for a simplistic API consisting of IPC and events.

In the ongoing EURO-MILS project, we will extend this generic theory in order make it sufficiently rich to be instantiated with a realistic functional model of PikeOS.

6.0.1 Acknowledgement.

This work corresponds to the formal deliverable D31.1 of the Euro-MILS project funded by the European Unions Programme

\[ FP7/2007 – 2013 \]

under grant agreement number ICT-318353.

References


