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>16th August, 2018</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>
<tr>
<td>Abstract:</td>
<td>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.</td>
</tr>
<tr>
<td>Keywords:</td>
<td>separation kernel with control, formal model, instantiation, IPC, events, Isabelle/HOL</td>
</tr>
</tbody>
</table>
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 50
  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 70
  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 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

```latex
theory Option-Binders
imports Main
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 \( B2 \): “\( m_1 \parallel m_2 \rightarrow \alpha \)” represents “First compute \( m_1 \) and \( m_2 \), if one of them is None then return True, otherwise pass the result to \( \alpha \)”.

**definition** \( B :: \mid a \text{ option} \rightarrow (a \Rightarrow \text{bool}) \Rightarrow \text{bool} \) (infixl \( \rightarrow \) \( 65 \))

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

**definition** \( B2 :: \mid a \text{ option} \rightarrow a \text{ option} \Rightarrow (a \Rightarrow a \Rightarrow \text{bool}) \Rightarrow \text{bool} \)

**where** \( B2 m1 m2 \alpha \equiv m1 \rightarrow (\lambda a \cdot m2 \rightarrow (\lambda b \cdot \alpha a b)) \)

**syntax** \( B2 :: [\mid a \text{ option}, a \text{ option}, (a \Rightarrow a \Rightarrow \text{bool})] \Rightarrow \text{bool} ((\cdot \parallel \cdot \rightarrow \cdot) [0, 0, 10] 10) \)

Some rewriting rules for the binders

**lemma** \( \text{rewrite-B2-to-cases}[\text{simp}]: \)

**shows** \( B2 s t f = (\text{case } s \text{ of None } \Rightarrow \text{True} | (\text{Some } s1) \Rightarrow (\text{case } t \text{ of None } \Rightarrow \text{True} | (\text{Some } t1) \Rightarrow f s1 t1)) \)

**unfolding** \( \text{B-def B-def by(\text{cases } s, cases \text{ } t, \text{simp}+)} \)

**lemma** \( \text{rewrite-B-None}[\text{simp}]: \)

**shows** \( \text{None } \rightarrow \alpha \Rightarrow \text{True} \)

**unfolding** \( \text{B-def by(auto)} \)

**lemma** \( \text{rewrite-B-m-True}[\text{simp}]: \)

**shows** \( m \rightarrow (\lambda a \cdot \text{True}) \Rightarrow \text{True} \)

**unfolding** \( \text{B-def by(\text{cases } m, \text{simp}+)} \)

**lemma** \( \text{rewrite-B2-cases}: \)

**shows** \( (\text{case } a \text{ of None } \Rightarrow \text{True} | (\text{Some } s) \Rightarrow (\text{case } b \text{ of None } \Rightarrow \text{True} | (\text{Some } t) \Rightarrow f s t)) \)

\[ = (\forall s t . a = (\text{Some } s) \land b = (\text{Some } t) \Rightarrow f s t) \]

**by(\text{cases } a, \text{simp, cases } b, \text{simp}+) \)

**definition** \( \text{strict-equal :: } \mid a \text{ option} \rightarrow a \Rightarrow \text{bool} \)

**where** \( \text{strict-equal } m a \equiv \text{case } m \text{ of None } \Rightarrow \text{False} | (\text{Some } a') \Rightarrow a' = a \)

**end**

### 2.2 Theorems on lists

**theory** List-Theorems

**imports** Main

**begin**

**definition** \( \text{lastn :: } \text{nat} \Rightarrow a \text{ list } \Rightarrow a \text{ list} \)

**where** \( \text{lastn } n x = \text{drop } ((\text{length } x) - n) x \)

**definition** \( \text{is-sub-seq :: } \mid a \Rightarrow a \Rightarrow a \text{ list } \Rightarrow \text{bool} \)

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

**definition** \( \text{prefixes :: } a \text{ list set } \Rightarrow a \text{ list set} \)

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

**lemma** \( \text{drop-one}[\text{simp}]: \)

**shows** \( \text{drop } (\text{Suc } 0) x = tl x \) \( \text{by (induct } x, \text{auto)} \)

**lemma** \( \text{length-ge-one}: \)

**shows** \( x \neq [] \Rightarrow \text{length } x \geq 1 \) \( \text{by (induct } x, \text{auto)} \)

**lemma** \( \text{take-but-one}[\text{simp}]: \)

**shows** \( x \neq [] \Rightarrow \text{lastn } ((\text{length } x) - 1) x = tl x \) \( \text{unfolding lastn-def} \)

**using** \( \text{length-ge-one}[\text{where } x=x] \) \( \text{by auto} \)

**lemma** \( \text{Suc-m-minus-n}[\text{simp}]: \)

**shows** \( m \geq n \Rightarrow \text{Suc } m - n = \text{Suc } (m - n) \) \( \text{by auto} \)
**lemma** lastn-one-less:
shows \( n > 0 \land n \leq \text{length } x \land \text{lastn } n \ x = (a\#y) \rightarrow \text{lastn } (n-1) \ x = y \)** unfolding lastn-def

using \( \text{drop-Suc[where } n=\text{length } x - n \text{ and } xs=x] \) drop-tl[where \( n=\text{length } x - n \text{ and } xs=x] \)

by(auto)

**lemma** list-sub-implies-member:
shows \( \forall a \ x . \text{set } (a\#x) \subseteq Z \rightarrow a \in Z \)

by(auto)

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

by(auto)

**lemma** second-elt-is-hd-tl:
shows \( \text{tl } x = (a\#x') \rightarrow a = x!1 \)

by(cases x,auto)

**lemma** length-ge-2-implies-tl-not-empty:
shows \( \text{length } x \geq 2 \rightarrow \text{tl } x \neq \[] \)

by(cases x,auto)

**lemma** length-lt-2-implies-tl-empty:
shows \( \text{length } x < 2 \rightarrow \text{tl } x = \[] \)

by(cases x,auto)

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

proof

assume \( \text{length } x \geq 2 \)

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

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

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

qed

**lemma** hd-drop-is-nth:
shows \( n < \text{length } x \rightarrow \text{hd } (\text{drop } n \ x) = x!n \)

proof(induct x arbitrary: \( n \))

case Nil

thus ?thesis by simp

next
case (Cons \( a \ x \))

have \( \text{hd } (\text{drop } n \ (a\#x)) = (a\#x)!n \)

proof(cases \( n \))

case 0

thus ?thesis by simp

next
case (Suc \( m \))

from Suc Cons show ?thesis by auto

qed

thus ?case by auto

qed

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

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

**lemma** drop-yields-results-implies-nbound:
shows \( \text{drop } n \ x \neq \[] \rightarrow n < \text{length } x \)

by(induct x,auto)

**lemma** consecutive-is-sub-seq:
shows \( a\#(b\#x) = \text{lastn } n \ y \rightarrow \text{is-sub-seq } a \ b \ y \)

proof

assume 1: \( a\#(b\#x) = \text{lastn } n \ y \)

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

lemma sub-seq-in-prefixes:
assumes 3 y \in \text{prefixes } X, \text{is-sub-seq } a a' y
shows 3 y \in X, \text{is-sub-seq } a a' y
proof-
from assms obtain y where y: y \in \text{prefixes } X \land \text{is-sub-seq } a a' y by auto
then obtain n x where x: n > 0 \land x \in X \land \text{take } n x = y
\text{unfolding prefixes-def by auto}
from y obtain i where \text{sub-seq-index } \text{Suc } i < \text{length } y \land y ! i = a \land y ! \text{Suc } i = a'
\text{unfolding is-sub-seq-def by auto}
from \text{sub-seq-index } x have \text{is-sub-seq } a a' x
\text{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\} \subseteq set x by(induct x,auto)
lemma x-is-hd-snd-tl:
shows length x \geq 2 \longrightarrow x = (\text{hd } x) \# x!1 \# tl(tl x)
proof(induct x)
case Nil
\text{show } \?case by auto
case (Cons a x)
\text{show } \?case by(induct xx,auto)
qed

lemma tl-x-not-x:
shows x \# [] \longrightarrow tl x \# x by(induct x,auto)
lemma tl-hd-x-not-tl-x:
shows x \# [] \land \text{hd } x \# [] \longrightarrow tl (\text{hd } x) \# tl x \# x \text{ using tl-x-not-x by(induct x,simp,auto)}
end

3 A generic model for separation kernels

theory K
imports List-Theorems Option-Binders
begin

This section defines a detailed generic model of separation kernels called CISK (Controlled Inter-
ruptible 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 \( \text{run} \), 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 sensical 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)

The model makes use of the following types:

’real_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 kernel. One call from a domain to the kernel will typically trigger a succession of one or more kernel actions. Therefore, an execution is represented as a list of sequences of kernel actions. Non-kernel actions are not taken into account.

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

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

Short-hand notations for using function control.

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

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

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

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

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

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

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

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

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

begin
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 $(\text{step } s' a)$. The current execution of the current domain is updated.

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

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

### 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 $ia$. Function $vpeq$ is adopted from Rushby and is an equivalence relation representing whether two states are equivalent from the point of view of the given domain.

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

- **Step Atomicity** Each atomic kernel step can be executed within one time slot. Therefore, the domain that is currently active does not change by executing one action.

- **Time-based Interrupts** As interrupts occur according to a prefixed time-based schedule, the domain that is active after a call of switch depends on the currently active domain only (cswitch consistency). Also, cswitch can only change which domain is currently active (cswitch consistency).

- **Control Consistency** States that are equivalent yield the same control. That is, the next action and the updated execution depend on the currently active domain only (next_action_consistent, next_execs_consistent), the state as updated by the control function remains in vpeq (next_state_consistent, locally_respects_next_state). Finally, function control cannot change which domain is active (current_next_state).

```plaintext
definition actions-in-execution: 'action-t execution ⇒ 'action-t set where actions-in-execution exec ≡ \{ a . \exists aseq ∈ set exec . a ∈ set aseq \}
```

locale Separation-Kernel = Kernel kstep output-f s0 current cswitch interrupt kprecondition realistic-execution control kinvolved
for kstep :: 'state-t ⇒ 'action-t ⇒ 'state-t
and output-f :: 'state-t ⇒ 'action-t ⇒ 'output-t
and s0 :: 'state-t
and current :: 'state-t ⇒ 'dom-t — Returns the currently active domain
and cswitch :: 'time-t ⇒ 'state-t ⇒ 'state-t — Switches the current domain
and interrupt :: 'time-t ⇒ bool — Returns t iff 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
+
+ fixes ifp :: 'dom-t ⇒ 'dom-t ⇒ bool
and vpeq :: 'dom-t ⇒ 'state-t ⇒ 'state-t ⇒ bool
assumes vpeq-transitive: ∀ a b c u. (vpeq u a b ∧ vpeq u b c) → vpeq u a c
and vpeq-symmetric: ∀ a b u. vpeq u a b → vpeq u b a
and vpeq-reflexive: ∀ a u. vpeq u a a
and ifp-reflexive: ∀ u. ifp u u
and weakly-step-consistent: ∀ s t u a. vpeq u s t ∧ vpeq (current s) s t ∧ kprecondition s a ∧ kprecondition t a ∧ current s = current t → vpeq u (kstep s a) (kstep t a)
and locally-respects: ∀ a s t. ¬ifp (current s) u ∧ kprecondition s a → vpeq u s (kstep s a)
and output-consistent: ∀ a s t. vpeq (current s) s t ∧ current s = current t → (output-f s a) = (output-f t a)
and step-atomicity: ∀ s a. current (kstep s a) = current s
and cswitch-independent-of-state: ∀ n s t. current s = current t → current (cswitch n s) = current (cswitch n t)
and cswitch-consistency: ∀ u s t n . vpeq u s t → vpeq u (cswitch n s) (cswitch n t)
and next-action-consistent: ∀ s t u execs. vpeq (current s) s t ∧ (∀ d ∈ involved (next-action s execs) . vpeq d s t) ∧ current s = current t → next-action s execs = next-action t execs
and next-actions-consistent: ∀ s t u execs . vpeq (current s) s t ∧ (∀ d ∈ involved (next-action s execs) . vpeq d s t) ∧ current s = current t → fst (snd (control s (current s) (execs (current s)))) = fst (snd (control t (current s) (execs (current s))))
and next-state-consistent: ∀ s t u execs . vpeq (current s) s t ∧ vpeq u s t ∧ current s = current t → vpeq u (next-state s execs) (next-state t execs)
and current-next-state: ∀ s execs . current (next-state s execs) = current s
and locally-respects-next-state: ∀ s u execs . ¬ifp (current s) u → vpeq u s (next-state s execs)
and involved-ifp: ∀ a s . ∀ d ∈ (involved a) . kprecondition s (the a) → ifp d (current s)
and next-action-from-exec: ∀ s execs . next-action s execs → (λ a . a ∈ actions-in-execution (execs (current s)))
and next-actions-subset: ∀ s execs u . actions-in-execution (next-actions execs u) ⊆ actions-in-execution (execs u)
begin

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

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 \( u \).

abbreviation unrelated :: 'dom-t ⇒ 'dom-t ⇒ bool
where unrelated d u ≡ ¬ifp d u
To formulate the new theorem to prove, we redefine purging: all domains that may not influence domain \( u \) are replaced by arbitrary action sequences.

**definition** `purge`: 

\[
\text{('dom-t \Rightarrow 'action-t execution) \Rightarrow 'dom-t \Rightarrow ('dom-t \Rightarrow 'action-t execution)}
\]

**where** `purge execs u \equiv \lambda d . (\text{if unrelated } d \ u \text{ then}
\text{\text{(SOME alpha . realistic-execution alpha)}}
\text{\text{else execs } d})`

A normal run from initial state \( s_0 \) ending in state \( s_f \) is equivalent to a run purged for domain \((\text{currents}_f)\).

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

\[
\equiv \forall \ \text{execs } a \ n . \ \text{run } n (\text{Some } s_0) \ \text{execs} \rightarrow \hspace{1cm} \hspace{1cm}
(\lambda s-f \ . \ \text{run } n (\text{Some } s_0) (\text{purge execs (current } s-f)) \rightarrow \hspace{1cm} \hspace{1cm}
(\lambda s-f2 . \ \text{output-f } s-f a = \text{output-f } s-f2 a \wedge \text{current } s-f = \text{current } s-f2))
\]

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 :: 'state-t option \Rightarrow 'state-t option \Rightarrow 'dom-t \Rightarrow bool`

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

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

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

\[
\equiv \forall \ \text{execs } m s m t n u . \ \text{equivalent-states } m s m t u \rightarrow \hspace{1cm} \hspace{1cm}
(\text{run } n \ m s \ m t) \rightarrow \hspace{1cm} \hspace{1cm}
\text{run } n m t (\text{purge execs } u) \rightarrow \hspace{1cm} \hspace{1cm}
(\lambda r s \ t . \ \text{vpeq } u r s t \wedge \text{current } r s = \text{current } r t))
\]

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 :: 'dom-t \Rightarrow ('dom-t \Rightarrow 'action-t execution) \Rightarrow ('dom-t \Rightarrow 'action-t execution) \Rightarrow bool`

**where** `purged-relation u execs1 execs2 u \equiv \forall d . \text{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`

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

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** `pre-c-t` : `precondition (next-state t execs2) (next-action t execs2)`

**assumes** `not-ifp-curr-u` : `\text{ifp}^{**} (current ) u`

**assumes** `vpeq-s-t` : `\forall v . \text{ifp}^{**} v u \rightarrow \text{vpeq } v s t`

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

**proof** :

\[
\begin{cases}
\text{fix } v
\end{cases}
\]
assume \textit{ifp-v-uc} \textit{ifp}^{**} v u

from \textit{ifp-v-u} \textit{not-ifp-curr-u} have unrelated: \textit{ifp}^{**} \textit{(current t)} v using rtranclp-trans by metis
from this current-next-state[THEN spec,THEN spec,\textit{where} x1=t] locally-respects[THEN spec,THEN spec,THEN spec,\textit{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

thus \textit{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 \textit{can} interact with \(u\) (the domain for which is purged).

\textbf{lemma vpeq-ns-nt-ifp-w:}
\begin{itemize}
\item \textit{assumes} vpeq-s-t: \(\forall v. \textit{ifp}^{**} v u \rightarrow vpeq v s t\)
\item \textit{and} current-s-t: \textit{current s = current t'}
\item \textit{shows} precondition (next-state s execs) \(a \land\) precondition (next-state t' execs) \(a \rightarrow (\textit{ifp}^{**} (current s) u \rightarrow (\forall v. \textit{ifp}^{**} v u \rightarrow vpeq v (step (next-state s execs) a) (step (next-state t' execs) a)))\)
\end{itemize}
\begin{itemize}
\item \textit{proof--}
\item \textit{fix a}
\item \textit{assume} precs: precondition (next-state s execs) \(a \land\) precondition (next-state t' execs) \(a\)
\item \textit{assume} ifp-curr: \textit{ifp}^{**} (current s) \(u\)
\item \textit{from} vpeq-s-t \textit{have} vpeq-curr-s-t: \textit{ifp}^{**} (current s) \(u \rightarrow vpeq (current s) s t'\) by auto
\item \textit{from} ifp-curr precs
\end{itemize}
next-state-consistent [THEN spec, THEN spec, where x1 = 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 = the a]
show \( \forall \, v \cdot \neg ifp^{++} v u \implies vpeq v \) (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: \( \forall \, v \cdot \neg ifp^{++} v u \implies vpeq v s \implies vpeq v t' \)
shows \( \neg ifp^{++} (\text{current } s) u \land \text{precondition} (\text{next-state } t' \text{ execs2}) (\text{next-action } t' \text{ execs2}) \implies (\forall \, v \cdot \neg ifp^{++} v u \implies vpeq v \) (step (next-state s execs) (next-action s execs)) (step (next-state t' execs2) (next-action t' execs2))

**Proof--**

{ assume not-ifp: \( \neg ifp^{++} (\text{current } s) u \)
assume prec-t: precondition (next-state t' execs2) (next-action t' execs2)
fix a a' v
assume ifp-v-u: \( \neg ifp^{++} v u \)
from not-ifp and purged-a-a2 have \( \neg ifp^{++} (\text{current } s) u \) unfolding purged-relation-def by auto
from this and ifp-v-u have not-ifp-curr-v: \( \neg ifp^{++} (\text{current } s) v \) using rtranclp-trans by metis
from this current-next-state[THEN spec, THEN spec, where x1 = s and x = execs2] prec-s vpeq-reflexive
locally-respects[THEN spec, THEN spec, THEN spec, where x1 = next-state s execs and x2 = the (next-action s execs) 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-ns: vpeq v s (step (next-state s execs) (next-action s execs)) by blast
from not-ifp-curr-v current-s-t current-next-state[THEN spec, THEN spec, where x1 = t' and x = execs2] prec-t locally-respects[THEN spec, THEN spec, where x = next-state t' execs2] vpeq-reflexive have 0: 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 not-ifp-curr-v current-s-t current-next-state have 1: \( \neg ifp^{++} (\text{current } t') v \) using rtranclp-trans by auto
from 0 1 locally-respects-next-state vpeq-transitive have vpeq-t-nt: vpeq v t' (step (next-state t' execs2) (next-action t' execs2)) by blast
from vpeq-s-ns and vpeq-t-nt and vpeq-s-t and ifp-v-u and vpeq-symmetric and vpeq-transitive have vpeq-s-ns: vpeq v (step (next-state s execs) (next-action s execs)) (step (next-state t' execs2) (next-action t' execs2)) by blast
}
thus ?thesis by auto
qed

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

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

shows view-partitioned-ind
D31.1 – Formal Specification of a Generic Separation Kernel

proof

{ fix execs execs2 s t n u 
  have equivalent-states s t u \land purged-relation u execs execs2 \longrightarrow equivalent-states (run n s execs) (run n t execs2) u 
  proof
  \((\text{induct } n s \text{ execs arbitrary: } t u \text{ execs2 rule: run.induct})\)
  case \((1 s execs t u execs2)\)
  \(\text{show } ?\text{case by auto}\)
  next
  case \((2 n execs t u execs2)\)
  \(\text{show } ?\text{case by simp}\)
  next
  case \((3 n s execs t u execs2)\)
  assume interrupt-s : \(\text{interrupt } (Suc n)\)
  assume IH : \(\{\text{purged-relation } u \text{ execs execs2} \longrightarrow\) equivalent-states \((\text{run } n \text{ Some (cswitch } (Suc n) s)) \text{ execs} \text{ execs2}\) \((\text{run } n t \text{ execs2}) u\}\)
  { fix \(t'\)
    assume \(t = \text{Some } t'\)
    fix rs
    assume rs : \(\text{run } (Suc n) (\text{Some } s) \text{ execs} = \text{Some } rs\)
    fix rt
    assume rt : \(\text{run } (Suc n) (\text{Some } t') \text{ execs2} = \text{Some } rt\)
    assume vpeq-s-t : \(\forall v. ifp^{**} v u \longrightarrow vpeq v s t'\)
    assume current-s-t : \(\text{current } s = \text{current } t'\)
    assume purged-a-a2 : \(\text{purged-relation } u \text{ execs execs2}\)

    — 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-ns-rt).

    from current-s-t cswitch-independent-of-state
    have current-ns-nt : \(\text{current } (cswitch (Suc n) s) = \text{current } (cswitch (Suc n) t')\) by blast
    from cswitch-consistency vpeq-s-t
    have vpeq-ns-nt : \(\forall v. ifp^{**} v u \longrightarrow vpeq v (cswitch (Suc n) s) (cswitch (Suc n) t')\) by auto
    from current-ns-nt vpeq-ns-nt interrupt-s vpeq-reflexive purged-a-a2 current-s-t IH[where u=u and t=Some (cswitch (Suc n) t') and ?execs2.0=execs2]
    have current-rs-rt : \(\text{current } rs = \text{current } rt\) 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 : \(\neg\text{interrupt } (Suc n)\)
  assume thread-empty-s : \(\text{thread-empty}(\text{execs } (\text{current } s))\)
The following terminology is used: states \( \text{rs} \) and \( \text{rt} \) (for: \( \text{run-s} \) and \( \text{run-t} \)) are the states after one step. States \( \text{ns} \) and \( \text{nt} \) have the same active domain. Statement \( \text{current-ns-nt} \) states that after one step \( \text{ns} \) and \( \text{nt} \) have the same active domain. Statement \( \text{current-s-t} \) states that after one step states \( \text{s} \) and \( \text{t} \) have the same active domain. Statement \( \text{vpeq-ns-nt} \) states that after one step states \( \text{ns} \) and \( \text{nt} \) are vpeq for all domains \( \text{v} \) that may influence \( \text{u} \) (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 \( \text{s} \) as the thread is empty). Statement \( \text{current-s-t} \) states that after one step states \( \text{ns} \) and \( \text{nt} \) are vpeq for all domains \( \text{v} \) that may influence \( \text{u} \) (vpeq-rs-rt).

---

### Proof

**Local respect first part.**

- From **thread-empty-s** and **purged-a-a2** and **current-s-t** have **purged-a-na2**: \( \neg \text{ifp}^* \) (current \( t' \)) \( u \) \( \rightarrow \) purged-relation \( u \) \( \text{execs} \) (next-execs \( t' \) execs2) by (unfold next-execs-def, unfold purged-relation-def, auto)

- From **step-atomicity**, current-next-state current-s-t have current-s-t: current \( s \) = current \( t' \) unfolding next-def

- From **step-atomicity**, current-next-state current-s-t have current-s-t: current \( s \) = current \( t' \) unfolding next-def

---

### Case distinction

- 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.

- From **t-not-empty current-next-state and vpeq-s-t-u and thread-empty-s and purged-a-a2 and current-s-t** IH[where \( t=\text{Some} \ t' \) and \( u=u \) and \( ?\text{execs2.0=}\text{execs2} \)] have equivalent-states (run \( n \) (Some \( s \) execs)) (run \( n \) (Some \( t' \) execs2)) using rs rt by (auto)

---

### Case t-prec

- From locally-respects-next-state current-next-state t-prec not-ifp-curr-t vpeq-s-t locally-respects vpeq-s-nt have vpeq-s-nt: \( \forall \ \text{v} . \ \text{ifp}^* \ \text{v} \ \text{u} \ \rightarrow \ \text{vpeq} \ \text{v} \ \text{s} \) (step (next-state \( t' \) execs2) (next-action \( t' \) execs2)) by auto

---

### Case t-not-prec

- From **t-not-empty current-next-state and vpeq-s-t-u and thread-empty-s and purged-a-a2 and current-s-t** have not-ifp-curr-t: \( \neg \text{ifp}^* \) (current (next-state \( t' \) execs2)) \( u \) unfolding purged-relation-def by auto show ?thesis proof (cases precondition (next-state \( t' \) execs2) (next-action \( t' \) execs2) rule : case-split [case-names t-prec t-not-prec])

---

### Case t-empty

- From **t-empty** have purged-a-a2 and vpeq-s-t and current-s-t IH[where \( t=\text{Some} \ t' \) and \( u=u \) and \( \text{execs2.0=}\text{execs2} \)] have equivalent-states (run \( n \) (Some \( s \) execs)) (run \( n \) (Some \( t' \) execs2)) using rs rt by (auto)
\begin{verbatim}
execs2) \) (next-execs t' execs2) u 
  using rs rt by auto 
  from t-not-empty t-prec vpeq-s-nt this thread-empty-s not-interrupt 
  show thesis using rs rt by auto
next 
case t-not-prec 
  thus ?thesis using rt t-not-empty not-interrupt by(auto)
qed 
qed 
\end{verbatim}
next

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

\textbf{assume} \text{not-interrupt} \rightarrow \text{interrupt} \ (\text{Suc} \ n)

\textbf{assume} \ \text{thread-not-empty-s} \rightarrow \text{thread-empty} (\text{execs} (\text{current} \ s))

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

\textbf{assume} \ \text{IH} : \ \forall \ u \ \text{execs2}.

\begin{align*}
& \text{equivalent-states} \ (\text{Some} \ (\text{step} \ (\text{next-state} \ s \ \text{execs}) \ (\text{next-action} \ s \ \text{execs}))) \ t \ u \wedge \\
& \text{purged-relation} \ u \ (\text{next-exec} \ s \ \text{execs} \ \text{execs2}) \rightarrow \\
& \text{equivalent-states} \\
& (\text{run} \ n \ (\text{Some} \ (\text{step} \ (\text{next-state} \ s \ \text{execs}) \ (\text{next-action} \ s \ \text{execs}))) \ (\text{next-exec} \ s \ \text{execs} \ \text{execs2})) \\
& (\text{run} \ n \ t \ \text{execs2}) \ u
\end{align*}

\[
\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{execs2} = \text{Some} \ rt
\end{align*}

\textbf{assume} \ \text{vpeq-s-t} : \ \forall \ v . \ \text{ifp}^* u \ u \rightarrow \ \text{vpeq} \ v \ s \ t' \\

\textbf{assume} \ \text{current-s-t} : \ \text{current} \ s = \text{current} \ t' \\

\textbf{assume} \ \text{purged-a-a2} : \ \text{purged-relation} \ u \ \text{execs} \ \text{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, state \( s \) executes an action). Statement current-ns-nt states that after one step states \( ns \) and \( nt \) have the same active domain. Statement vpeq-ns-nt states that after one step states \( ns \) and \( nt \) are vpeq for all domains \( v \) that may influence \( u \) (vpeq-rs-rt).

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

\textbf{from} \ \text{ifp-reflexive} \ \text{and} \ \text{vpeq-s-t} \ \text{have} \ \text{vpeq-s-t-u} \ \text{vpeq} \ u \ s \ t' \ \text{by} \ \text{auto}

\textbf{from} \ \text{step-atomicity} \ \text{and} \ \text{current-s-t} \ \text{current-next-state}

\text{have} \ \text{current-ns-nt} : \ \text{current} \ (\text{step} \ (\text{next-state} \ s \ \text{execs}) \ (\text{next-action} \ s \ \text{execs})) = \text{current} \ (\text{step} \ (\text{next-state} \ t' \ \text{execs2}) \ (\text{next-action} \ t' \ \text{execs2}))

\text{unfolding} \ \text{step-def}

\text{by} \ (\text{cases next-action} \ s \ \text{execs}, \text{cases next-action} \ t' \ \text{execs2}, \text{simp}, \text{simp}, \text{cases next-action} \ t' \ \text{execs2}, \text{simp}, \text{simp})

\textbf{from} \ \text{vpeq-s-t} \ \text{have} \ \text{vpeq-curr-s-t} : \ \text{ifp}^* (\text{current} \ s) \ u \rightarrow \ \text{vpeq} \ (\text{current} \ s) \ s \ t' \ \text{by} \ \text{auto}

\textbf{from} \ \text{prec-s} \ \text{have} \ \text{vpeq-curr-s-t} \ : \ \text{ifp}^* (\text{current} \ s) \ u \rightarrow \ \text{vpeq} \ (\text{current} \ s) \ s \ t' \ \text{by} \ \text{auto}

\textbf{from} \ \text{prec-s} \ \text{have} \ \text{vpeq-involved} : \ \text{ifp}^* (\text{current} \ s) \ u \rightarrow (\forall \ v \ \text{involved} \ (\text{next-action} \ s \ \text{execs}) . \ \text{vpeq} \ v \ s \ t')

\textbf{using} \ \text{current-next-state}

\textbf{unfolding} \ \text{involved-def} \ \text{precondition-def} \ B-def

\textbf{by} (\text{cases next-action} \ s \ \text{execs}, \text{simp}, \text{auto}, \text{metis converse-rtranclp-into-rtranclp})

\textbf{from} \ \text{current-s-t} \ \text{next-exec-s-consistent} \ \text{vpeq-curr-s-t} \ \text{vpeq-involved}

\textbf{have} \ \text{next-exec-s-t} : \ \text{ifp}^* (\text{current} \ s) \ u \rightarrow \ \text{next-exec} \ t' \ \text{execs} = \text{next-exec} \ s \ \text{execs}

\textbf{unfolding} \ \text{next-exec-def}

\textbf{by} (\text{auto})

\textbf{from} \ \text{current-s-t} \ \text{purged-a-a2} \ \text{thread-not-empty-s} \ \text{next-action-consistent}[\text{THEN} \ \text{spec}, \text{THEN} \ \text{spec.where} \ x1=s \ \text{and} \ \text{x}=t'] \ \text{vpeq-curr-s-t} \ \text{vpeq-involved}

\textbf{have} \ \text{next-action-s-t} : \ \text{ifp}^* (\text{current} \ s) \ u \rightarrow \ \text{next-action} \ t' \ \text{execs2} = \text{next-action} \ s \ \text{execs}

\textbf{by} (\text{unfold next-exec-def, unfold purged-relation-def, auto})

\textbf{from} \ \text{purged-a-a2} \ \text{current-s-t} \ \text{next-exec-consistent}[\text{THEN} \ \text{spec}, \text{THEN} \ \text{spec}, \text{THEN} \ \text{spec.where} \ x2=s \ \text{and} \ \text{x}=\text{execs}]

\text{vpeq-curr-s-t} \ \text{vpeq-involved}

\textbf{have} \ \text{purged-na-na2} : \ \text{purged-relation} \ u \ (\text{next-exec} \ s \ \text{execs}) \ (\text{next-exec} \ t' \ \text{execs2})

\textbf{unfolding} \ \text{next-exec-def} \ \text{purged-relation-def}
— 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.

**proof**

- **cases ifp** (current s) u rule :case-split[case-names curr-ifp-u curr-not-ifp-u]
  - **case curr-ifp-u**
    - **show ?thesis**
      - **proof**
        - **cases precondition (next-state t' execs2) (next-action t' execs2) rule :case-split[case-names prec-t prec-not-t]**
          - **case prec-t**
            - **have thread-not-empty-t: ~thread-empty(execs2 (current t')) using thread-not-empty-t curr-ifp-u by auto**
            - **from**
              - current-ns-nt next-execs-t next-action-s-t purged-a-a2
              - curr-ifp-u prec-t 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**
              - \[\text{IH}[\text{where } u = u \land \text{execs2.0} = (\text{next-execs } t' \text{ execs2}) \land t = \text{Some (step (next-state t' execs2) (next-action t' execs2))}]\]
              - current-ns-nt purged-na-na2
              - have equivalent-states (run n (Some (step (next-state s execs) (next-action 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 prec-t thread-not-empty-t prec-s and this and not-interrupt and thread-not-empty-s and next-action-s-t**
                - **show ?thesis using rs rt by auto**
          - **next**
            - **case prec-not-t**
              - **from curr-ifp-u prec-not-t thread-not-empty-t not-interrupt show ?thesis using rt by simp**
              - **qed**
            - **next**
              - **case curr-not-ifp-u**
                - **show ?thesis**
                  - **proof**
                    - **cases thread-empty(execs2 (current t')) rule :case-split[case-names t-empty t-not-empty]**
                    - **case t-not-empty**
                      - **show ?thesis**
                        - **proof**
                          - **cases precondition (next-state t' execs2) (next-action t' execs2) rule :case-split[case-names t-prec t-not-prec]**
                            - **case t-prec**
                              - **from curr-not-ifp-u t-prec IH[where u = u \land \text{execs2.0} = (\text{next-execs } t' \text{ execs2}) \land 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))

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

\( u \) by auto

from this \( t \)-prec curr-not-ifp-u \( t \)-not-empty prec-s not-interrupt thread-not-empty-s show ?thesis using \( rs rt \) by auto

next case \( t \)-not-prec

from \( t \)-not-prec \( t \)-not-empty not-interrupt show ?thesis using \( rt \) by simp

qed

next case \( t \)-empty

from curr-not-ifp-u and prec-s and vpeq-s-t and locally-respects and vpeq-ns-t current-next-state locally-respects-next-state

have vpeq-ns-t (\( \forall \, \, v \cdot \, \text{ifp}^\ast \ast \, v \, u \rightarrow \, \text{vpeq} \, v \, \text{step} \, \text{(next-state s execs)} \, (\text{next-action s execs}) \, t' \))

by blast

from curr-not-ifp-u IH[where \( t=\text{Some} \, t' \, \text{and} \, u=u \, \text{and} \, ?\text{execs2.0}=?\text{execs2} \, \text{and} \, \text{current-ns-t and} \, \text{next-execs-t and} \, \text{purged-na-a and} \, \text{vpeq-ns-t and} \, \text{this}]

have equivalent-states (run n (Some (step (next-state s execs) (next-action s execs))) (next-execs s execs))

\( (run n (\text{Some} \, t' \, \text{execs2}) \, u \) by auto

from this not-interrupt thread-not-empty-s \( t \)-empty prec-s show ?thesis using \( rs \) \( rt \) by auto

qed

{ fix \( v \)

assume ia : \( \text{ifp}^\ast \ast \, v \, u \)

have vpeq v rs rt

proof (cases \( \text{ifp}^\ast \ast \) (current s) u rule :\text{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 :\text{case-split[case-names t-prec t-not-prec]})

\( \text{case} \, t \)-prec

have thread-not-empty-t : \( \neg \text{thread-empty}(\text{execs2} \, (\text{current} \, t')) \) 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 \( ?\text{execs2.0}=?\text{execs2} \, \text{and} \, t=\text{Some} \, \text{step} \, \text{(next-state t' execs2)} \, \text{(next-action t' execs22))}] 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))

\( (run n (\text{Some} \, \text{step} \, \text{(next-state t' execs2)} \, \text{(next-action t' execs22)}) \, (\text{next-execs t' execs22}) \) u

by auto

from ia curr-ifp-u t-prec thread-not-empty-t prec-s and this \( \text{and not-interrupt and thread-not-empty-s and next-action-s-t}

show ?thesis using \( rs \) \( rt \) by auto

next

\( \text{case} \, t \)-not-prec

from curr-ifp-u t-not-prec thread-not-empty-t not-interrupt show ?thesis using \( rt \) by simp
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 unwinding-implies-view-partitioned-ind have view-partitioned-inductive: view-partitioned-ind
by blast
have purged-relation: \( \forall u \; \text{execs} \cdot \text{purged-relation} \; u \; \text{execs} \) (purge \( \text{execs} \) \( u \))
by (unfold purged-relation-def, unfold purge-def, auto)
{ fix execs s t u
assume I: equivalent-states s t u

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

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

**Shows:** NI-unrelated

**Proof:**

```plaintext
{ fix execs a n
  from unwinding-implies-view-partitioned
  have vp: view-partitioned by blast
  from vp and vpeq-reflexive
  have I: ∀ u. (run n (Some s0) execs ∥ run n (Some s0) (purge execs u)) → (λrs rt. vpeq u rs rt ∧ current rs = current rt)
  unfolding view-partitioned-def by auto
  have run n (Some s0) execs → (λs-f. run n (Some s0) (purge execs (current s-f))) → (λs-f2. output-f s-f a = output-f s-f2 a ∧ current s-f = current s-f2)
  proof(cases run n (Some s0) execs)
  case None
  thus ?thesis unfolding B-def by simp
  next
  case (Some rs)
  thus ?thesis
  proof(cases run n (Some s0) (purge execs (current rs)))
  case None
  from Some this show ?thesis unfolding B-def by simp
  next
  case (Some rt)
  from run n (Some s0) execs = Some rs Some l[THEN spec,where x=current rs]
  have vpeq: vpeq (current rs) rs rt ∧ current rs = 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 (run n (Some s0) execs = Some rs) Some
  show ?thesis unfolding B-def by auto
  qed
  qed
}
thus ?thesis unfolding NI-unrelated-def by auto
```

### 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 where domain $B$ is inactive must be equivalent to the run where domain $B$ is inactive and domain $A$ is
replaced by an attacker. Domain $C$ must be independent of domain $A$, when domain $B$ is inactive.

The aim of this subsection is to formalize the semantics where $A$ can write to $C$ via $B$ only. We define
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**

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

**Primitive rec**

```
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**

```
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**

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

**Definition**

```
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 $A ∼ B ∼ C$, but prohibits $A ∼ C$.

**Definition**

```
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**

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

**Abbreviation**

```
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)
```

EURO-MILS D31.1 Page 22 of 94
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 : : \'(\text{dom-t} \Rightarrow \text{dom-t}) \Rightarrow \text{action-t} \rightarrow \text{execution} \Rightarrow \text{dom-t} \Rightarrow \text{action-t} \rightarrow \text{execution} \Rightarrow \text{bool} \)
where ipurged-relation1 u execs1 execs2 \( \equiv \forall d . \ \text{ifp} d u \rightarrow \text{execs1} d = \text{execs2} d \) \( \land \) \( \text{intermediary} d u \rightarrow \text{execs1} d = \text{()} \)

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: \( \forall v . \ \text{ifp} v u \land \sim \text{intermediary} v u \rightarrow \text{vpeq} v s t' \)
and prec-s: \( \text{precondition} (\text{next-state} s \text{execs}) (\text{next-action} s \text{execs}) \)
shows \( \forall d \in \text{involved} (\text{next-action} s \text{execs}) . \ \text{vpeq} d s t' \)
proof
fix v 
assume involved: v \( \in \) \( \text{involved} (\text{next-action} s \text{execs}) \)
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 \( \land \sim \) 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
D31.1 – Formal Specification of a Generic Separation Kernel

from ifp-curr not-intermediary-curr ifp-v-curr intermediary-v
have False unfolding intermediary-def by auto
hence $v \equiv v' \; \text{by} \; \text{auto}$
}
ultimately
show $v \equiv v' \; \text{unfolding} \; \text{intermediary-def} \; \text{by} \; \text{auto}$
qed

thus $\text{thesis} \; \text{by} \; \text{auto}$
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 $\in$ set (remove-gateway-communications $u$ (execs $u$))
assume 2 : $a \in$ 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 $\in$ set (remove-gateway-communications $u$ exec) $\land$ $a \in$ set aseq $\land$ intermediary $v$ $u$ $\rightarrow$ $v \notin$ involved (Some $a$)
    by (induct exec, auto)
  }
  from 1 2 3 this show $\text{thesis} \; \text{by} \; \text{metis}$
  qed
}
from this
show $\text{thesis}$
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 $\land$ iequivalent_states $s$ $t$ $u$ $\land$ ipurged_relation $u$ 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 $\text{case by} \; \text{auto}$
  next
  case (2 $n$ execs $t$ $u$ execs2)
  show $\text{case by} \; \text{simp}$
  next
  case (3 $n$ execs $t$ $u$ execs2)
    assume interrupt-s: interrupt (Suc $n$)
    assume IH: ($\forall$ t execs2. does-not-communicate-with-gateway $u$ execs $\land$
      iequivalent_states (Some (cswitch (Suc $n$) $s$)) $t$ $u$ $\land$ ipurged_relation $u$ execs execs2 $\rightarrow$
\[
\text{iequivalent-states (run } n \ (\text{Some (cswitch (Suc } n \ s)) \ \text{execs}) \ (\text{run } n \ t \ \text{execs2}) \ u) \\
\]

\[
\{ \\
\text{fix } t' = \\text{'state-t} \\
\text{assume } t = \text{Some } t' \\
\text{fix } rs \\
\text{assume } rs: \text{run } (\text{Suc } n) \ (\text{Some } s) \ \text{execs} = \text{Some } rs \\
\text{fix } rt \\
\text{assume } rt: \text{run } (\text{Suc } n) \ (\text{Some } t') \ \text{execs2} = \text{Some } rt \\
\text{assume } \text{no-gateway-comm: does-not-communicate-with-gateway } u \ \text{execs} \\
\text{assume } vpeq-s-t: \forall \ v. \ \text{ifp } v \ u \ \land \text{~intermediary } v \ u \ \rightarrow vpeq \ v \ s \ t' \\
\text{assume } \text{current-s-t: current } s = \text{current } t' \\
\text{assume } \text{purged-a-a2: ipurged-relation1 } u \ \text{execs} \ \text{execs2} \\
\}
\]

\[
\text{from current-s-t cswitch-independent-of-state} \\
\text{have current-ns-nt: current } (\text{cswitch } (\text{Suc } n) \ s) = \text{current } (\text{cswitch } (\text{Suc } n) \ t') \\
\text{by blast} \\
\text{from cswitch-consistency vpeq-s-t} \\
\text{have vpeq-ns-nt: } \forall \ v. \ \text{ifp } v \ u \ \land \text{~intermediary } v \ u \ \rightarrow vpeq \ v \ (\text{cswitch } (\text{Suc } n) \ s) \ (\text{cswitch } (\text{Suc } n) \ t') \\
\text{by auto} \\
\text{from no-gateway-comm current-ns-nt vpeq-ns-nt interrupt-s vpeq-reflexive current-s-t purged-a-a2 IH [where u = u \text{ and } t = \text{Some (cswitch (Suc } n \ t')} \text{ and } ?\text{execs2.0=execs2}] \\
\text{have current-rs-rt: current } rs = \text{current } rt \ \text{using rs rt by(auto)} \\
\}
\]

\[
\text{from current-rs-rt and this have iequivalent-states (Some } rs \ (\text{Some } rt) \ u \ \text{by auto)} \\
\]

\[\text{thus } ?\text{case by(simp add-option.splits.cases t.simp+)}\]

\[\text{next}\]

\[
\text{case (Suc } n \ \text{execs}) \ (\text{Suc } n \ \text{execs}) \\
\text{assume not-interrupt: ~interrupt } (\text{Suc } n) \\
\text{assume thread-empty-s: thread-empty } (\text{execs } (\text{current } s)) \\
\]

\[
\text{assume IH: } (\forall u \ \text{execs2. does-not-communicate-with-gateway } u \ \text{execs} \land \text{iequivalent-states } (\text{Some } s) \ u \ \land \text{ipurged-relation1 } u \ \text{execs} \ \text{execs2} \ \rightarrow \text{iequivalent-states } (\text{run } n \ (\text{Some } s) \ \text{execs}) \ (\text{run } n \ t \ \text{execs2}) \ u) \\
\]

\[
\{ \\
\text{fix } t' \\
\text{assume } t: t = \text{Some } t' \\
\text{fix } rs \\
\text{assume } rs: \text{run } (\text{Suc } n) \ (\text{Some } s) \ \text{execs} = \text{Some } rs \\
\text{fix } rt \\
\text{assume } rt: \text{run } (\text{Suc } n) \ (\text{Some } t') \ \text{execs2} = \text{Some } rt \\
\text{assume } \text{no-gateway-comm: does-not-communicate-with-gateway } u \ \text{execs} \\
\text{assume } vpeq-s-t: \forall \ v. \ \text{ifp } v \ u \ \land \text{~intermediary } v \ u \ \rightarrow vpeq \ v \ s \ t' \\
\text{assume } \text{current-s-t: current } s = \text{current } t' \\
\text{assume } \text{purged-a-a2: ipurged-relation1 } u \ \text{execs} \ \text{execs2} \\
\}
\]

\[
\text{from ifp-reflexive vpeq-s-t have vpeq-u-s-t: vpeq } u \ s \ t' \ \text{unfolding intermediary-def by auto} \\
\text{from step-atomicity current-next-state current-s-t have current-s-nt: current } s = \text{current } (\text{step } (\text{next-state } t') \ \text{execs2}) \ (\text{next-action } t' \ \text{execs2})\]

unfolding step-def by (cases next-action s execs,cases next-action t' execs2,simp,simp,cases next-action t' execs2,simp,simp)
from vpeq-s-t have vpeq-curr-s-t: ifp (current s) u \land \neg \text{intermediary} (current s) u \rightarrow vpeq \ (current s) s t'
by auto

have iequivalent-states (run \ (Suc n) \ (Some s) execs) (run \ (Suc n) \ (Some t') execs2) u
proof(cases thread-empty(execs2 \ (current t')))

case True

from purged-a-a2 and vpeq-s-t and current-s-t IH\[\text{where } t=\text{Some } t' \text{ and } u=u \text{ and } ?\text{execs}2.0=\text{execs2}\]
no-gateway-comm

have iequivalent-states (run n (Some s) execs) (run n (Some t') execs2) u using rs rt by(auto)
from this not-interrupt True thread-empty-s
show ?thesis using rs rt by(auto)

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

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

  from I ifp-v v-not-intermediary have not-ifp-curr-v: \neg \text{ifp} (current t') v unfolding intermediary-def by auto
  from not-ifp-curr-v prec-t locally-respects[THEN spec,THEN spec,THEN spec,where x1=next-state t' execs2 and x=x and x2=the \ (next-action t' execs2)]
current-next-state vpeq-reflexive

  have vpeq v \ (next-state t' execs2) \ (step \ (next-state t' execs2) \ (next-action t' execs2))
  unfolding step-def precondition-def B-def
by \ (cases next-action t' execs2.auto)
from this vpeq-transitive not-ifp-curr-v locally-respects-next-state

  have vpeq-t-n: vpeq v t' \ (step \ (next-state t' execs2) \ (next-action t' execs2))
by blast
from vpeq-s-t ifp-v v-not-intermediary vpeq-t-n vpeq-transitive vpeq-symmetric vpeq-reflexive

  have vpeq v s \ (step \ (next-state t' execs2) \ (next-action t' execs2))
by \ (metis)
}

  hence vpeq-ns-nt: \forall v. ifp v u \land \neg \text{intermediary} v u \rightarrow vpeq v s \ (step \ (next-state t' execs2) \ (next-action t' execs2)) by auto
from False purged-a-a2 current-s-t thread-empty-s have purged-a-na2: ipursed-relation1 u execs (next-execs t' execs2)

  unfolding ipursed-relation1-def next-execs-def by(auto)
from vpeq-ns-nt no-gateway-comm

and IH[\text{where } t=\text{Some} \ (step \ (next-state t' execs2) \ (next-action t' execs2))] and \ ?\text{execs}2.0=(next-execs t' execs2) and u=u]
and current-s-nt purged-a-na2
have eq-ns-nt: iequivalent-states (run n (Some s) execs)

  \ (run n (Some \ (step \ (next-state t' execs2) \ (next-action t' execs2)))) \ (next-execs t' execs2)
execs2) u by auto
  from prec-t eq-ns-nt not-interrupt False thread-empty-s
  show ?thesis using t rs rt by(auto)
qed 
}
thus ?case by(simp add-option.splits,cases t,simp+)
next
  case (5 n execs s t u execs2)
    assume not-interrupt: ~interrupt (Suc n)
    assume thread-not-empty-s: ~thread-empty(execs (current s))
    assume not-prec-s: ~precondition (next-state s execs) (next-action s execs)
    hence run (Suc n) (Some s) execs = None using not-interrupt thread-not-empty-s by simp
thus ?case by(simp add-option.splits)
next
  case (6 n execs s t u execs2)
    assume not-interrupt: ~interrupt (Suc n)
    assume thread-not-empty-s: ~thread-empty(execs (current s))
    assume prec-s: precondition (next-state s execs) (next-action s execs)
    assume IH: (∀ u execs2, does-not-communicate-with-gateway u (next-execs s execs) ∧
      inequivalent-states (Some (step (next-state s execs) (next-action s execs))) t u ∧
      ipurged-relation1 u (next-execs s execs) execs2 →
      inequivalent-states
        (run n (Some (step (next-state s execs) (next-action s execs))) (next-execs s execs))
        (run n t execs2) u)
      
      fix t'
      assume t: t = Some t'
      fix rs
      assume rs: run (Suc n) (Some s) execs = Some rs
      fix rt
      assume rt: run (Suc n) (Some t') execs2 = Some rt

    assume no-gateway-comm: does-not-communicate-with-gateway u execs
    assume vpeq-s-t: ∀ v. ifp v u ∧ ~intermediary v u → vpeq v s t'
    assume current-s-t: current s = current t'
    assume purged-a-a2: ipurged-relation1 u execs execs2

      from ifp-reflexive vpeq-s-t have vpeq-u-s-t: vpeq u s t' unfolding intermediary-def by auto
      from step-atomicity 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 step-atomicity current-next-state current-s-t have current-ns-t: current (step (next-state s execs) (next-action s execs)) = current t'
        unfolding step-def
        by (cases next-action s execs,auto)
      from vpeq-s-t have vpeq-curr-s-t: ifp (current s) u ∧ ~intermediary (current s) u → vpeq (current s) s t'
        unfolding intermediary-def by auto
      from current-s-t purged-a-a2
        have eq-execs ifp (current s) u ∧ ~intermediary (current s) u → execs (current s) = execs2 (current s)
            by(auto simp add: ipurged-relation1-def)
      from vpeq-involved-domains no-gateway-comm vpeq-s-t vpeq-involved-domains prec-s
        have vpeq-involved: ifp (current s) u ∧ ~intermediary (current s) u → (∀ d ∈ involved (next-action s execs) . vpeq d s t')
          by blast
\[
\text{from current-s-t next-exec-consistent}[\text{THEN spec,THEN spec,THEN spec,where x2=s and x1=t' and x=execs}]
\]
\[
vpeq-curr-s-t vpeq-involved \quad \text{have next-exec-cons-ifp (current s) u \land \neg \text{-intermediary (current s) u \rightarrow next-execs t' execs = next-execs s execs}}\]
\[
\text{by(auto simp add: next-execs-def)}
\]
\[
\text{from current-s-t and purged-a-a2 and thread-not-empty-s next-action-consistent}[\text{THEN spec,THEN spec,where x1=s and x=t'}] vpeq-curr-s-t vpeq-involved \quad \text{have next-action-s-t-ifp (current s) u \land \neg \text{-intermediary (current s) u \rightarrow next-action t' execs2 = next-action s execs}}\]
\[
\text{by(unfold next-action-def,unfold ipurged-relation1-def,auto)}
\]
\[
\text{from purged-a-a2 and thread-not-empty-s and current-s-t}
\]
\[
\text{have thread-not-empty-t-ifp (current s) u \land \neg \text{-intermediary (current s) u \rightarrow \neg thread-empty(execs2 (current t'))}}\]
\[
\text{unfolding ipurged-relation1-def by auto}
\]
\[
\text{have vpeq-ns-nt-1: } \forall a . \text{ precond (next-state s execs) a } \land \text{ precond (next-state t' execs) a } \equiv \text{ ifp (current s) u } \land \neg \text{-intermediary (current s) u } \Rightarrow (\forall v . \text{ ifp v u } \land \neg \text{-intermediary v u } \rightarrow \text{ vpeq v (step (next-state s execs) a)) (step (next-state t' execs) a))}
\]
\[
\text{proof}\}
\]
\[
\text{fix a}
\]
\[
\text{assume precs: precond (next-state s execs) a } \land \text{ precond (next-state t' execs) a}
\]
\[
\text{assume ifp-curr: ifp (current s) u } \land \neg \text{-intermediary (current s) u}
\]
\[
\text{from ifp-curr precs}
\]
\[
\text{next-state-consistent}[\text{THEN spec,THEN spec,where x1=s and x=t'}] vpeq-curr-s-t vpeq-s-t
\]
\[
\text{current-next-state current-s-t weakly-step-consistent}[\text{THEN spec,THEN spec,THEN spec,THEN spec,where x3=next-state s execs and x2=next-state t' execs and x=the a}]
\]
\[
\text{show } \forall v . \text{ ifp v u } \land \neg \text{-intermediary v u } \rightarrow \text{ vpeq v (step (next-state s execs) a)) (step (next-state t' execs) a))}
\]
\[
\text{unfolding step-def precond-def B-def}
\]
\[
\text{by (cases a auto)}
\]
\[
\text{qed}
\]
\[
\text{have no-gateway-comm-na: does-not-communicate-with-gateway u (next-execs s execs)}
\]
\[
\text{proof}\}\}
\]
\[
\text{fix a}
\]
\[
\text{assume a } \in \text{ actions-in-execution (next-execs s execs u)}
\]
\[
\text{from this no-gateway-comm[unfolded does-not-communicate-with-gateway-def,THEN spec,where x=a]}
\]
\[
\text{next-execs-subset[THEN spec,THEN spec,THEN spec,where x2=s and x1=execs and x0=u]}
\]
\[
\text{have } \forall v . \text{ intermediary v u } \rightarrow v \notin \text{ involved (Some a)}
\]
\[
\text{unfolding actions-in-execution-def}
\]
\[
\text{by (auto)}
\]
\[
\text{thus ?thesis unfolding does-not-communicate-with-gateway-def by auto}
\]
\[
\text{qed}
\]
\[
\text{have iequivalent-states (run (Suc n) (Some s) execs) (run (Suc n) (Some t') execs2) u}
\]
\[
\text{proof (cases ifp (current s) u } \land \neg \text{-intermediary (current s) u rule case-split\{case-names T F\})}
\]
\[
\text{case T}
\]
\[
\text{show ?thesis}
\]
\[
\text{proof (cases thread-empty(execs2 (current t')) rule case-split\{case-names T2 F2\})}
\]
\[
\text{case F2}
\]
\[
\text{show ?thesis}
\]
\[
\text{proof (cases precond (next-state t' execs2) (next-action t' execs2) rule case-split\{case-names T3 F3\})}
\]
\[
\text{case T3}
\]
\[
\text{from T purged-a-a2 current-s-t}
\]
\[
\text{next-execs-consistent[THEN spec,THEN spec,where x1=s and x=t'] vpeq-curr-s-t vpeq-involved}
\]
\[
\text{have purged-na-na2: ipurged-relation1 u (next-execs s execs) (next-execs t' execs2)
\text{ unfolding ipurged-relation1-def next-execs-def}
\text{by auto}
\text{from IH[where t=Some (step (next-state t' execs2) (next-action t' execs2) and ?execs2.0=next-execs t'
\]
execs2 and \( w = u \)

\[
\text{purged-na-na2 current-ns-nt vpeq-ns-nt-1 [\{ \text{where } a = (\text{next-action } s \text{ execs} ) \} T T3 \text{ prec-s next-action-s-t eq-execss current-s-t no-gateway-comm-na} \text{ have eq-ns-nt: } \text{iequivalent-states (run } n (\text{Some (step (next-state } s \text{ execs} ) (\text{next-action } s \text{ execs} ))) (\text{next-execss } s \text{ execs} )) \text{ (run } n (\text{Some (step (next-state } t' \text{ execs2} ) (\text{next-action } t' \text{ execs2} ))) (\text{next-execss } t' \text{ execs2}) \text{ )}\] \]

\( u \)

unfolding next-state-def
by (auto,metis)
from this not-interrupt thread-not-empty-s prec-s F2 T3
have current-rs-rt: current rs = current rt using rs rt by auto
{
  fix \( v \)
  assume ia: ifp \( v u \) \& \( \lnot \text{intermediary } v u \)
  from this eq-ns-nt not-interrupt thread-not-empty-s prec-s F2 T3
  have vpeq v rs rt using rs rt by auto
}
from this and current-rs-rt show ?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 prec-s next-action-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: \( \lnot \text{intermediary (current s) u unfolding ipurged-relation1-def by auto} \)
let \(?nt = \text{if thread-empty(execs2 (current t')) then t' else step (next-state } t' \text{ execs2} ) (\text{next-action } t' \text{ execs2} )) \)
let \(?na2 = \text{if thread-empty(execs2 (current t')) then execs2 else next-execss } t' \text{ execs2} \)

have prec-t: \( \lnot \text{thread-empty(execs2 (current t')) \implies \text{precondition (next-state } t' \text{ execs2} ) (\text{next-action } t' \text{ execs2} )) \)
proof
  assume thread-not-empty-t: \( \lnot \text{thread-empty(execs2 (current t'))} \)
  {
    assume not-prec-t: \( \lnot \text{precondition (next-state } t' \text{ execs2} ) (\text{next-action } t' \text{ execs2} ) \)
    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: ifp \( v u \)
  assume v-not-intermediary: \( \lnot \text{intermediary } v u \)
have not-ifp-curr-v: ¬ifp (current s) v
proof
assume ifp-curr-v: ifp (current s) v
thus False
proof-
{ assume ind-source (current s) u
from this ifp-curr-v ifp-v have intermediary v u unfolding intermediary-def by auto
from this v-not-intermediary have False unfolding intermediary-def by auto }
moreover
{ assume unrelated: unrelated (current s) u
from this ifp-v ifp-curr-v have False using rtranclp-trans r-into-rtranclp by metis }
ultimately show ?thesis using 1 2 by auto
qed

from this current-next-state[THEN spec,THEN spec,where x1=s and x=execs] prec-s
locally-respects[THEN spec,THEN spec,where x=next-state s execs] vpeq-reflexive
have vpeq v (next-state s execs) (step (next-state s execs) (next-action s execs))
unfolding step-def precondition-def B-def
by (cases next-action s execs,auto)
from not-ifp-curr-v this locally-respects-next-state vpeq-transitive
have vpeq-s-ns: vpeq v s (step (next-state s execs) (next-action s execs))
by blast
from current-ns-nt current-ns-t current-next-state
have vpeq-t-nt: thread-empty (execs2 (current t')) → vpeq v (next-state t' execs2) (step (next-state t' execs2)
(next-action t' execs22)) by metis
from this vpeq-reflexive
have vpeq-t-nt: vpeq v t' ?nt
by auto
from vpeq-s-t ifp-v v-not-intermediary
have vpeq v s t' by auto
from this vpeq-s-ns vpeq-t-nt vpeq-transitive vpeq-symmetric vpeq-reflexive
have vpeq v (step (next-state s execs) (next-action s execs)) ?nt
by (metis (hide-lams, no-types))
}
hence vpeq-ns-nt: ∀ v. ifp 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: ipursed-relation1 u
(next-execs s execs) ?na2
unfolding ipursed-relation1-def next-execs-def intermediary-def by(auto)
from current-ns-nt current-ns-t current-next-state have current-ns-nt:
current (step (next-state s execs) (next-action s execs)) = current ?nt
by auto
from prec-s vpeq-ns-nt no-gateway-comm-na
and IH[where $t = $\text{Some } ?nt$ and $\text{execs}2.0 = ?na2$ and $u = u$]
and current-ns-nt purged-na-na2
have eq-ns-nt: inequivalent-states (run n ($\text{Some } (\text{step } (\text{next-state } s \text{ execs}2) (\text{next-action } s \text{ execs}))$) (next-exec
s execs)) (run n ($\text{Some } ?nt$) ?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 $v u$ $\land$ $\neg$intermediary $v u$
  from this eq-ns-nt not-interrupt thread-not-empty-s prec-s prec-t
  have vpeq $v rs rt$ using $rs rt$ by (cases thread-empty(execs2 (current $t'$)), simp, simp)
}
from current-rs-rt and this show ?thesis using $rs rt$ by auto
qed

hence iview-partitioned-inductive: $\forall$ $u$ $s$ $t$ execs execs2 $n$. does-not-communicate-with-gateway $u$ execs $\land$ inequivalent-states
$s$ $t$ $u$ $\land$ ipurged-relation1 $u$ execs execs2 $\rightarrow$ inequivalent-states (run n $s$ execs) (run n $t$ execs2) $u$
by blast
have ipurged-relation: $\forall$ $u$ execs. ipurged-relation1 $u$ (ipurge-l execs $u$) (ipurge-r execs $u$)
by(unfold ipurged-relation1-def , unfold ipurge-l-def , unfold ipurge-r-def , auto)
{
  fix execs $s$ $t$ $n$ $u$
  assume 1: inequivalent-states $s$ $t$
  from ifp-reflexive
  have dir-source: $\forall$ $u$. ifp $u u$ $\land$ $\neg$intermediary $u u$ unfolding intermediary-def by auto
  from ipurge-l-removes-gateway-communications
  have does-not-communicate-with-gateway $u$ (ipurge-l execs $u$)
  by auto
  from 1 this view-partitioned-inductive ipurged-relation
  have inequivalent-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 $u$) $\parallel$ run n $t$ (ipurge-r execs $u$) $\rightarrow$ ($\lambda rs rt. \text{vpeq } u rs rt \land current rs = current rt$)
  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 $\Rightarrow$ 'state-t option $\Rightarrow$ bool
where mcurrents $m1 m2 = m1 \parallel m2 \rightarrow (\lambda s t. current s = current t)$

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 $s t$
s show mcurrents (run n $s$ execs) (run n $t$ execs2)
proof
{
  fix n s execs t execs2
  have mcurrents s t \longrightarrow mcurrents (run n s execs) (run n t execs2)
proof (induct n s execs arbitrary: t execs2 rule: run.induct)
case (\{ s \exists t \}) execs2
  from this show \{ \text{case using current-s-t}\ unfolding B-def by auto
next
case (2 n execs t execs2)
  show \{ \text{case unfolding mcurrents-def by (auto)}
next
case (3 n s execs t execs2)
  assume interrupt\_\_\_interrupt (Suc n)
  assume IH:\ (\forall t \exists execs2. mcurrents (Suc \{ current (Suc n) s \}) t \longrightarrow mcurrents (run n \{ current (Suc n) s \}) (run n t execs2))
  \{
    fix t'
    assume t: t = (Some t')
    assume curr: mcurrents (Some s) t
    from t curr cswitch-independent-of-state[THEN spec,THEN spec,THEN spec,where x1=s] have current-s-t: current (Suc n) s = current (Suc n) t'
    unfolding mcurrents-def by simp
    from current-s-t IH[where t=Some (cswitch (Suc n) t') and ?execs2.0=?execs2]
    have mcurrents-s-t: mcurrents (run n \{ current (Suc n) s \}) execs (run n \{ current (Suc n) t \}) execs2 (auto)
    unfolding mcurrents-def by (auto)
    from mcurrents-s-t interrupt t
    have mcurrents (run (Suc n) \{ current (Suc n) s \}) execs (run (Suc n) t execs2)
    unfolding mcurrents-def B2-def B-def by (cases run n \{ current (Suc n) s \}) execs, cases run (Suc n) t execs2,auto
  }
  thus \{ \text{case unfolding mcurrents-def B2-def by (cases t,auto)}
next
case (4 n execs s t execs2)
  assume not-interrupt: \neg interrupt (Suc n)
  assume thread-empty-s: thread-empty(execs (current s))
  assume IH:\ (\forall t \exists execs2. mcurrents (Some s) t \longrightarrow mcurrents (run n (Some s) execs) (run n t execs2))
  \{
    fix t'
    assume t: t = (Some t')
    assume curr: mcurrents (Some s) t
    \{
      assume thread-empty-t: thread-empty(execs2 (current t'))
      from t curr not-interrupt thread-empty-s this IH[where ?execs2.0=?execs2 and t=Some t']
      have mcurrents (run (Suc n) (Some s) execs) (run (Suc n) t execs2) by auto
    }
    moreover
    \{
      assume not-prec-t: \neg thread-empty(execs2 (current t')) \land \neg precondition (next-state t' execs2) (next-action t' execs2)
      from t this not-interrupt
      have mcurrents (run (Suc n) (Some s) execs) (run (Suc n) t execs2)
      unfolding mcurrents-def by (simp add: rewrite-B2-cases)
    }
    moreover
    \{

\[\begin{align*}
\text{assume } & \text{step}-t: \neg \text{thread-empty}(\text{execs2} (\text{current} t')) \land \text{precondition} (\text{next-state} t' \text{ execs2}) (\text{next-action} t' \\
\text{execs2}) \\
\text{have } & m\text{currents} (\text{Some} s) (\text{Some (step (next-state t' execs2) (next-action t' execs2)))}) \\
\text{using } & \text{step-atomicity curr t current-next-state unfolding mcurrents-def} \\
\text{unfolding } & \text{step-def} \\
\text{by } & (\text{cases next-action t' execs2,auto}) \\
\text{from } & t \text{ step-t curr not-interrupt thread-empty-s this IH[where ?execs2.0=next-execs t' execs2 and t=Some} \\
\text{(step (next-state t' execs2) (next-action t' execs2))}] \\
\text{have } & m\text{currents} (\text{run (Suc n) (Some s) execs}) (\text{run (Suc n) t execs2}) \\
\text{by } & \text{auto} \\
\text{ultimately have } & m\text{currents} (\text{run (Suc n) (Some s) execs}) (\text{run (Suc n) t execs2}) \text{ by blast} \\
\text{thus } & ?\text{case unfolding mcurrents-def B2-def by (cases t,auto) next} \\
\text{case } & (5 n \text{ execs s t execs2}) \\
\text{assume } & \text{not-interrupt-s: \neg \text{interrupt} (Suc n) } \\
\text{assume } & \text{thread-not-empty-s: \neg \text{thread-empty} (\text{execs (current s)}) } \\
\text{assume } & \text{not-prec-s: \neg \text{precondition} (\text{next-state s execs}) (\text{next-action s execs}) } \\
\text{hence } & \text{run (Suc n) (Some s) execs = None using not-interrupt-s thread-not-empty-s by simp} \\
\text{thus } & ?\text{case unfolding mcurrents-def by (simp add-option splits) next} \\
\text{case } & (6 n \text{ execs s t execs2}) \\
\text{assume } & \text{not-interrupt: \neg \text{interrupt} (Suc n) } \\
\text{assume } & \text{thread-not-empty-s: \neg \text{thread-empty} (\text{execs (current s)}) } \\
\text{assume } & \text{prec-s: \text{precondition} (\text{next-state s execs}) (\text{next-action s execs}) } \\
\text{assume IH: } & (\land \text{execs2}. \\
& \text{mcurrents} (\text{Some (step (next-state s execs) (next-action s execs))}) t \rightarrow \\
& \text{mcurrents} (\text{run n (Some (step (next-state s execs) (next-action s execs))}) (\text{next-execs s execs}) (\text{run n t execs2})) \\
& \{ \\
\text{fix } & t' \\
\text{assume } & t: t = (Some t') \\
\text{assume } & \text{curr: mcurrents (Some s) t} \\
& \{ \\
\text{assume } & \text{thread-empty-t: \neg \text{thread-empty} (\text{execs2 (current t'))} \\
\text{have } & \text{mcurrents} (\text{Some (step (next-state s execs) (next-action s execs))}) (\text{Some t'}) \\
\text{using } & \text{step-atomicity curr t current-next-state unfolding mcurrents-def} \\
\text{unfolding } & \text{step-def} \\
\text{by } & (\text{cases next-action s execs,auto}) \\
\text{from } & t \text{ curr not-interrupt thread-not-empty-s prec-s thread-empty-t this IH[where ?execs2.0=execs2 and} \\
\text{t=Some t']} \\
\text{have } & \text{mcurrents} (\text{run (Suc n) (Some s) execs}) (\text{run (Suc n) t execs2}) \\
\text{by } & \text{auto} \\
\text{moreover } & \{ \\
\text{assume } & \text{not-prec-t: \neg \text{thread-empty} (\text{execs2 (current t'))} \land \neg \text{precondition} (\text{next-state t' execs2}) (\text{next-action t'} \\
\text{execs2}) } \\
\text{from } & t \text{ this not-interrupt} \\
\text{have } & \text{mcurrents} (\text{run (Suc n) (Some s) execs}) (\text{run (Suc n) t execs2}) \\
\text{unfolding } & \text{mcurrents-def B2-def by (auto)} \\
\text{moreover } & \{ \\
\text{assume } & \text{step-t: \neg \text{thread-empty} (\text{execs2 (current t'))} \land \text{precondition} (\text{next-state t' execs2}) (\text{next-action t'} \\
\text{execs2}) }
\end{align*}\]
proof
NI-indirect-sources
shows
unwinding-implies-NI-indirect-sources
theorem
unwinding-implies-NI-indirect-sources:
shows
NI-indirect-sources
proof

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-noi-interrupt thread-noi-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 ?case unfolding mcurrents-def B2-def by(cases t,auto)
qed
}
thus ?thesis using current-s-t by auto
qed


Salve [THEN spec \( \text{where } x=\text{current s-f} \) ]

have vpeq (current s-f) s-ipurge-l s-ipurge-r \( \land \) current s-ipurge-l = current s-ipurge-r

unfolding B-def by auto

from this \( \text{have output-f s-ipurge-l a = output-f s-ipurge-r a} \)

using output-consistent by auto

from \( \text{run n (Some s0)} \) execs = Some s-f \( \text{run n (Some s0)} \) (ipurge-l execs (current s-f)) = Some s-ipurge-b

this Some

show \( ?\text{thesis unfolding B-def by auto} \)

qed

qed

qed

}\)

thus \( ?\text{thesis unfolding NI-indirect-sources-def by auto} \)

qed

---

**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 definition. 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 \( \text{AS}_\text{set} \). Consider for example the following action sequence:

\[
\gamma = [\text{COPY_Init}, \text{COPY_Check}, \text{COPY_Copy}]
\]

If action sequence \( \gamma \) is a member of \( \text{AS}_\text{set} \), this means that the attack surface contains an action COPY, which consists of three consecutive atomic kernel actions. Interrupts can occur anywhere between these atomic actions.

Given a set of valid action sequences such as \( \gamma \), generic function precondition can be defined. It now consists of 1.) a generic invariant and 2.) more refined preconditions for the current action.

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

- “\( \text{AS}_\text{precondition s d COPY_Init} \)” since COPY_Init is the start of an action sequence.

- “\( \text{AS}_\text{precondition (step s COPY_Init) d COPY_Check} \)” since (COPY_Init, COPY_CHECK) is a sub sequence.

- “\( \text{AS}_\text{precondition (step s COPY_Check) d COPY_Copy} \)” 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.

\[
\text{locale Interruptible-Separation-Kernel = Separation-Kernel kstep output-f s0 current cswitch interrupt kprecondition realistic-execution control kinvolved ifp vpeq}
\]

\[
\text{for kstep :: 'state-t ⇒ 'action-t ⇒ 'state-t}
\]

\[
\text{and output-f :: 'state-t ⇒ 'action-t ⇒ 'output-t}
\]

\[
\text{and s0 :: 'state-t}
\]

\[
\text{and current :: 'state-t ⇒ 'dom-t — Returns the currently active domain}
\]

\[
\text{and cswitch :: 'time-t ⇒ 'state-t ⇒ 'state-t — Switches the current domain}
\]

\[
\text{and interrupt :: 'time-t ⇒ bool — Returns t iff an interrupt occurs in the given state at the given time}
\]

\[
\text{and kprecondition :: 'state-t ⇒ 'action-t ⇒ bool — Returns t if an precondition holds that relates the current action to the state}
\]

\[
\text{and realistic-execution :: 'action-t execution ⇒ bool — In this locale, this function is completely unconstrained.}
\]

\[
\text{and control :: 'state-t ⇒ 'dom-t ⇒ 'action-t execution ⇒ (('action-t option) × 'action-t execution × 'state-t)}
\]

\[
\text{and kinvolved :: 'action-t ⇒ 'dom-t set}
\]

\[
\text{and ifp :: 'dom-t ⇒ 'dom-t ⇒ bool}
\]

\[
\text{and vpeq :: 'dom-t ⇒ 'state-t ⇒ 'state-t ⇒ bool}
\]

\[
+\text{ fixes AS-set :: ('action-t list) set — Returns a set of valid action sequences, i.e., the attack surface}
\]

\[
\text{and invariant :: 'state-t ⇒ bool}
\]

\[
\text{and AS-precondition :: 'state-t ⇒ 'dom-t ⇒ 'action-t ⇒ bool}
\]

\[
\text{and aborting :: 'state-t ⇒ 'dom-t ⇒ 'action-t ⇒ bool}
\]

\[
\text{and waiting :: 'state-t ⇒ 'dom-t ⇒ 'action-t ⇒ bool}
\]

\[
\text{assumes empty-in-AS-set :: [] ∈ AS-set}
\]

\[
\text{and invariant-s0 :: invariant s0}
\]

\[
\text{and invariant-after-cswitch ∀ s n . invariant s → invariant (cswitch n s)}
\]

\[
\text{and precondition-after-cswitch ∀ s d n a . AS-precondition s d a → AS-precondition (cswitch n s) d a}
\]

\[
\text{and AS-pre-first-action :: s d aseq . invariant s ∧ aseq ∈ AS-set ∧ aseq ≠ [] → AS-precondition s d (hd aseq)}
\]

\[
\text{and AS-pre-first-step :: s a a' . (3 aseq ∈ AS-set . is-sub-seq a a' aseq) ∧ invariant s ∧ AS-precondition s (current s) a ∧ ~aborting s (current s) a ∧ ~waiting s (current s) a → AS-precondition (kstep s a) (current s) a')}
\]

\[
\text{and AS-pre-dom-independent :: s d a a' . current s ≠ d ∧ AS-precondition s d a → AS-precondition (kstep s a a') d a}
\]

\[
\text{and spec-of-invariant :: s a . invariant s → invariant (kstep s a)}
\]

\[
\text{and kprecondition-def :: kprecondition s a ≡ invariant s ∧ AS-precondition s (current s) a}
\]

\[
\text{and realistic-execution-def :: realistic-execution aseq ≡ set aseq ≤ AS-set}
\]

\[
\text{and control-spec :: s d aseq . case control s d aseq of (a,aseqs',s') ⇒}
\]

\[
\text{(thread-empty aseqs ∧ (a,aseqs') = (None,[[]])) → — Nothing happens}
\]

\[
\text{(aseqs ≠ [] ∧ hd aseqs ≠ []) ∧ ~aborting s′ d (the a) ∧ ~waiting s′ d (the a) ∧ (a,aseqs') =}
\]

\[
\text{(Some (hd (hd aseqs)), (tl (hd aseqs)))(tl aseqs))) → — Execute the first action of the current action sequence}
\]

\[
\text{(aseqs ≠ [] ∧ hd aseqs ≠ []) ∧ waiting s′ d (the a) ∧ (a,aseqs',s') = (Some (hd (hd aseqs)), aseqs, s)) → — Nothing happens, waiting to execute the next action}
\]
(a, aseqs') = (None, tl aseqs)

and next-action-after-cswitch: ∀ s n d aseqs . fst (control (cswitch n s) d aseqs) = fst (control s d aseqs)

and next-action-after-next-state: ∀ s execs d . current s ≠ d → fst (control (next-state s execs) d (execs d))

= None ∨ fst (control (next-state s execs) d (execs d)) = fst (control s d (execs d))

and next-action-after-step: ∀ s a d aseqs . current s ≠ d → fst (control (step s a) d aseqs) = fst (control s d aseqs)

and next-state-precondition: ∀ s d a execs. AS-precondition s d a → AS-precondition (next-state s execs) d a

and next-state-invariant: ∀ s execs . invariant s → invariant (next-state s execs)

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

begin

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_equals_run is proven by doing induction. It is however not inductive and can there-
fore not be proven directly: a realistic execution is not necessarily realistic after performing one ac-
tion. We generalize to do induction: Predicate realistic_executions_ind is the inductive version of realistic_executions. All action sequences in the tail of the executions must be complete action sequences (i.e.,
they must be from 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 ∈ \exists 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.

definition 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 \( ?c = \text{control} \ s \ (\text{current} \ s) \ (\text{execs} \ (\text{current} \ s)) \)

\{
    \text{assume} \ c-\text{empty}: \text{let} \ (a, \text{aseqs}', s') = ?c \in \\
    (a, \text{aseqs}') = (\text{None}, [])
\}

\text{from} \text{na-def} \ d-\text{is-curr} \ c-\text{empty}
\quad \text{have} \ ?\text{thesis}
\quad \text{unfolding} \ \text{realistic-executions-ind-def} \ \text{next-exec-def} \ \text{by} \ (\text{auto})
\}

\text{moreover}
\{
    \text{let} \ ?ct = \text{execs} \ (\text{current} \ s)
    \text{let} \ ?\text{execs}' = (tl \ (hd \ ?ct)) \#(tl \ ?ct)
    \text{let} \ ?a' = \text{Some} \ (hd \ (hd \ ?ct))
    \text{assume} \ \text{hd-thread-not-empty}: \text{hd} \ (\text{execs} \ (\text{current} \ s)) \neq []
    \text{assume} \ c-\text{executing}: \text{let} \ (a, \text{aseqs}', s') = ?c \in \\
    (a, \text{aseqs}') = (?a', ?\text{execs}')
\}

\text{from} \text{na-def} \ c-\text{executing} \ d-\text{is-curr}
\quad \text{have} \ \text{as-defs}: \text{aseq} = \text{tl} \ (\text{hd} \ ?ct) \land \text{aseqs} = \text{tl} \ ?ct
\quad \text{unfolding} \ \text{next-exec-def} \ \text{by} \ (\text{auto})
\quad \text{from} \ \text{realistic}[\text{unfolded realistic-executions-ind-def}, \text{THEN} \ \text{spec}, \text{where} \ x=d] \ d-\text{is-curr}
\quad \text{have} \ \text{subset}: \text{set} \ (tl \ ?\text{execs}') \subseteq \text{AS-set}
\quad \text{unfolding} \ \text{Let-def realistic-AS-partial-def}
\quad \text{by} \ (\text{cases} \ \text{execs} \ d, \text{auto})
\quad \text{from} \ d-\text{is-curr} \ \text{thread-not-empty} \ \text{hd-thread-not-empty} \ \text{realistic}[\text{unfolded realistic-executions-ind-def}, \text{THEN} \ \text{spec}, \text{where} \ x=d]
\quad \text{obtain} \ n \ \text{aseq}' \ \text{where} \ n-\text{aseq}' \ n \leq \text{length} \ \text{aseq}' \land \ \text{aseq}' \in \text{AS-set} \land \ \text{hd} \ ?ct = \text{lastn} \ n \ \text{aseq}'
\quad \text{unfolding} \ \text{realistic-AS-partial-def}
\quad \text{by} \ (\text{cases} \ \text{execs} \ d, \text{auto})
\quad \text{from} \ \text{this} \ \text{hd-thread-not-empty} \ \text{have} \ n > 0 \ \text{unfolding} \ \text{lastn-def} \ \text{by} \ (\text{cases} \ n, \text{auto})
\quad \text{from} \ \text{this} \ n-\text{aseq}' \ \text{lastn-one-less}[\text{where} \ n=n \ \text{and} \ x=\text{aseq}' \ \text{and} \ a=\text{hd} \ (\text{hd} \ ?ct) \ \text{and} \ y=\text{tl} \ (\text{hd} \ ?ct)] \ \text{hd-thread-not-empty}
\quad \text{have} \ n - 1 \leq \text{length} \ \text{aseq}' \land \ \text{aseq}' \in \text{AS-set} \land \ \text{tl} \ (\text{hd} \ ?ct) = \text{lastn} \ (n - 1) \ \text{aseq}'
\quad \text{by} \ \text{auto}
\quad \text{from} \ \text{this} \ \text{as-defs} \ \text{subset} \ \text{have} \ ?\text{thesis}
\quad \text{unfolding} \ \text{realistic-AS-partial-def}
\quad \text{by} \ \text{auto}
\}

\text{moreover}
\{
    \text{let} \ ?ct = \text{execs} \ (\text{current} \ s)
    \text{let} \ ?\text{execs}' = ?ct
    \text{let} \ ?a' = \text{Some} \ (hd \ (hd \ ?ct))
    \text{assume} \ c-\text{waiting}: \text{let} \ (a, \text{aseqs}', s') = ?c \in \\
    (a, \text{aseqs}') = (?a', ?\text{execs}')
\}

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

The lemma that proves that the total run function is equivalent to the partial run function, i.e., that in this refinement the case of the run function where the precondition is False will never occur.

lemma run-total-equals-run:
assumes realistic-exec: realistic-executions execs
and invariant: invariant s
shows strict-equal (run n (Some s) execs) (run-total n s execs)
proof-
{
  fix n ms s execs
  have strict-equal ms s ∧ realistic-executions-ind execs ∧ precondition-ind s execs —— strict-equal (run n ms execs) (run-total n s execs)
  proof (induct n ms execs arbitrary: s rule: run.induct)
    case (1 s execs sa)
    show ?case by auto
  next
    case (2 n execs s)
    show ?case unfolding strict-equal-def by auto
  next
    case (3 n s execs sa)
    assume interrupt: interrupt (Suc n)
    assume IH: (s ∧ sa. strict-equal (Some (cswitch (Suc n) s)) sa ∧ realistic-executions-ind execs ∧ precondition-ind
sa execs →

strict-equal (run n (Some (cswitch (Suc n) s)) execs) (run-total n sa execs)

\{ 
assume equal-s-sa: strict-equal (Some s) sa
assume realistic: realistic-executions-ind execs
assume inv-sa: precondition-ind sa execs
have inv-nsa: precondition-ind (cswitch (Suc n) sa) execs
proof -
\{ 
fix d
have fst (control (cswitch (Suc n) sa) d (execs d)) \rightarrow AS-precondition (cswitch (Suc n) sa) d
using next-action-after-cswitch inv-sa[unfolded precondition-ind-def, THEN conjunct2, THEN spec where x=d]
precondition-after-cswitch
unfolding Let-def B-def precondition-ind-def
by (cases fst (control (cswitch (Suc n) sa) d (execs d)), auto)
\} 
thus \text{thesis using inv-sa invariant-after-cswitch unfolding precondition-ind-def by auto}
qed
from equal-s-sa realistic inv-nsa inv-sa IH[where sa=cswitch (Suc n) sa]
have equal-ns-nt: strict-equal (run n (Some (cswitch (Suc n) s)) execs) (run-total n (cswitch (Suc n) sa) execs)
unfolding strict-equal-def by (auto)
\}
from this interrupt show ?case by auto
next
case (4 n execs s sa)
assume not-interrupt: ¬interrupt (Suc n)
assume thread-empty: thread-empty(execs (current s))
assume IH: (∀.a. strict-equal (Some s) sa ∧ realistic-executions-ind execs ∧ precondition-ind sa execs \rightarrow strict-equal (run n (Some s) execs) (run-total n sa execs))
have current-s-sa: strict-equal (Some s) sa \rightarrow current s = current sa unfolding strict-equal-def by auto
\{
assume equal-s-sa: strict-equal (Some s) sa
assume realistic: realistic-executions-ind execs
assume inv-sa: precondition-ind sa execs
from equal-s-sa realistic inv-sa IH[where sa=sa]
have equal-ns-nt: strict-equal (run n (Some s) execs) (run-total n sa execs)
unfolding strict-equal-def by (auto)
\}
from this current-s-sa thread-empty not-interrupt show ?case by auto
next
case (5 n execs s sa)
assume not-interrupt: ¬interrupt (Suc n)
assume thread-not-empty: ¬thread-empty(execs (current s))
assume not-prec: ¬precondition (next-state s execs) (next-action s execs)
— In locale ISK, the precondition can be proven to hold at all times. This case cannot happen, and we can prove False.
\{
assume equal-s-sa: strict-equal (Some s) sa
assume realistic: realistic-executions-ind execs
assume inv-sa: precondition-ind sa execs
from equal-s-sa have s-sa: s = sa unfolding strict-equal-def by auto
from inv-sa have
next-action sa execs \rightarrow AS-precondition sa (current sa)
unfolding precondition-ind-def B-def next-action-def
by (cases next-action sa execs, auto)
from this next-state-precondition
have next-action sa execs → AS-precondition (next-state sa execs) (current sa)
unfolding precondition-ind-def B-def
by (cases next-action sa execs,auto)
from inv-sa this s-sa next-state-invariant current-next-state
have prec-s∶precondition (next-state s execs) (next-action s execs)
unfolding precondition-ind-def kprecondition-def precondition-def B-def
by (cases next-action sa execs,auto)
from this not-prec have False by auto
}
thus ?case by auto

next
case (6 n execs s sa)
  assume not-interrupt∶¬interrupt (Suc n)
  assume thread-not-empty∶¬thread-empty(execs (current s))
  assume prec∶precondition (next-state s execs) (next-action s execs)
  assume IH∶(∀sa. strict-equal (Some (step (next-state s execs) (next-action s execs)))) sa ∧
    realistic-executions-ind (next-execs s execs) ∧ precondition-ind sa (next-execs s execs) →
    strict-equal (run n (Some (step (next-state s execs) (next-action s execs))) (next-execs s execs)) (run-total
    n sa (next-execs s execs)))
  have current-s-sa∶strict-equal (Some s) sa → current s = current sa 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) ?ns 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-thac 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
Some realistic — 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 \(\text{thesis}\)

proof \(\{\)
\(\text{cases } ?a = \text{fst } (\text{control } ?\text{ns } d (?\text{na } d))\)
\(\text{assume } \text{snd-action-not-none}: \ ?a \neq \text{None}\)
\(\text{have } \text{AS-precondition } ?\text{ns } d \left(\text{the } ?a\right)\)
\(\text{proof } \{\)
\(\text{case True}\)
\(\text{have } \?\text{thesis}\)
\(\text{proof } \{\)
\(\text{cases } ?\text{na } d = \text{excs } (\text{current } s)\) \(\text{rule: case-split[case-names waiting executing]}\)
\(\text{case executing} — \text{The kernel is executing two consecutive actions } a \text{ and } a’. \text{ We show that } [a, a’] \text{ is a subsequence in some action in AS-set. The PO’s ensure that the precondition is inductive.}\)
\(\text{from executing } \text{True Some control-spec[THEN spec,THEN spec,THEN spec,where } x2=s \text{ and } x1=d \text{ and } x=\text{execs } d]\)
\(\text{have } a\text{-def: } a = \text{hd } (\text{excs } (\text{current } s)) \land ?\text{na } d = (tl (\text{hd } (\text{excs } (\text{current } s)))) \# (tl (\text{execs } (\text{current } s))))\)
\(\text{unfolding } \text{next-action-def next-execs-def Let-def}\)
\(\text{by (auto)}\)
\(\text{from } a\text{-def True } \text{snd-action-not-none control-spec[THEN spec,THEN spec,THEN spec,where } x2=?\text{ns and } x1=d \text{ and } x=?=\text{na } d]\)
\(\text{second-el-is-hd-tl[where } x=\text{hd } (\text{execs } (\text{current } s)) \text{ and } a=\text{hd}(tl(\text{hd } (\text{execs } (\text{current } s)))) \text{ and } x=^=\text{tl}(tl(\text{hd } (\text{execs } (\text{current } s))))]\)
\(\text{have } na\text{-def: the } ?a’ = (\text{hd } (\text{execs } (\text{current } s)))!1\)
\(\text{unfolding } \text{next-execs-def}\)
\(\text{by (auto)}\)
\(\text{from } \text{Some realistic[unfolded realistic-executions-ind-def,THEN spec,where } x=d]\text{ True thread-not-empty}\)
\(\text{obtain } n \text{ aseq’ where witness: } n \leq \text{length aseq’ } \land \text{aseq’ } \in \text{AS-set } \land \text{hd(excs } d) = \text{lastn } n \text{ aseq’}\)
\(\text{unfolding } \text{realistic-AS-partial-def } \text{by } (\text{cases excs } d,\text{auto})\)
\(\text{from } \text{True executing length-It-2-implies-tl-empty[where } x=\text{hd } (\text{execs } (\text{current } s))\]\\(\text{Some control-spec[THEN spec,THEN spec,THEN spec,where } x2=s \text{ and } x1=d \text{ and } x=\text{execs } d]\)
\(\text{snd-action-not-none control-spec[THEN spec,THEN spec,THEN spec,where } x2=\text{?ns and } x1=d \text{ and } x=\text{?na } d]\)
\(\text{have } \text{in-action-sequence: length } (\text{hd } (\text{execs } (\text{current } s)))) \geq 2\)
\(\text{unfolding } \text{next-action-def next-execs-def}\)
\(\text{by auto}\)
\(\text{from this witness consecutive-is-sub-seq[where } a=a \text{ and } b=\text{the } ?a’ \text{ and } n=n \text{ and } y=\text{aseq’ and } x=tl(tl(\text{hd } (\text{execs } (\text{current } s))))]\)
\(\text{a-def na-def True in-action-sequence}\)
\(\text{x-is-hd-snd-tl[where } x=\text{hd } (\text{execs } (\text{current } s))\]
have \( I \; \exists \; a' \in \text{AS-set} \cdot \text{is-sub-seq a (the } a' \text{) aseq}' \)
by (auto)
from True inv-sa [unfolded precondition-ind-def, THEN conjunct2, THEN spec, where \( x = \text{current s} \) s-sa
have 2: AS-precondition s (current s) a
unfolding strict-equal-def next-action-def B-def by auto
from executing True Some control-spec [THEN spec, THEN spec, THEN spec, where \( x_2 = s \) and \( x_1 = d \) and \( x = \text{execs d} \)]
have not-aborting: \( \neg \text{aborting (next-state s execs) (current s) (the } a' \) )
unfolding next-action-def next-state-def next-execs-def
by auto
from executing True Some control-spec [THEN spec, THEN spec, THEN spec, where \( x_2 = s \) and \( x_1 = d \) and \( x = \text{execs d} \)]
have not-waiting: \( \neg \text{waiting (next-state s execs) (current s) (the } a' \) )
unfolding next-action-def next-state-def next-execs-def
by auto
from True this
1 2 inv-s
sub-seq-in-prefixes [where \( X = \text{AS-set} \)] Some next-state-invariant
current-next-state [THEN spec, THEN spec, where \( x_1 = s \) and \( x = \text{execs} \) ]
AS-prec-after-step [THEN spec, THEN spec, THEN spec, where \( x_2 = \text{next-state s execs} \) and \( x_1 = a \) and
\( x = \text{the } a' \) ]
next-state-precondition not-aborting not-waiting
show ?thesis
unfolding step-def
by auto
next
case waiting — The kernel is delaying action a. Thus the action after a, which is \( a' \), is equal to a.
from \( tl-hd-x-not-tl-x \) [where \( X = \text{AS-set} \)] True waiting control-spec [THEN spec, THEN spec, THEN spec, WHERE \( x_2 = s \) and \( x_1 = d \) and \( x = \text{execs d} \)]
Some
have a-def: ?na d = execs (current s) \( \land \) next-state s execs = s \( \land \) 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 \( x_2 = ?ns \) and \( x_1 = d \) 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 \( x = \text{current s} \) 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.

from None True snd-action-not-none control-spec [THEN spec, THEN spec, THEN spec, WHERE \( x_2 = ?ns \) ...
and \(x1=d\) and \(x=\text{?ns}\ d\]
  control-spec[THEN spec,THEN spec,THEN spec,where \(x2=\text{?a}\) and \(x1=d\) and \(x=\text{execs}\ d\]
  have \(\text{na-def}\) the \(?'a'=\text{hd}\ (\text{tl}\ (\text{execs}\ (\text{current}\ s)))\) \& \(?'a\ d=\text{tl}\ (\text{execs}\ (\text{current}\ s))\)
  unfolding next-action-def\ next-exec-def
  by(auto)
  from True None snd-action-not-none control-spec[THEN spec,THEN spec,THEN spec,where \(x2=\text{?ns}\) and \(x1=d\) and \(x=\text{execs}\ d\]
  this
  have 1: \(\text{tl}\ (\text{execs}\ (\text{current}\ s))\) \# [] \& \(\text{hd}\ (\text{tl}\ (\text{execs}\ (\text{current}\ s)))\) \# []
  by auto
  from this realistic unfolding realistic-executions-ind-def,THEN spec,where \(x=d\] True thread-not-empty
  have \(\text{hd}\ (\text{tl}\ (\text{execs}\ (\text{current}\ s)))\) \in AS-set
  by (cases \text{execs}\ d,auto)
  from True snd-action-not-none this
  inv-n\-s\-ns\ this na-def 1
  AS-prec-first-action[THEN spec,THEN spec,THEN spec,where \(x2=\text{?ns}\) and \(x1=\text{current}\ s\) and \(x=\text{?ns}\ (current) s\)]
  and \(x1=d\]
  show \(?\text{thesis}\) by auto
  qed
  }
  thus ?thesis
    using control-spec[THEN spec,THEN spec,THEN spec,where \(x2=\text{?ns}\) and \(x1=\text{current}\ s\) and \(x=\text{?ns}\ (current) s\)]
      thread-not-empty True snd-action-not-none
    by (auto simp add: Let-def)
next
  case False
  from False have equal-na-a: \(?'\text{na}\ d=\text{execs}\ d\)
  unfolding next-exec-def by auto
  from this False current-next-state next-action-after-step
  have \(?'a'=\text{fst}\ (\text{control}\ (\text{next-state}\ s\ \text{execs}\ d)\ (\text{next-exec} s\ \text{execs}\ d))\)
  unfolding next-action-def by auto
  from inv-sa unfolding precondition-ind-def,THEN conjunct2,THEN spec,where \(x=d\] s-sa equal-na-a this
  next-action-after-next-state[THEN spec,THEN spec,THEN spec,where \(x=d\) and \(x2=\text{?ns}\) and \(x1=\text{execs}\]
  snd-action-not-none False
  have \(\text{AS-precondition}\ s\ \text{d}\ (\text{the}?'a')\)
  unfolding precondition-ind-def next-action-def B-def by (cases \text{fst}\ (\text{control}\ s\ a\ d\ (\text{execs}\ d)))auto
  from equal-na-a False this next-state-precondition current-next-state
  AS-prec-dom-independent[THEN spec,THEN spec,THEN spec,THEN spec,where \(x3=\text{next-state}\ s\ \text{execs}\]
  and \(x2=d\) and \(x=\text{the}?'a\) and \(x1=\text{the}?'a')]
  show \(?\text{thesis}\)
    unfolding step-def
    by (cases \text{next-action}\ s\ \text{execs},auto)
  qed
  }
  hence \(\text{fst}\ (\text{control}\ ?\text{ns}\ d\ (?'\text{na}\ d))\rightarrow\text{AS-precondition}\ ?\text{ns}\ d\) unfolding B-def
  by (cases \text{fst}\ (\text{control}\ ?\text{ns}\ d\ (?'\text{na}\ d)),auto)
  }
  thus ?thesis by auto
  qed
  from this inv-n\-s\ show ?thesis
    unfolding precondition-ind-def B-def Let-def
    by (auto)
  qed
  from equal-n\-n\-n\-a realistic-na invariant-na s-sa IH[where \(s=\text{?ns}\]
  have equal-n\-n\-t\ strict-equal (run n) (Some \(?'\text{na}\) (run-total n (\text{step}\ (\text{next-state}\ s\ \text{execs})\ (\text{next-action}\ s\ \text{execs}))\)
    (\text{next-exec}\ s\ \text{execs}))
  }
Theorem unwinding implies _isecure_ gives security for all realistic executions. For unrealistic executions, it holds vacuously and therefore does not tell us anything. In order to prove security for this refinement (i.e., for function _run_total_), we have to prove that purging yields realistic runs.

**Lemma** realistic-purge:

\[ \forall \text{ execs d} \cdot \text{realistic-executions execs} \rightarrow \text{realistic-executions (purge execs d)} \]
proof-
{ 
  fix execs d 
  assume realistic-executions execs 
  hence realistic-executions (purge execs d) 
  using someI[where \( P = \text{realistic-execution} \) and \( x = \text{execs} \ d \) ] 
  unfolding realistic-executions-def purge-def by(simp) 
}
thus \( \exists \)thesis by auto
qed

lemma remove-gateway-comm-subset: 
shows set (remove-gateway-communications d exec) \( \subseteq \) set exec \( \cup \) {[]} 
by(induct exec.auto)

lemma realistic-ipurge-l: 
shows \( \forall \) execs d . realistic-executions execs \( \rightarrow \) 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 \( \text{exec} = \text{execs} \ d \) ] 1 have realistic-executions (ipurge-l execs d) 
  unfolding realistic-execution-def realistic-executions-def ipurge-l-def by(auto) 
}
thus \( \exists \)thesis by auto
qed

lemma realistic-ipurge-r: 
shows \( \forall \) execs d . realistic-executions execs \( \rightarrow \) 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 \( \text{exec} = \text{execs} \ d \) ] 1 have realistic-executions (ipurge-r execs d) 
  using someI[where \( \lambda \ x . \text{realistic-execution} \ x \) and \( x = \text{execs} \ d \) ] 
  unfolding realistic-execution-def realistic-executions-def ipurge-r-def by(auto) 
}
thus \( \exists \)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 
  \( \equiv \forall \) execs a n . realistic-executions execs \( \rightarrow \) 
  (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 
  \land current s-f = current (run-total n s0 (purge execs (current s-f))))

definition NI-indirect-sources-total:bool 
where NI-indirect-sources-total 
  \( \equiv \forall \) execs a n . realistic-executions execs \( \rightarrow \) 
  (let s-f = run-total n s0 execs in 
  output-f (run-total n s0 (ipurge-l execs (current s-f))) a =
output-f \((\text{run-total } n \ s0 \ (\text{ipurge-r execs } (\text{current s-f})))\) \(a\)

**definition** isecure-total :\: bool

**where** isecure-total ≡ NI-unrelated-total \(\land\) NI-indirect-sources-total

**theorem** unwinding-implies-secure-total:

**shows** isecure-total

**proof**

\begin{itemize}
\item from **unwinding-implies-iseecure** have secure-partial: NI-unrelated unfolding isecure-def by blast
\item from **unwinding-implies-iseecure** have isecure1-partial: NI-indirect-sources unfolding isecure-def by blast
\end{itemize}

have NI-unrelated-total: NI-unrelated-total

**proof**

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

have NI-indirect-sources-total: NI-indirect-sources-total

**proof**

\begin{itemize}
\item \{ fix execs a n  
\item assume realistic: realistic-executions execs  
\item from invariant-s0 realistic run-total-equals-run[where \(n=n\) and \(s=s0\) and \(execs=execs\)]  
\item have 1: strict-equal \((\text{run } n \ (\text{Some s0}) \ \text{execs}) \ (\text{run-total } n \ s0 \ \text{execs})\) by auto
\item have \(\text{let } s-f = \text{run-total } n \ s0 \ \text{execs in output-f } s-f \ a = \text{output-f } (\text{run-total } n \ s0 \ \text{ipurge-l execs } (\text{current s-f}))) \ a = \text{output-f } (\text{run-total } n \ s0 \ \text{ipurge-r execs } (\text{current s-f}))) \ a\)
\item proof (cases \(\text{run } n \ (\text{Some s0}) \ \text{execs}\))
case None
  thus ?thesis unfolding NI-unrelated-total-def strict-equal-def by auto
next
case (Some s-f)
  from realistic-ipurge-l invariant-s0 realistic run-total-equals-run
    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 invariant-s0 realistic run-total-equals-run
    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 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 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-l
    have \text {isecure}_1-partial[unfolded NI-indirect-sources-def, THEN spec, THEN spec, THEN spec, where}

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

and aborting :: 'state-t \Rightarrow 'dom-t \Rightarrow 'action-t \Rightarrow bool — Returns true if the action is aborted.
and waiting :: 'state-t \Rightarrow 'dom-t \Rightarrow 'action-t \Rightarrow bool — Returns true if execution of the given action is delayed.
and set-error-code :: 'state-t \Rightarrow 'action-t \Rightarrow 'state-t — Sets an error code when actions are aborted.

assumes vpeq-transitive :: \forall \ a \ b \ u. (vpeq u a b \land vpeq u b c) \rightarrow vpeq u a c
and vpeq-symmetric :: \forall \ a \ b. vpeq u a b \rightarrow vpeq u b a
and vpeq-reflexive :: \forall \ a. vpeq u a a
and ifp-reflexive :: \forall \ a. ifp u a
and weakly-step-consistent :: \forall \ s \ t \ a. vpeq u s t \land vpeq (current s) s t \land invariant s \land AS-precondition s (current s) \land invariant t \land AS-precondition t (current t) \land current s = current t \rightarrow vpeq u (kstep s a) (kstep t a)
and locally-respects :: \forall \ a \ s. \neg ifp (current s) u \land invariant s \land AS-precondition s (current s) \land current s = current t \rightarrow current (cswitch n s) = current (cswitch n t)

and cswitch-consistency :: \forall \ u s t n. vpeq u s t \rightarrow vpeq u (cswitch n s) (cswitch n t)
and empty-in-AS-set :: [] \in \ AS-set
and invariant-s0 invariant s0
and invariant-after-cswitch :: \forall \ s \ n. \neg invariant s \rightarrow invariant (cswitch n s)
and precondition-after-cswitch :: \forall \ s \ d \ n a. \ AS-precondition s d a \rightarrow AS-precondition (cswitch n s) d a
and AS-prec-first-action :: \forall \ s \ d \ aseq. \neg invariant s \land aseq \in AS-set \land aseq \notin [] \rightarrow AS-precondition s d (hd aseq)
and AS-prec-after-step :: \forall \ s a a'. (\exists \ bseq \in AS-set. \ aseq = sub-seq a a' bseq) \land invariant s \land AS-precondition s (current s) \land \neg aborting s (current s) \land \neg waiting s (current s) a \land AS-precondition (kstep s a) (current s) a'
and AS-prec-dom-independent :: \forall \ s \ d \ a a'. current s = d \land AS-precondition s d a \rightarrow AS-precondition (kstep s a') d a
and spec-of-invariant :: \forall \ a \ s. invariant s \rightarrow invariant (kstep s a)
and aborting-switch-independent :: \forall \ n \ s \ . aborting (cswitch n s) = aborting s
and aborting-error-update :: \forall \ s \ d a a'. current s \notin d \land aborting s d a \rightarrow aborting (set-error-code s a') d a
and aborting-after-step :: \forall \ s \ a \ d. current s \notin d \land aborting (kstep s a) d = aborting s d
and aborting-consistent :: \forall \ s \ t \ u. vpeq u s t \land aborting s u \land aborting t u
and waiting-switch-independent :: \forall \ a \ n \ s. waiting (cswitch n s) = waiting s
and waiting-error-update :: \forall \ s \ d a a'. current s \notin d \land waiting s d a \rightarrow waiting (set-error-code s a') d a
and waiting-consistent :: \forall \ s \ t \ u a. vpeq (current s) s t \land (\forall \ d \in kinvolved a . vpeq d s t) \land vpeq u s t \rightarrow waiting s u a = waiting t u a
and spec-of-waiting :: \forall \ a \ s. waiting s (current s) a \rightarrow kstep s a = s
and set-error-consistent :: \forall \ s \ t \ u a . vpeq u s t \rightarrow vpeq u (set-error-code s a) (set-error-code t a)
and set-error-locally-respects :: \forall \ a \ u . \neg ifp (current s) a \rightarrow vpeq u s (set-error-code s a)
and current-set-error-code :: \forall \ a \ s. current (set-error-code s a) = current s
and precondition-after-set-error-code :: \forall \ s \ d \ a a'. AS-precondition s d a \land aborting s (current s) a' \rightarrow AS-precondition (set-error-code s a') d a
and invariant-after-set-error-code :: \forall \ a . \neg invariant (set-error-code s a)
and involved-ifp :: \forall \ a \ . \neg d \in (kinvolved a) . AS-precondition s (current s) a \rightarrow ifp d (current s)

begin
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.

```agda
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
```

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].

```agda
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.

```agda
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.

```agda
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-execs s execs in
  case control-a of None ⇒ run n control-s control-x
  | (Some a) ⇒ run n (kstep control-s a) control-x)
using not0-implies-Suc by (metis prod-cases3.auto)
termination by lexicographic-order
```

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 ≡ ∀ 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 ≡ ∀ 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 idfference 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 ∀ 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: ∀ 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-execs-consistent:
  shows ∀ s t execs . vpeq (current s) s t ∧ (∀ d ∈ involved (next-action s execs) . vpeq d s t) ∧ current s = current t → next-exec s execs = next-exec t execs
  proof-
  { fix s t execs
    assume vpeq: vpeq (current s) s t
    assume vpeq-involved: ∀ d ∈ involved (next-exec 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-exec s execs = next-exec t execs
    unfolding Kernel.next-exec-def
    by(cases (s,(current s),execs (current s)) rule: CISK-control.cases,auto)
  }
  thus ?thesis by auto
  qed
t → fst (snd (CISK-control s (current s) (execs (current s)))) = fst (snd (CISK-control t (current s) (execs (current s))))
proof-
{
  fix s t execs
  assume vpeq: vpeq (current s) s t
  assume vpeq-involved: ∀ 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 1: aborting t (current s) = aborting s (current s) by auto
  from 1 vpeq current-s-t vpeq-involved waiting-consistent
    THEN spec, 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 fst (snd (CISK-control s (current s) (execs (current s)))) = fst (snd (CISK-control t (current s) (execs (current s))))
    unfolding Kernel.next-action-def Kernel.involved-def
    by (cases (s,(current s),execs (current s)) rule: CISK-control.cases,auto split:if-split-asm)
} thus ?thesis by auto
qed

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

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

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

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

lemma next-action-after-cswitch:
shows ∀ s n d aseqs . fst (CISK-control (cswitch n s) d aseqs) = fst (CISK-control s d aseqs)
proof--
{ 
  fix s n d aseqs
  have 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 d . 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 1
by(cases (sa,current sa,execs (current sa)) rule: CISK-control.cases,auto split: if-split-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 a d 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
{
fix a
assume 1: next-action s execs = Some a
\[ \textbf{proof} \]
\[
\begin{align*}
&\text{shows} \forall \text{execs } u. \text{actions-in-execution}(\text{execs } u) \subseteq \text{actions-in-execution}(\text{execs } u) \\
&\text{shows} \forall a b c u. \text{vpeq}(a, b, c) \Rightarrow \text{vpeq}(u, a, b) \rightarrow \text{vpeq}(u, a, c) \\
&\text{using} \text{vpeq-transitive} \text{ by blast} \\
&\text{show} \forall a b u. \text{vpeq}(a, b) \Rightarrow \text{vpeq}(u, a, b) \\
&\text{using} \text{vpeq-symmetric} \text{ by blast} \\
&\text{show} \forall a u. \text{vpeq}(a, u) \\
&\text{using} \text{vpeq-reflexive} \text{ by blast} \\
&\text{show} \forall u. \text{ifp}(u, u) \\
&\text{using} \text{ifp-reflexive} \text{ by blast} \\
&\text{show} \forall s t a. \text{vpeq}(s, t) \land \text{vpeq}(current s, t) \land \text{kprecondition}(s, t) \land \text{kprecondition}(a, t) \land current s = current t \\
&\Rightarrow \text{vpeq}(u, (\text{kstep } s, a)) (\text{kstep } t, a) \\
&\text{using} \text{weakly-step-consistent} \text{ by blast} \\
&\text{show} \forall a s u. \neg \text{ifp}(current s, u) \land \text{kprecondition}(s, a) \Rightarrow \text{vpeq}(u, s) (\text{kstep } s, a) \\
&\text{using} \text{locally-respects} \text{ by blast} \\
&\text{show} \forall s t a. \text{current}(\text{kstep } s, a) = current s \\
&\text{using} \text{step-atomicity} \text{ by blast} \\
&\text{show} \forall n s t. \text{current } s = current t \Rightarrow \text{current}(\text{cswitch } n, s) = current(\text{cswitch } n, t) \\
&\text{using} \text{cswitch-independent-of-state} \text{ by blast} \\
&\text{show} \forall u s t n. \text{vpeq}(u, s, t) \rightarrow \text{vpeq}(u, (\text{cswitch } n, s)) (\text{cswitch } n, t) \\
&\text{using} \text{cswitch-consistency} \text{ by blast} \\
&\text{show} \forall s t. \text{execs } . \text{vpeq}(current s, s) \land (\forall d \in \text{involved}(next-action s \text{ execs })) . \text{vpeq}(d, s) \land current s = current t \\
&\Rightarrow next-action s \text{ execs } = next-action t \text{ execs} \\
&\text{using} \text{next-action-consistent} \text{ by blast} \\
&\text{show} \forall s t. \text{execs } . \text{vpeq}(current s, s) \land (\forall d \in \text{involved}(next-action s \text{ execs })) . \text{vpeq}(d, s) \land current s = 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)))) \\
\end{align*}
\]
\textbf{QED}
(current s)))
  using next-exec-consistent by blast
  show ∀ s t u execs. vpeq (current s) s t ∧ vpeq u s t ∧ current s = current t ⨸ vpeq u (next-state s execs)
  (next-state t execs)
  using next-state-consistent by auto
  show ∀ s execs. current (next-state s execs) = current s
  using current-next-state by auto
  show ∀ s u execs. ¬ ifp (current s) u ⨸ vpeq u s (next-state s execs)
  using locally-respects-next-state by auto
  show [] ∈ AS-set
  using empty-in-AS-set by blast
  show ∀ s n a. invariant s ⨸ invariant (cswitch n s)
  using invariant-after-cswitch by blast
  show ∀ s a d n a. AS-precondition s d a ⨸ AS-precondition (cswitch n s) d a
  using pre-condition-after-cswitch by blast
  show invariant s0
  using invariant-s0 by blast
  show ∀ s d a seq . invariant s ∧ aseq ∈ AS-set ∧ aseq ≠ [] ⨸ AS-precondition s d (hd aseq)
  using AS-pre-first-action by blast
  show ∀ s a a' . (∃ aseq: AS-set. is-sub-seq a a' aseq) ∧ invariant s ∧ AS-precondition s (current s) a ∧ ¬ aborting s (current s) a ∧ ¬ waiting s (current s) a ⨸ AS-precondition (kstep s a) (current s) a'
  using AS-pre-after-step by blast
  show ∀ s d a a' . current s ≠ d ∧ AS-precondition s d a ⨸ AS-precondition (kstep s a') d a
  using AS-prec-dom-independent by blast
  show ∀ s a . invariant s ⨸ invariant (kstep s a)
  using spec-of-invariant by blast
  show ∀ s a . kprecondition s a ≡ kprecondition s a by auto
  show ∀ aseq . realistic-execution aseq ≡ set aseq ∈ AS-set
  unfolding realistic-execution-def by auto
  show ∀ s a . ∀ d ∈ involved a. kprecondition s (the a) ⨸ ifp d (current s)
  using involved-if unfolding Kernel.involved-def by (auto split: option.splits)
  show ∀ s execs. next-action s execs ⨸ (∀ a . a ∈ actions-in-execution (execs (current s)))
  using next-action-from-exec by blast
  show ∀ s execs u . actions-in-execution (next-execs s execs u) ⊆ actions-in-execution (execs u)
  using next-exec-subset by blast
  show ∀ s d aseqs.
  case CISK-control s d aseqs of
  (a, aseqs', s') ⇒
  thread-empty aseqs ∧ (a, aseqs') = (None, []) ⨸
  aseqs ≠ [] ∧ hd aseqs ≠ [] ∧ ¬ aborting s' d (the a) ∧ ¬ waiting s' d (the a) ∧ (a, aseqs') = (Some (hd (hd aseqs))), tl (hd aseqs) ≠ tl aseqs) ∨
  aseqs ≠ [] ∧ hd aseqs ≠ [] ∧ waiting s' d (the a) ∧ (a, aseqs', s') = (Some (hd (hd aseqs))), aseqs, s) ∨ (a, aseqs') = (None, tl aseqs)
  using CISK-control-spec by blast
  show ∀ s n d aseqs. fst (CISK-control (cswitch n s) d aseqs) = fst (CISK-control s d aseqs)
  using next-action-after-cswitch by auto
  show ∀ s execs d.
  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))
  using next-action-after-step by auto
  show ∀ s d a execs. next-state s d a ⨸ next-precondition (next-state s execs) d a
using next-state-precondition by auto
show \( \forall s \text{ execs}, \text{ invariant } s \rightarrow \text{ invariant } (\text{next-state } s \text{ execs}) \)
using next-state-invariant by auto
show \( \forall s \text{ a, waiting } s \text{ (current } s) \text{ a } \rightarrow kstep s \text{ a } = s \)
using spec-of-waiting by blast
qed

note interpreted = int Interruptible-Separation-Kernel-axioms
note run-total-induct = Interruptible-Separation-Kernel.run-total.induct\[s \text{ o current } cswitch interrupt kprecondition realistic-execution\] 

CISK-control kinvolved ifp vpeq AS-set invariant AS-precondition

aborting waiting - interrupt

have run-equals-run-total:
\( \land \ n \text{ s execs } . \text{ run } n \text{ s execs } \equiv \text{ Interruptible-Separation-Kernel.run-total kstep current cswitch interrupt CISK-control} n \text{ s execs} \)
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 \( \emptyset \) 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 \( \emptyset \) 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 \( \emptyset \) 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 \( ?\text{thesis unfolding isecure-def by auto} \)
qed

end
end

4 Instantiation by a separation kernel with concrete actions

theory Step-configuration
imports Main
begin

In the previous section, no concrete actions for the step function were given. The foremost point we want to make by this instantiation is to show that we can instantiate the CISK model of the previous section with an implementation that, for the step function, as actions, provides events and interprocess communication (IPC). System call invocations that can be interrupted at certain interrupt points are split into several atomic steps. A communication interface of events and IPC is less “trivial” than it may seem at a first glance, for example the L4 microkernel API only provided IPC as communication primitive [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_subset. A use case for this is when you have statically
configured access to files by subjects, but whether a file can be read/written also depends on whether the
file has been dynamically opened or not. The instantiation provides infrastructure for such an invariant
on the relation of a dynamic policy to a static policy, and shows how the invariant is maintained, without
modeling any API for an open/close operation.

4.1 Model of a separation kernel configuration

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.

typedec partition-id-t
typedec thread-id-t
typedec page-t — physical address of a memory page
typedec 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.

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

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

c consts
  partition :: thread-id-t ⇒ 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 ifp relation. We also express a static subject-subject sp-spec-subj-obj and subject-object sp-spec-subj-subj access control policy for the implementation of the model. The following locale summarizes all properties we need.

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

assumes sp-spec-file-provider: ∀ p1 p2 f m1 m2 .
  sp-spec-subj-obj p1 (FILEP f) m1 ∧
  sp-spec-subj-obj p2 (FILEP f) m2 → sp-spec-subj-subj p1 p2

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

and ifp-reflexive:
  ∀ p . ifp p p

and ifp-compatible-with-sp-spec:
  ∀ a b . sp-spec-subj-subj a b → ifp a b ∧ ifp b a

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

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 configuration-subj-obj :: 'a ⇒ obj-id-t ⇒ mode-t ⇒ bool
begin

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

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

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

show sp-spec-file-provider:

∀ p1 p2 f m1 m2 .

sp-spec-subj-obj p1 (FILEP f) m1 ∧

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

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

show sp-spec-no-wronly-pages:

∀ p x . sp-spec-subj-obj p (PAGE x) WRITE → sp-spec-subj-obj p (PAGE x) READ

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

show ifp-reflexive:

∀ p . ifp p p

unfolding ifp-def by auto

show ifp-compatible-with-sp-spec:

∀ a b . sp-spec-subj-subj a b → ifp a b ∧ ifp b a

unfolding ifp-def by auto

show ifp-compatible-with-ipc:

∀ a b c x . (sp-spec-subj-subj a b ∧ sp-spec-subj-obj b (PAGE x) WRITE ∧ sp-spec-subj-obj c (PAGE x) READ) → ifp a c

unfolding ifp-def by auto

qed

end

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

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

**interpretation** Policy: abstract-policy-derivation configured-subj-obj.

**interpretation** Policy-properties: policy-axioms Policy.sp-spec-subj-obj Policy.sp-spec-subj-subj Policy.ifp

using Policy.correct by auto

**lemma** example-how-to-use-properties-in-proofs:

shows ∀ p . Policy.ifp p p

using Policy-properties.ifp-reflexive by auto

end

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 | RECV

**datatype** ipc-stage-t = PREP | WAIT | BUF page-t
**datatype** ev-consume-t = EV-CONSUME-ALL | EV-CONSUME-ONE
**datatype** ev-wait-stage-t = EV-PREP | EV-WAIT | EV-FINISH
**datatype** ev-signal-stage-t = EV-SIGNAL-PREP | EV-SIGNAL-FINISH

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

### 4.3.2 System state

typedec obj-t — value of an object

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

**consts**

| partition :: thread-id-t ⇒ partition-id-t |

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

**record** thread-t =

| ev-counter = nat — event counter |

**record** state-t =

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

Later (Section 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 := \lambda q q’ q’’ . (part-id = q ∧ obj-id = q’ ∧ m = q’’) ) |
| \sp-impl-subj-obj s q q’ q’’ |

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

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

| add-comm-right p p’ s ≡ |
| s ( \sp-impl-subj-subj := \lambda 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.

**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**.

**4.4 Preconditions and invariants for the atomic step**

**theory** Step-invariants  
**imports** Step  
**begin**  

The dynamic/implementation policies have to be compatible with the static configuration.
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
    unfolding atomic-step-invariant-def atomic-step-precondition-def ipc-precondition-def
    sp-subset-def set-object-value-def Let-def
  by (simp split: 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
    unfolding atomic-step-invariant-def atomic-step-precondition-def ipc-precondition-def
    sp-subset-def set-object-value-def Let-def
  by (simp split: int-point-t.splits ipc-stage-t.splits ipc-direction-t.splits)
qed

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

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

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

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

4.4.2 Summary theorems on atomic step invariants

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

theorem atomic-step-preserves-invariants:
  fixes s :: state-t
  and tid :: thread-id-t
  assumes atomic-step-invariant s
  shows atomic-step-invariant (atomic-step s a)
  proof (cases a)
    case SK-IPC
    then show ?thesis unfolding atomic-step-def
    using assms atomic-ipc-preserves-invariants
    by simp
    next case (SK-EV-WAIT ev-wait-stage consume)
    then show ?thesis
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 csswitch-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: 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-vpeq
  imports Step Step-invariants
begin
  The view consists of
  1. View of object values.
2. View of subject-subject dynamic policy. The threads can discover the policy at runtime, e.g. by calling ipc() and observing success or failure.

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

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

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

definition vpeq-subj-obj :∶∶ partition-id-t ⇒ state-t ⇒ state-t ⇒ bool where
  vpeq-subj-obj u s t ≡ ∀ ob m1 . (Policy.sp-spec-subj-obj u ob m → sp-impl-subj-obj s u ob m = sp-impl-subj-obj t u ob m) \\ ∧ (Policy.sp-spec-subj-obj p1 ob PROVIDE ∧ (Policy.sp-spec-subj-obj u ob READ ∨ Policy.sp-spec-subj-obj u ob WRITE) → sp-impl-subj-obj s p1 ob PROVIDE = sp-impl-subj-obj t p1 ob PROVIDE)

definition vpeq-local :∶∶ partition-id-t ⇒ state-t ⇒ state-t ⇒ bool where
  vpeq-local u s t ≡ ∀ tid . (partition tid) = u → (thread s tid) = (thread t tid)

definition vpeq :∶∶ partition-id-t ⇒ state-t ⇒ state-t where
  vpeq u s t ≡ vpeq-obj u s t ∧ vpeq-subj-subj u s t ∧ vpeq-subj-obj u s t ∧ vpeq-local u s t

4.5.1 Elementary properties

lemma vpeq-rel:
  shows vpeq-refl : vpeq u s s
  and vpeq-sym [sym] : vpeq u s t → vpeq u t s
  and vpeq-trans [trans]: [[ vpeq u s1 s2 ; vpeq u s2 s3 ]] → vpeq u s1 s3
unfolding vpeq-def vpeq-obj-def vpeq-subj-subj-def vpeq-subj-obj-def vpeq-local-def
by auto

Auxiliary equivalence relation.

lemma set-object-value-ign:
  assumes eq-obs: " Policy.sp-spec-subj-obj u x READ
  shows vpeq u s (set-object-value x y s)
proof −
  from assms show ?thesis
  unfolding vpeq-def vpeq-obj-def vpeq-subj-subj-def vpeq-subj-obj-def vpeq-local-def
  by auto
qed

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

theorem cswitch-consistency-and-respect:
  fixes u :: partition-id-t
  and s :: state-t
  and 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 no:~ Policy.ifp (partition tid) u
  and inv: atomic-step-invariant s
  and prec: atomic-step-precondition s tid (SK-IPC dir stage partner pag)
  and ipt-case: ipt = SK-IPC dir stage partner page
  shows vpeq u s (atomic-step-ipc tid dir stage partner page s)
proof(cases stage)
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 ~ Policy.sp-spec-subj-obj u (PAGE mypage) READ
  using no Policy-properties.ifp-compatible-with-ipc
  by auto
  thus \(?\)thesis
  using BUF SEND assms
lemma ev-signal-respects-policy:
  assumes no: ¬Policy.ifp (partition tid) u
  and inv: atomic-step-invariant s
  and prec: atomic-step-precondition s tid (SK-EV-SIGNAL EV-SIGNAL-FINISH partner)
  and ipt-case: ipt = SK-EV-SIGNAL EV-SIGNAL-FINISH partner
  shows vpeq u s (atomic-step-ev-signal tid partner s)
proof -
  from assms have 1: (partition partner) ≠ u
  unfolding atomic-step-precondition-def ev-signal-precondition-def
  by 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-subj u s (atomic-step-ev-signal tid partner s)
  unfolding vpeq-subj-subj-def atomic-step-ev-signal-def
  by simp
  have 5: vpeq-subj-obj u s (atomic-step-ev-signal tid partner s)
  unfolding vpeq-subj-obj-def atomic-step-ev-signal-def
  by simp
  with 2 3 4 5 show thesis
  unfolding vpeq-def
  by simp
qed

lemma ev-wait-all-respects-policy:
  assumes no: ¬Policy.ifp (partition tid) u
  and inv: atomic-step-invariant s
  and ipt-case: ipt = SK-EV-WAIT ev-wait-stage EV-CONSUME-ALL
  and ipt-case: ipt = SK-EV-WAIT ev-wait-stage EV-CONSUME-ALL
  and ipt-case: ipt = SK-EV-WAIT ev-wait-stage EV-CONSUME-ALL
  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 1: (partition tid) ≠ 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
lemma ev-wait-one-respects-policy:
assumes noc ː ¬ Policy.ifp (partition tid) u
and invː atomic-step-invariant s
and precː atomic-step-precondition s tid ipt
and ipt-caseː ipt = SK-EV-WAIT ev-wait-stage EV-CONSUME-ONE
shows vpeq u s (atomic-step-ev-wait-one tid s)
proof −
from assms have 1ː (partition tid) ʃ u
unfolding Policy.ifp-def
by simp
then have 2ː vpeq-local u s (atomic-step-ev-wait-one tid s)
unfolding vpeq-local-def atomic-step-ev-wait-one-def
by simp
have 3ː vpeq-obj u s (atomic-step-ev-wait-one tid s)
unfolding vpeq-obj-def atomic-step-ev-wait-one-def
by simp
have 4ː 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 5ː 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 noc ː ¬ Policy.ifp (partition (current s)) u
and invː atomic-step-invariant s
and precː 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 splitː int-point-t.splits ev-consume-t.splits ev-wait-stage-t.splits ev-signal-stage-t.splits)
qed

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** \( \text{ipc-precondition-weakly-step-consistent} \):

- \text{assumes} \( \text{eq-tid} : \text{vpeq (partition tid)} \ s1 \ s2 \)
  - and \( \text{inv1} : \text{atomic-step-invariant} \ s1 \)
  - and \( \text{inv2} : \text{atomic-step-invariant} \ s2 \)

- \text{shows} \( \text{ipc-precondition tid dir partner page s1} = \text{ipc-precondition tid dir partner page s2} \)

**Proof**

- let \( ?\text{sender} = \text{case dir of SEND} \Rightarrow \text{tid} | \text{RECV} \Rightarrow \text{partner} \)
- let \( ?\text{receiver} = \text{case dir of SEND} \Rightarrow \text{partner} | \text{RECV} \Rightarrow \text{tid} \)
- let \( ?A = \text{sp-impl-subj-subj s1 (partition ?sender) (partition ?receiver)} \)
  = \( \text{sp-impl-subj-subj s2 (partition ?sender) (partition ?receiver)} \)
- let \( ?B = \text{sp-impl-subj-obj s1 (partition tid) (PAGE page) ?local-access-mode} \)
  = \( \text{sp-impl-subj-obj s2 (partition tid) (PAGE page) ?local-access-mode} \)

\( ?A \) : \( \text{True} \)

- \text{using} \( \text{eq-tid unfolding vpeq-def vpeq-subj-subj-def} \)
  \text{by} \( \text{(simp split: ipc-direction-t.splits)} \)

\( ?B \) : \( \text{True} \)

- \text{using} \( \text{eq-tid unfolding vpeq-def vpeq-subj-obj-def} \)
  \text{by} \( \text{(simp split: ipc-direction-t.splits)} \)

\( ?A \) : \( \text{False} \)

- \text{have} \( ?\text{subset s1} \text{ and } ?\text{subset s2} \)
  \text{using} \( \text{inv1 inv2 unfolding atomic-step-invariant-def sp-subset-def by auto} \)
  \text{hence} \( \text{¬ sp-impl-subj-subj s1 (partition ?sender) (partition ?receiver)} \)
  \( \text{and} \ (\text{¬ sp-impl-subj-subj s2 (partition ?sender) (partition ?receiver)} \)
  \text{using} \( \text{False unfolding sp-subset-def by auto} \)
  \text{thus} \( ?A \text{ by auto} \)

\( ?B \) : \( \text{False} \)

- \text{have} \( ?\text{subset s1} \text{ and } ?\text{subset s2} \)
  \text{using} \( \text{inv1 inv2 unfolding atomic-step-invariant-def sp-subset-def by auto} \)
  \text{hence} \( \text{¬ sp-impl-subj-obj s1 (partition tid) (PAGE page) ?local-access-mode} \)
  \( \text{and} \ (\text{¬ sp-impl-subj-obj s2 (partition tid) (PAGE page) ?local-access-mode} \)
  \text{using} \( \text{False unfolding sp-subset-def by auto} \)
  \text{thus} \( ?B \text{ by auto} \)

\( ?\text{thesis} \text{ using } A \text{ B unfolding ipc-precondition-def by auto} \)

**Qed**

**Lemma** \( \text{ev-signal-precondition-weakly-step-consistent} \):

- \text{assumes} \( \text{eq-tid} : \text{vpeq (partition tid)} s1 s2 \)
  - and \( \text{inv1} : \text{atomic-step-invariant} s1 \)
  - and \( \text{inv2} : \text{atomic-step-invariant} s2 \)

- \text{shows} \( \text{ev-signal-precondition tid partner s1} = \text{ev-signal-precondition tid partner s2} \)

**Proof**
let \( ?A = \text{sp-impl-subj-subj } s1 \text{ (partition } tid) \text{ (partition partner)} = \text{sp-impl-subj-subj } s2 \text{ (partition } tid) \text{ (partition partner)} \)

have \( ?A \)

proof (cases Policy.\text{sp-spec-subj-subj} \text{ (partition } tid) \text{ (partition partner)})

  case True
  
  thus \( ?A \)
  
  using eq-tid unfolding vpeq-def vpeq-subj-subj-def
  
  by (simp split: 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 \( \neg \text{sp-impl-subj-subj } s1 \text{ (partition } tid) \text{ (partition partner)} \)

  and \( \neg \text{sp-impl-subj-subj } s2 \text{ (partition } tid) \text{ (partition partner)} \)

  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 \text{(set-object-value } x y s1) \text{(set-object-value } x y s2)

proof –

let \( ?s1' = \text{set-object-value } x y s1 \) and \( ?s2' = \text{set-object-value } x y s2 \)

have \( E1: \text{vpeq-obj u } ?s1' ?s2' \)

proof –

\{ fix \( x' \)

assume \( 1: \text{Policy.sp-spec-subj-obj u } x' \text{ READ} \)

have \( \text{obj } ?s1' x' = \text{obj } ?s2' x' \)\)

proof (cases \( x = x' \))

  case True

  thus \( \text{obj } ?s1' x' = \text{obj } ?s2' x' \) unfolding set-object-value-def by auto

  next case False

  hence \( 2: \text{obj } ?s1' x' = \text{obj } s1 x' \)

  and \( 3: \text{obj } ?s2' x' = \text{obj } s2 x' \)

  unfolding set-object-value-def by auto

  have \( 4: \text{obj } s1 x' = \text{obj } s2 x' \)

  using 1 eq-obs unfolding vpeq-def vpeq-obj-def by auto

  from 2 3 4 show \( \text{obj } ?s1' x' = \text{obj } ?s2' x' \)

  by simp

qed \}\n
thus \( \text{vpeq-obj u } ?s1' ?s2' \) unfolding vpeq-obj-def by auto

qed

have \( E4: \text{vpeq-subj-obj u } ?s1' ?s2' \)

proof –

have \( \text{sp-impl-subj-subj } ?s1' = \text{sp-impl-subj-subj } s1 \)

and \( \text{sp-impl-subj-subj } ?s2' = \text{sp-impl-subj-subj } s2 \)

unfolding set-object-value-def by auto

thus \( \text{vpeq-subj-subj u } ?s1' ?s2' \)

using eq-obs unfolding vpeq-def vpeq-subj-subj-def by auto

qed

have \( E5: \text{vpeq-subj-obj u } ?s1' ?s2' \)

proof –

have \( \text{sp-impl-subj-obj } ?s1' = \text{sp-impl-subj-obj } s1 \)

and \( \text{sp-impl-subj-obj } ?s2' = \text{sp-impl-subj-obj } s2 \)

unfolding set-object-value-def by auto

thus \( \text{vpeq-subj-obj u } ?s1' ?s2' \)

}
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 /uni22C0
   mypage. ∃[ dir = SEND; stage = BUF mypage ] ⇒ ?thesis
proof -
fix mypage
assume dir-send: dir = SEND
assume stage-buf: stage = 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 ?thesis
using eq-obs unfolding atomic-step-ipc-def
by (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
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-one tid s1)
(atomic-step-ev-wait-one tid s2)

using assms

unfolding vpeq-def vpeq-subj-subj-def vpeq-obj-def vpeq-subj-obj-def vpeq-local-def
atomic-step-ev-wait-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 f s ≡ s (sp-impl-subj-subj := extend-f 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
hence ∀ v . Policy.sp-spec-subj-subj u v → extend-f f ?g1 u v = extend-f f ?g2 u v
and ∀ v . Policy.sp-spec-subj-subj v u → extend-f f ?g1 v u = extend-f f ?g2 v u
unfolding extend-f-def by auto
hence 1: vpeq-subj-subj u (extend-subj-subj f s1) (extend-subj-subj f s2)
unfolding vpeq-subj-subj-def extend-subj-subj-def
by auto
have 2: vpeq-obj u (extend-subj-subj f s1) (extend-subj-subj f s2)
using assms unfolding vpeq-def vpeq-obj-def extend-subj-subj-def by fastforce
have 3: vpeq-subj-obj u (extend-subj-subj f s1) (extend-subj-subj f s2)
using assms unfolding vpeq-def vpeq-subj-obj-def extend-subj-subj-def by fastforce
have 4: vpeq-local u (extend-subj-subj f s1) (extend-subj-subj f s2)
from 1 2 3 4 show ?thesis
using assms unfolding vpeq-def by fast
qed

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 (Step.partition tid).

corollary atomic-step-weakly-step-consistent:
assumes eq-obs: vpeq u s1 s2
and eq-act: vpeq (partition (current s1)) 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 s2 (current s2) ipt
and eq-curr: current s1 = current s2
shows vpeq u (atomic-step s1 ipt) (atomic-step s2 ipt)
proof
  show ?thesis
  using assms
    ipc-weakly-step-consistent
    ev-wait-all-weakly-step-consistent
    ev-wait-one-weakly-step-consistent
    ev-signal-weakly-step-consistent
    vpeq-refl
    unfolding atomic-step-def
  apply (cases ipt, auto)
  apply (simp split: ev-consume-t.splits ev-wait-stage-t.splits)
  by (simp split: ev-signal-stage-t.splits)
qed

4.8 Separation kernel model

theory Separation-kernel-model
imports ../../step/Step
    ../../step/Step-invariants
    ../../step/Step-vpeq
    ../../step/Step-vpeq-locally-respects
    ../../step/Step-vpeq-weakly-step-consistent

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.

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

definition s0 :: state-t where
  s0 ≡ (sp-impl-subj-subj = Policy.sp-spec-subj-subj,
    sp-impl-subj-obj = Policy.sp-spec-subj-obj,
    current = initial-current,
    obj = initial-obj,
    thread = λ . (ev-counter = 0))

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

4.8.2 Types for instantiation of the generic model

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

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

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

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

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

lemma rstate-up-down[simp]:
  assumes atomic-step-invariant s
  shows (↓↑ s) = s
  using 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 ⇒ thread-id-t where
  rcurrent s = current ↓ s
definition \texttt{rstep} :: \texttt{rstate-t} \Rightarrow \texttt{raction-t} \Rightarrow \texttt{rstate-t} where
\texttt{rstep s a} \equiv \uparrow(\texttt{atomic-step (s) a})

Each CISK domain is identified with a thread id.

type-synonym \texttt{rdom-t} = \texttt{thread-id-t}

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

datatype \texttt{visible-obj-t} = \texttt{VALUE obj-t} | \texttt{EXCEPTION}

type-synonym \texttt{routput-t} = \texttt{page-t} \Rightarrow \texttt{visible-obj-t}

definition \texttt{routput-f} :: \texttt{rstate-t} \Rightarrow \texttt{raction-t} \Rightarrow \texttt{routput-t} where
\texttt{routput-f s a p} \equiv if \texttt{sp-impl-subj-obj (s) (partition (rcurrent s)) (PAGE p) READ} then
\texttt{VALUE (obj (s) (PAGE p))}
else
\texttt{EXCEPTION}

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

definition \texttt{rprecondition} :: \texttt{rstate-t} \Rightarrow \texttt{rdom-t} \Rightarrow \texttt{raction-t} \Rightarrow \texttt{bool} where
\texttt{rprecondition s d a} \equiv \texttt{atomic-step-precondition (s) d a}

abbreviation \texttt{rinvariant} where
\texttt{rinvariant s} \equiv \texttt{True} — The invariant is already in the state type.

Translate view-partitioning and interaction-allowed relations.

definition \texttt{rvpeq} :: \texttt{rdom-t} \Rightarrow \texttt{rstate-t} \Rightarrow \texttt{rstate-t} \Rightarrow \texttt{bool} where
\texttt{rvpeq u s1 s2} \equiv \texttt{vpeq (partition u) (s1) (s2)}

definition \texttt{rifp} :: \texttt{rdom-t} \Rightarrow \texttt{rdom-t} \Rightarrow \texttt{bool} where
\texttt{rifp u v} = Policy.ifp (partition u) (partition v)

Context Switches

definition \texttt{rcswitch} :: \texttt{nat} \Rightarrow \texttt{rstate-t} \Rightarrow \texttt{rstate-t} where
\texttt{rcswitch n s} \equiv \uparrow((s) (\texttt{current} \equiv (\texttt{SOME t} . \texttt{True}) \})

4.8.3 Possible action sequences

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

definition \texttt{is-SK-IPC} :: \texttt{raction-t list} \Rightarrow \texttt{bool} where
\texttt{is-SK-IPC aseq} \equiv \exists \texttt{dir partner page} .
\texttt{aseq} = [\texttt{SK-IPC dir PREP partner page,SK-IPC dir WAIT partner page,SK-IPC dir Buf (SOME page') . True)} ] partner page]

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

definition \texttt{is-SK-EV-WAIT} :: \texttt{raction-t list} \Rightarrow \texttt{bool} where
\texttt{is-SK-EV-WAIT aseq} \equiv \exists \texttt{consume} .
\texttt{aseq} = [\texttt{SK-EV-WAIT EV-PREP consume} ,
\texttt{SK-EV-WAIT EV-WAIT consume} ,
\texttt{SK-EV-WAIT EV-FINISH consume} ]

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

definition \texttt{is-SK-EV-SIGNAL} :: \texttt{raction-t list} \Rightarrow \texttt{bool} where
\texttt{is-SK-EV-SIGNAL aseq} \equiv \exists \texttt{partner} .
aseq = [SK-EV-SIGNAL EV-SIGNAL-PREP partner, SK-EV-SIGNAL EV-SIGNAL-FINISH partner]

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

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

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 raborting
  where raborting s ≡ aborting (↓s)
abbreviation rwaiting
  where rwaiting s ≡ waiting (↓s)

definition rset-error-code :: rstate-t ⇒ raction-t ⇒ rstate-t
  where rset-error-code s a ≡ s

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

definition rkinvolved :: int-point-t ⇒ rdom-t set
  where rkinvolved a ≡
    case a of
      SK-IPC dir WAIT partner page
        ⇒ {partner}
      SK-EV-SIGNAL EV-SIGNAL-FINISH partner =⇒ {partner}
      _ =⇒ {}
using assms atomic-step-weakly-step-consistent rstate-invariant atomic-step-preserves-invariants
unfolding rcurrent-def rstep-def rvpeq-def rprecondition-def
by auto

lemma inst-local-respect:
  assumes not-ifp: ~rifp (rcurrent s) u
  and prec: rprecondition s (rcurrent s) a
  shows rvpeq u s (rstep s a)
using assms atomic-step-respects-policy rstate-invariant atomic-step-preserves-invariants
unfolding rifp-def rprecondition-def rvpeq-def rstep-def rcurrent-def
by auto

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

lemma inst-cswitch-independent-of-state:
  assumes rcurrent s = rcurrent t
\begin{verbatim}
shows recurr (rcswitch n s) = recurr (rcswitch n t)
using rstate-invariant csbirch-preserves-invariants unfolding recurr-def rcsbirch-def by simp

lemma inst-csbirch-consistency:
  assumes rvpq u s t
  shows rvpq u (rcswitch n s) (rcswitch n t)
proof-
  have 1: vpeq (partition u) (↓s) (↓(rcswitch n s))
    using rstate-invariant csbirch-consistency-and-respect csbirch-preserves-invariants
    unfolding rcsbirch-def
    by auto
  have 2: vpeq (partition u) (↓t) (↓(rcswitch n t))
    using rstate-invariant csbirch-consistency-and-respect csbirch-preserves-invariants
    unfolding rcsbirch-def
    by auto
  from 1 2 assms show ?thesis unfolding rvpq-def using vpq-rel by metis
qed

For the \textsc{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 \textsc{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 \textsc{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 precon-
dition holds initially (for the first step of an action sequence) and after doing one step.

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

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

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

 assume WAIT: aseq = [SK-EV-WAIT EV-PREP consume, SK-EV-WAIT EV-WAIT consume, SK-EV-WAIT EV-FINISH consume]
 {assume 0: n=0
 from 0 WAIT prec not-aborting
 have ?thesis
 unfolding rprecondition-def atomic-step-precondition-def
 by(auto)
 }
 moreover
 {
assume \( I : n = 1 \)
from \( I \) \( \text{WAIT} \) \( \text{prec} \) \( \text{not-waiting} \)
have \( ?\text{thesis} \)
unfolding \( \text{rprecondition-def atomic-step-precondition-def} \)
by (auto)
}
moreover
from \( \text{WAIT} \)
have \( \text{length aseq} = 3 \)
by auto
ultimately
have \( ?\text{thesis} \)
using \( n\text{-bound} \)
by arith
}
thus \( ?\text{thesis} \)
using assms
unfolding \( \text{is-SK-EV-WAIT-def} \)
by auto
qed

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

lemma \( \text{prec-after-EV-SIGNAL-step} \):
assumes \( \text{prec: rprecondition s (rcurrent s) (aseq ! n)} \)
and \( n\text{-bound: Suc n < length aseq} \)
and \( \text{SIGNAL: is-SK-EV-SIGNAL aseq} \)
and \( \text{not-aborting: ~raborting s (rcurrent s) (aseq ! n)} \)
and \( \text{not-waiting: ~rwaiting s (rcurrent s) (aseq ! n)} \)
shows \( \text{rprecondition (rstep s (aseq ! n)) (rcurrent s) (aseq ! Suc n)} \)
proof–
{ fix partner
  assume \( \text{SIGNAL1: aseq = [SK-EV-SIGNAL EV-SIGNAL-PREP partner, SK-EV-SIGNAL EV-SIGNAL-FINISH partner]} \)
  { assume \( 0: n=0 \)
from \( 0 \) \( \text{SIGNAL1} \) \( \text{prec not-aborting} \)
have \( ?\text{thesis} \)
unfolding \( \text{rprecondition-def atomic-step-precondition-def ev-signal-precondition-def aborting-def rstep-def atomic-step-def} \)
by auto
}
moreover
from \( \text{SIGNAL1} \)
have \( \text{length aseq} = 2 \)
by auto
ultimately
have \( ?\text{thesis} \)
using \( n\text{-bound} \)
by arith
}
thus \( ?\text{thesis} \)
using assms
unfolding \( \text{is-SK-EV-SIGNAL-def} \)
by auto
qed

lemma \( \text{on-set-object-value} \):
shows \( \text{sp-impl-subj-subj (set-object-value ob val s) = sp-impl-subj-subj s} \)
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
by (auto split: int-point-tplits ipc-stage-tplits ipc-direction-tplits ev-signal-stage-tplits ev-consume-tplits ev-wait-stage-tplits ev-signal-stage-tplits)

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
by (auto split: int-point-tplits ipc-stage-tplits ipc-direction-tplits ev-consume-tplits ev-wait-stage-tplits ev-signal-stage-tplits)

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
by (auto split: int-point-tplits ipc-stage-tplits ipc-direction-tplits ev-consume-tplits ev-wait-stage-tplits ev-signal-stage-tplits)

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
by (auto split: int-point-tplits ipc-stage-tplits ipc-direction-tplits ev-consume-tplits ev-wait-stage-tplits ev-signal-stage-tplits)

lemma prec-dom-independent:
sshows \( \forall \) s \( d a a' \) . rcurrent s \( \not= \) d \( \land \) rprecondition s d a \( \longrightarrow \) rprecondition (rstep s a') d a
unfolding rcurrent-def rprecondition-def rstep-def atomic-step-def
by (auto split: int-point-tplits ev-consume-tplits ev-wait-stage-tplits ev-signal-stage-tplits)

lemma ipc-precondition-after-cswitch[simp]:
sshows ipc-precondition d dir partner page ((1 s)(current := new-current))
= ipc-precondition d dir partner page (↓ s)

\textbf{unfolding} ipc-precondition-def
\textbf{by (auto split: ipc-direction-t.splits)}

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

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

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

\textbf{lemma} waiting-after-IPC-step:
lemma raborting-consistent:
shows \( \forall s t u. \, \text{rvpeq } u s t \rightarrow \text{raborting } s u = \text{raborting } t u \)
proof-
{ 
fix s t u 
assume \( \text{vpeq: rvpeq } u s t \)
{ 
fix \( a \)
from \( \text{vpeq ipc-precondition-weakly-step-consistent rstate-invariant} \)

have \( \land \, \text{tid dir partner page } . \, \text{ipc-precondition } u \, \text{dir partner page } (\downarrow s) = \text{ipc-precondition } u \, \text{dir partner page } (\downarrow t) \)

unfolding \( \text{rvpeq-def} \)
by auto
with \( \text{vpeq rstate-invariant} \) have \( \text{raborting } s u a = \text{raborting } t u a \)

unfolding \( \text{aborting-def rvpeq-def vpeq-def vpeq-local-def ev-signal-precondition-def} \)
\( \text{vpeq-subj-subj-def atomic-step-invariant-def sp-subset-def rep-def} \)

apply \( \text{(auto split: int-point-t.splits ipc-stage-t.splits ipc-direction-t.splits ev-wait-stage-t.splits)} \)
by blast
}
hence \( \text{raborting } s u = \text{raborting } t u \) by auto
}
thus \( \Box \text{thesis} \) by auto
qed

lemma aborting-dom-independent:
assumes \( \text{rcurrent } s \neq d \)
shows \( \text{raborting } (rstep s a) \, d \, a' = \text{raborting } s \, d \, a' \)
proof –
have \( \land \, \text{tid dir partner page } . \, \text{ipc-precondition } tid dir partner page s = \text{ipc-precondition } tid dir partner page (\text{atomic-step } s \, a) \)
\( \land \, \text{ev-signal-precondition } tid partner s = \text{ev-signal-precondition } tid partner (\text{atomic-step } s \, a) \)
proof –
fix \( \text{tid dir partner page } s \)
let \( ?s = \text{atomic-step } s \, a \)

have \( \forall p q . \, \text{sp-impl-subj-subj } s \, p \, q = \text{sp-impl-subj-subj } ?s \, p \, q \)
\( \land \, (\forall p x m . \, \text{sp-impl-subj-obj } s \, p \, x \, m = \text{sp-impl-subj-obj } ?s \, p \, x \, m) \)
unfolding \( \text{atomic-step-def atomic-step-ipc-def} \)
\( \text{atomic-step-ev-wait-all-def atomic-step-ev-wait-one-def} \)
\( \text{atomic-step-ev-signal-def set-object-value-def} \)
by \( \text{(auto split: int-point-t.splits ipc-stage-t.splits ipc-direction-t.splits ev-wait-stage-t.splits ev-consume-t.splits ev-signal-stage-t.splits)} \)
thus \( \text{ipc-precondition } tid dir partner page s = \text{ipc-precondition } tid dir partner page (\text{atomic-step } s \, a) \)
\( \land \, \text{ev-signal-precondition } tid partner s = \text{ev-signal-precondition } tid partner (\text{atomic-step } s \, a) \)
unfolding \( \text{ipc-precondition-def ev-signal-precondition-def by simp} \)
qed
moreover have \( \land \, b . \, (\downarrow (\uparrow (\text{atomic-step } (\downarrow s) \, b))) = \text{atomic-step } (\downarrow s) \, b \)
using \( \text{rstate-invariant atomic-step-preserves-invariants rstate-up-down} \) by auto
ultimately show \( ?\text{thesis} \)
unfolding aborting-def rstep-def ev-signal-precondition-def

by (simp split: int-point-t.splits ipc-stage-t.splits ev-wait-stage-t.splits
ev-signal-stage-t.splits)

qed

lemma ipc-precondition-of-partner-consistent:
assumes vpeq : \( \forall d \in rkinvolved (SK-IPC dir WAIT partner page) \). rvpeq d s t
shows ipc-precondition partner dir' u page' (\downarrow s) = ipc-precondition partner dir' u page' \downarrow t
proof--
  from assms ipc-precondition-weakly-step-consistent rstate-invariant
  show ?thesis
  unfolding rvpeq-def rkinvolved-def
  by auto

qed

lemma ev-signal-precondition-of-partner-consistent:
assumes vpeq : \( \forall d \in rkinvolved (SK-EV-SIGNAL EV-SIGNAL-FINISH partner) \). rvpeq d s t
shows ev-signal-precondition partner u (\downarrow s) = ev-signal-precondition partner u (\downarrow t)
proof--
  from assms ev-signal-precondition-weakly-step-consistent rstate-invariant
  show ?thesis
  unfolding rvpeq-def rkinvolved-def
  by auto

qed

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

qed

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

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

\textbf{lemma} involved-ifp:
\textbf{shows} \(\forall s a. \forall d \in rkinvolved a \cdot \text{rprecondition } s (\text{rcurrent } s) a \rightarrow \text{rifp } d (\text{rcurrent } s)\) \\
\textbf{proof} \-
\[
\text{fix } s a d \\
\text{assume d-involved: } d \in rkinvolved a \\
\text{assume prec: } \text{rprecondition } s (\text{rcurrent } s) a \\
\text{from d-involved prec have rifp } d (\text{rcurrent } s) \\
\text{using ipc-precondition-ensures-ifp ev-signal-precondition-ensures-ifp rstate-invariant} \\
\textbf{unfolding} rkinvolved-def rprecondition-def atomic-step-precondition-def rcurrent-def Kernel.involved-def \\
\text{by (cases a simp auto split: int-point-t.splits ipc-stage-t.splits ev-signal-stage-t.splits)} \\
\}
\text{thus } \textbf{thesis} \textbf{by auto}
qed

\textbf{lemma} spec-of-waiting-ev:
\textbf{shows} \(\forall s a. \text{rwaiting } s (\text{rcurrent } s) (\text{SK-EV-WAIT EV-FINISH EV-CONSUME-ALL}) \rightarrow \text{rstep } s a = s\) \\
\textbf{unfolding} waiting-def \\
\textbf{by auto}

\textbf{lemma} spec-of-waiting-ev-w:
\textbf{shows} \(\forall s a. \text{rwaiting } s (\text{rcurrent } s) (\text{SK-EV-WAIT EV-WAIT EV-CONSUME-ALL}) \rightarrow \text{rstep } s (\text{SK-EV-WAIT EV-WAIT EV-CONSUME-ALL}) = s\) \\
\textbf{unfolding} rstep-def atomic-step-def \\
\textbf{by} (auto split: int-point-t.splits ipc-stage-t.splits ev-wait-stage-t.splits)

\textbf{lemma} spec-of-waiting:
\textbf{shows} \(\forall s a. \text{rwaiting } s (\text{rcurrent } s) a \rightarrow \text{rstep } s a = s\) \\
\textbf{unfolding} waiting-def rstep-def atomic-step-def atomic-step-ipc-def \\
atomic-step-ev-signal-def atomic-step-ev-wait-all-def \\
atomic-step-ev-wait-one-def \\
\textbf{by} (auto split: int-point-t.splits ipc-stage-t.splits ev-wait-stage-t.splits)
end
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
(rcurrent)
rcurrent
rkinvolved
rifp
rvpeq
rAS-set
rinvariant
rprecondition
raborting
rwaiting
rsset-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 ∧ True ∧ rprecondition s (rcurrent s) a ∧ True ∧ rprecondition t (rcurrent t) a ∧ rcurrent s = rcurrent t −→ rvpeq u (rstep s a) (rstep t a)
using inst-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 u. ¬ rifp (rcurrent s) u ∧ True ∧ rprecondition s (rcurrent s) a −→ rvpeq u s (rstep s a)
using inst-local-respect by blast
— show cswitch is independent of state
show ∀ n s t. rcurrent s = rcurrent t −→ rcurrent (rcswitch n s) = rcurrent (rcswitch n t)
using inst-cswitch-independent-of-state by metis
— show cswitch consistency
show ∀ u s t n. rvpeq u s t −→ rvpeq u (rcswitch n s) (rcswitch n t)
using inst-cswitch-consistency by metis
— Show the empt action sequence is in AS-set
show [] ∈ rAS-set
unfolding rAS-set-def
by auto
— The invariant for the initial state, already encoded in rstate-t
show True by auto
— Step function of the invariant, already encoded in rstate-t

show ∀ s n. True → True by auto
— The precondition does not change with a context switch

show ∀ s d n a. rprecondition s d a → rprecondition (rcswitch n s) d a
using precondition-after-cswitch by blast
— The precondition holds for the first action of each action sequence

show ∀ s d a seq. True ∧ a seq ∈ rAS-set ∧ a seq ≠ [] → rprecondition s d (hd a seq)
using prec-first-IPC-action prec-first-EV-WAIT-action prec-first-EV-SIGNAL-action
unfolding rAS-set-def is-sub-seq-def by auto
— The precondition holds for the next action in an action sequence, assuming the sequence is not aborted or delayed

show ∀ s a a'. (∃ aseq: rAS-set. is-sub-seq a a' aseq) ∧ True ∧ rprecondition s (rcurrent s) a ∧ ¬ raborting s (rcurrent s) a →
  rprecondition (rstep s a) (rcurrent s) a'
unfolding rAS-set-def is-sub-seq-def by auto
— Steps of other domains do not influence the precondition

show ∀ s d a a'. rcurrent s ≠ d ∧ rprecondition s d a → rprecondition (rstep s a') d a
using prec-dom-independent by blast
— The precondition holds for the next action in an action sequence, assuming the sequence is not aborted or delayed

show ∀ s a. True → True by auto
— Aborting does not depend on a context switch

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

show ∀ s a d. rcurrent s ≠ d → raborting (rstep s a) d = raborting s d
using aborting-dom-independent by auto
— Aborting is consistent

show ∀ s t u. rvpeq u s t → raborting s u = raborting t u
using raborting-consistent by auto
— Waiting does not depend on a context switch

show ∀ n s. rwwaiting (rcswitch n s) = rwwaiting s
using waiting-switch-independent by auto
— Waiting does not depend on a context switch

show ∀ s t u a. rvpeq (rcurrent s) s t ∧ (∀ d ∈ rkinvolved a . rvpeq d s t)
  ∧ rvpeq u s t → rwwaiting s u a = rwwaiting t u a
unfolding Kernel.involved-def
using waiting-consistent by auto
— Domains that are involved in an action may influence the domain of the action

show ∀ s a. ∃ d ∈ rkinvolved a . rprecondition s (rcurrent s) a → rffp d (rcurrent s)
using involved-ifp by blast
— An action that is waiting does not change the state

show ∀ s a. rwwaiting s (rcurrent s) a → rstep s a = s
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
using rset-error-code-def
— Proof obligations for set-error-code. Right now, they are all trivial

show ∀ s t u a. rvpeq u s t → rvpeq u (rset-error-code s a) (rset-error-code t a)
using rset-error-code-def
by auto

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

show ∀ s d a a'. rprecondition s d a ∧ raborting s (rcurrent s) a' → rprecondition (rset-error-code s a') d a
  unfolding rset-error-code-def
  by auto

show ∀ s d a a'. rcurrent s ≠ d ∧ rwaiting s d a → rwaiting (rset-error-code s a') d a
  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
  (=) 42 — interrupt function (yet unspecified)
  rkinvolved — 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) noninference (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 OS’s 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- security 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 et 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 Sabelfelds [26] 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 OS’s, 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 (PO’s). These PO’s 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-security as a starting point. We add kernel control mechanisms, interrupts and context switches. Ideally, we would simply prove IP-security 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.
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 API in order to 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 Union’s Programme FP7/2007 – 2013 under grant agreement number ICT-318353.

References


