# D31.1

## Formal Specification of a Generic Separation Kernel

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

<table>
<thead>
<tr>
<th><strong>Deliverable type:</strong></th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Deliverable reference number:</strong></td>
<td>ICT-318353 / D31.1 / 0.0</td>
</tr>
<tr>
<td><strong>Activity and Work package contributing to deliverable:</strong></td>
<td>Activity 3 / WP 3.1</td>
</tr>
<tr>
<td><strong>Due date:</strong></td>
<td>September 2013 – M12</td>
</tr>
<tr>
<td><strong>Actual submission date:</strong></td>
<td>30th November, 2021</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Responsible organisation:</strong></th>
<th>Open University of The Netherlands</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Editors:</strong></td>
<td>Freek Verbeek, Julien Schmaltz</td>
</tr>
<tr>
<td><strong>Dissemination level:</strong></td>
<td>PU</td>
</tr>
<tr>
<td><strong>Revision:</strong></td>
<td>0.0 (r-2)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Abstract:</strong></th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Keywords:</strong></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 nº 318353.
Executive Summary

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

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

1 **Introduction** ................................................. 2

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

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

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

5 Related Work .............................................................. 91

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

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

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

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

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

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

Overview

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

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

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

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

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

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

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

## 2 Preliminaries

### 2.1 Binders for the option type

```lean
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 “\( m \to \alpha \)” means “First compute \( m \), if it is None then return True, otherwise pass the result to \( \alpha \)”. B2 is a short hand for sequentially doing two independent computations. The following syntax is associated to B2: “\( m_1 \| m_2 \to \alpha \)” represents “First compute \( m_1 \) and \( m_2 \), if one of them is None then return True, otherwise pass the result to \( \alpha \)”.

**definition** B :: \( 'a \Rightarrow \Rightarrow bool \) (infixl \( \to \) 65)

**where** B m \( \alpha \equiv case \ m \ of \ None \Rightarrow True \mid (Some \ a) \Rightarrow \alpha \ a \)

**definition** B2 :: \( 'a \Rightarrow \Rightarrow bool \Rightarrow bool \)

**where** B2 m1 m2 \( \alpha \equiv m \Rightarrow (\lambda \ a . \ m2 \Rightarrow (\lambda b . \ (\alpha \ a \ b))) \)

**syntax** B2 :: \( ['a \Rightarrow \Rightarrow 'a \Rightarrow \Rightarrow bool] \Rightarrow bool ((\cdot \ - \ -) [0, 0, 10] 10) \)

Some rewriting rules for the binders

**lemma** rewrite-B2-to-cases[simp]:

- shows B2 s t f = \((\case s \ of \ None \Rightarrow True \mid (Some \ s1) \Rightarrow (\case t \ of \ None \Rightarrow True \mid (Some \ t1) \Rightarrow f \ s1 \ t1))\)

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

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

- shows None \(\to\) \(\alpha \Rightarrow True \)

**unfolding** B-def by(auto)

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

- shows m \(\Rightarrow\) \((\lambda \ a . \ True) \Rightarrow True \)

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

**lemma** rewrite-B2-cases:

- shows \((\case a \ of \ None \Rightarrow True \mid (Some \ s) \Rightarrow (\case b \ of \ None \Rightarrow True \mid (Some \ t) \Rightarrow f \ s \ t))\)

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

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

**definition** strict-equal :: \( 'a \Rightarrow \Rightarrow 'a \Rightarrow bool \)

**where** strict-equal m a \(\equiv case \ m \ of \ None \Rightarrow False \mid (Some \ a') \Rightarrow a' = a \)

end

### 2.2 Theorems on lists

**theory** List-Theorems

**imports** Main

begin

**definition** lastn :: nat \(\Rightarrow\) \( 'a \ list \Rightarrow 'a \ list \)

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

**definition** is-sub-seq :: \( 'a \Rightarrow \Rightarrow 'a \ list \Rightarrow bool \)

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

**definition** prefixes :: \( 'a \ list \ set \Rightarrow 'a \ list \ set \)

**where** prefixes s \(\equiv\) \[x . \ \exists n \ y . \ n > 0 \ \land \ y \in s \ \land \ take \ n \ y = x\]

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

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

**lemma** length-ge-one:

- shows x \(\not\preceq\) \[] \(\Rightarrow\) \(\length x \geq 1 \ by\) (induct x,auto)

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

- shows x \(\not\preceq\) \[] \(\Rightarrow\) \(\lastn ((\length x) - 1) x = tl x \ unfolding \ lastn-def\)

**using** length-ge-one[where x=x] by auto

**lemma** Suc-m-minus-n[simp]:

- shows \(m \geq n \Rightarrow Suc \ m - n = Suc \ (m - n) \ by\ auto\)
lemma `lastn-one-less`:
shows \( n > 0 \land n \leq \text{length } x \quad \Rightarrow \quad \text{lastn} \ n \ x = (\ a \# y) \quad \rightarrow \quad \text{lastn} \ (n - 1) \ x = y \)
ungfolding `lastn-def`
using \( \text{drop-Suc}[\text{where } n=\text{length } x - n \text{ and } xs=x] \) \( \text{drop-tl}[\text{where } n=\text{length } x - n \text{ and } xs=x] \)
by \((\text{auto})\)

lemma `list-sub-implies-member`:
shows \( \forall \ a \ x \ . \ \text{set} \ (a \# x) \subseteq Z \quad \Rightarrow \quad \ a \in Z \)
by \((\text{auto})\)

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

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

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

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

lemma `first-second-is-sub-seq`:
shows \( \text{length } x \geq 2 \quad \Rightarrow \quad \text{is-sub-seq (hd } x) \ (x!1) \ x \)
proof
assume \( \text{length } x \geq 2 \)
hence \( I: \ (\text{Suc } 0) < \text{length } x \quad \Rightarrow \quad \text{auto} \)
hence \( x!0 = \text{hd } x \quad \Rightarrow \quad \text{by (cases } x,\text{auto}) \)
from this \( I \) show `is-sub-seq (hd } x) \ (x!1) \ x` unfgfolding `is-sub-seq-def` by \((\text{auto})\)
qed

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

lemma `def-of-hd`:
shows \( y = a \# x \quad \Rightarrow \quad \text{hd } y = a \quad \Rightarrow \quad \text{simp} \)
lemma `def-of-tl`:
shows \( y = a \# x \quad \Rightarrow \quad \text{tl } y = x \quad \Rightarrow \quad \text{simp} \)
lemma `drop-yields-results-implies-nbound`:
shows \( \text{drop } n \ x \neq [] \quad \Rightarrow \quad n < \text{length } x \)
by \((\text{induct } x,\text{auto})\)
lemma `consecutive-is-sub-seq`:
shows \( a \# (b \# x) = \text{lastn } n \ y \quad \Rightarrow \quad \text{is-sub-seq } a \ b \ y \)
proof
assume \( I: a \# (b \# x) = \text{lastn } n \ y \)
from I \( \text{drop-Suc}[\text{where } n=\text{(length } y) - n \text{ and } xs=y] \)
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:

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

- **dom_t** A domain is an entity executing actions and making calls to the kernel. This type represents the names of all domains. Later on, we define security policies in terms of domains.

- **action_t** Actions of type `action_t` represent atomic instructions that are executed by the kernel. As kernel actions are assumed to be atomic, we assume that after each kernel action an interrupt point can occur.

- **action_t execution** An execution of some domain is the code or the program that is executed by the domain. One call from a domain to the kernel will typically trigger a succession of one or more kernel actions. Therefore, an execution is represented as a list of sequences of kernel actions. Non-kernel actions are not 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 = case oa of None ⇒ s | (Some a) ⇒ kstep s a

**definition** precondition:: 'state-t ⇒ 'action-t option ⇒ bool
where precondition s a = a ⇒ kprecondition s

**definition** involved
where involved oa = case oa of None ⇒ {} | (Some a) ⇒ kinvolved a
Execution semantics are defined as follows: a run consists of consecutively running sequences of actions. These sequences are interruptable. Run first checks whether an interrupt occurs. When this happens, function cswitch may switch the context. Otherwise, function control is used to determine the next action \( a \), which also yields a new state \( s' \). Action \( a \) is executed by executing \( \text{step}\,(s'\,a) \). 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.

\[
\text{function run} \colon \text{time-t} \Rightarrow \text{state-t option} \Rightarrow \text{dom-t} \Rightarrow \text{action-t execution} \Rightarrow \text{state-t option} \\
\text{where} \\
\text{run} \ 0 \ s \ \text{execs} = s \\
\text{run} \ (\text{Suc} \ n) \ \text{None} \ \text{execs} = \text{None} \\
\text{interrupt} \ (\text{Suc} \ n) \Rightarrow \text{run} \ (\text{Suc} \ n) \ (\text{Some} \ s) \ \text{execs} = \text{run} \ n \ (\text{Some} \ (\text{cswitch} \ (\text{Suc} \ n) \ s)) \ \text{execs} \\
\text{纽带 interrupt} \ (\text{Suc} \ n) \Rightarrow \text{thread-empty(\text{execs} \ (\text{current} \ s))} \Rightarrow \text{run} \ (\text{Suc} \ n) \ (\text{Some} \ s) \ \text{execs} = \text{run} \ n \ (\text{Some} \ s) \ \text{execs} \\
\text{纽带 interrupt} \ (\text{Suc} \ n) \Rightarrow \text{thread-empty(\text{execs} \ (\text{current} \ s))} \Rightarrow \text{precondition} \ (\text{next-state} \ s \ \text{execs}) \ (\text{next-action} \ s \ \text{execs}) \Rightarrow \text{run} \ (\text{Suc} \ n) \ (\text{Some} \ s) \ \text{execs} = \text{None} \\
\text{纽带 interrupt} \ (\text{Suc} \ n) \Rightarrow \text{thread-empty(\text{execs} \ (\text{current} \ s))} \Rightarrow \text{precondition} \ (\text{next-state} \ s \ \text{execs}) \ (\text{next-action} \ s \ \text{execs}) \Rightarrow \\
\text{run} \ (\text{Suc} \ n) \ (\text{Some} \ s) \ \text{execs} = \text{run} \ n \ (\text{Some} \ (\text{step} \ (\text{next-state} \ s \ \text{execs}) \ (\text{next-action} \ s \ \text{execs}))) \ (\text{next-exec} \ s \ \text{execs}) \\
\text{using not0-implies-Suc by (metis option.exhaust prod-cases3.auto)} \\
\text{termination by lexicographic-order} \\
\end
\]

### 3.2 SK (Separation Kernel)

The 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).

\[
\text{definition actions-in-execution: } '\text{action-t execution} \Rightarrow '\text{action-t set} \\
\text{where} \text{actions-in-execution exec} \equiv \{ \ a \ . \ \exists \ aseq \in \text{set exec} . \ a \in \text{set aseq} \} \\
\]
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: ∀ a s t. 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 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 execs . vpeq (current s) s t ∧ (∀ d ∈ involved (next-action s execs) . vpeq d s t) ∧ current s = current t → dropdown (snd (control s (current s) (execs (current s)))) = fst (snd (control t (current s) (execs (current s))))
  and next-state-consistent: ∀ s t 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 → (∀ a . a ∈ actions-in-execution (execs (current s)))
  and next-exec-subset: ∀ s execs u . actions-in-execution (next-exec s execs u) ⊆ actions-in-execution (execs u)
begin

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

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 $u$. 

EURO-MILS D31.1 Page 10 of 94
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 ≡
('dom-t ⇒ 'action-t execution) ⇒ 'dom-t ⇒ ('dom-t ⇒ 'action-t execution)
where purge execs u ≡ λ d . (if unrelated d u then
(SOME alpha . realistic-execution alpha)
else execs d)

A normal run from initial state s0 ending in state s_f is equivalent to a run purged for domain
(currents_f).

definition NI-unrelated where NI-unrelated
≡ ∀ execs a n . run n (Some s0) execs →
(λ s s-f . run n (Some s0) (purge execs (current s-f)) →
(λ s f . output-f s-f a = output-f s f2 a ∧ current s-f = current s-f2))

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

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

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

abbreviation equivalent-states :: 'state-t option ⇒ 'state-t option ⇒ 'dom-t ⇒ bool
where equivalent-states s t u ≡ s t (FD v . ifp^** v u → vpeq v s t) ∧ current s = current t

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

definition view-partitioned::bool where view-partitioned
≡ ∀ execs ms mt n u . equivalent-states ms mt u →
(run n ms execs ∥
run n mt (purge execs u) →
(λ rs rt . vpeq u rs rt ∧ current rs = current rt))

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

definition purged-relation ≡ 'dom-t ⇒ ('dom-t ⇒ 'action-t execution) ⇒ ('dom-t ⇒ 'action-t execution) ⇒ bool
where purged-relation u execs1 execs2 ≡ ∀ d . ifp^** d u → 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
≡ ∀ execs1 execs2 s t n u . equivalent-states s t u ∧ purged-relation u execs1 execs2 → equivalent-states (run n s execs1) (run n t execs2)

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

lemma vpeq-s-n:
assumes prec-t: precondition (next-state t execs2) (next-action t execs2)
assumes not-ifp-curr-u: ¬ ifp^** (current t) u
assumes vpeq-s-t: ∀ v . ifp^** v u → vpeq v s t
shows (∀ v . ifp^** v u → vpeq v s (step (next-state t execs2) (next-action t execs2)))
proof-
{ 
  fix v
  assume ifp-v-uc: ifp^** v u

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

thus ?thesis by auto
qed

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

lemma vpeq-ns-t:
  assumes prec-s: precondition (next-state s execs) (next-action s execs)
  assumes not-ifp-curr-u: ¬ifp^** (current s) u
  assumes vpeq-s-t: ∀ v . ifp^** v u → vpeq v s t
  shows ∀ v . ifp^** v u → vpeq v (step (next-state s execs) (next-action s execs)) t

proof-
{ 
  fix v
  assume ifp-v-uc: ifp^** v u

from ifp-v-u and not-ifp-curr-u have unrelated: ¬ifp^** (current s) v using rtranclp-trans by metis
from this current-next-state[THEN spec,THEN spec,where xl=s] vpeq-reflexive
  unrelated locally-respects[THEN spec,THEN spec,THEN spec,where xl=next-state s execs and s=v and
  x2=the (next-action s execs)] prec-s
  have vpeq v (next-state s execs) (step (next-state s execs) (next-action s execs))
  unfolding step-def precondition-def B-def
  by (cases next-action s execs,auto)

from unrelated this locally-respects-next-state vpeq-transitive have vpeq v s (step (next-state s execs) (next-action s execs)) t by blast

from this and ifp-v-u and vpeq-s-t and vpeq-symmetric and vpeq-transitive have vpeq v (step (next-state s execs) (next-action s execs)) t by metis

thus ?thesis by auto
qed

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

lemma vpeq-ns-nt-ifp-u:
  assumes vpeq-s-t: ∀ v . ifp^** v u → vpeq v s t'
  and current-s-t: current s = current t'
  shows precondition (next-state s execs) a ∧ precondition (next-state t' execs) a → (ifp^** (current s) u → (∀ v . ifp^** v u → vpeq v (step (next-state s execs) a) (step (next-state t' execs) a)))

proof-
fix a
  assume precs: precondition (next-state s execs) a ∧ precondition (next-state t' execs) a
assume ifp-curr: ifp\^\ast\ast\ast ((current) s) u
from vpeq-s-t have vpeq-curr-s-t: ifp\^\ast\ast\ast ((current) s) u \rightarrow vpeq (current) s t \text{ by auto}
from ifp-curr prec
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 . ifp\^\ast\ast\ast v u \rightarrow 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-s-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 . ifp\^\ast\ast\ast v u \rightarrow vpeq v s t'
shows \neg ifp\^\ast\ast\ast (current) s u \land precondition (next-state t' execs2) (next-action t' execs2) \rightarrow (\forall v . ifp\^\ast\ast\ast v u \rightarrow 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\^\ast\ast\ast (current) s u 
assume prec-t: precondition (next-state t' execs2) (next-action t' execs2)
fix a a' v
assume ifp-v-uc ifp\^\ast\ast\ast v u
from not-ifp and purged-a-a2 have \neg ifp\^\ast\ast\ast (current) s u unfolding purged-relation-def by auto
from this and ifp-v-uc have not-ifp-curr-v: \neg ifp\^\ast\ast\ast (current) s v using rtranclp-trans by metis
from this current-next-state[THEN spec.THEN spec.THEN spec.where x1=s and x=execs] prec-s vpeq-reflexive
locally-respects[THEN spec.THEN spec.THEN spec.where x1=next-state s execs and x2=the (next-action s 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 \theta: 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 I: \neg ifp\^\ast\ast\ast (current) t' v using rtranclp-trans by auto
from 0 I 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-uc 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 \text{thesis by auto}
qed

A run with a purged list of actions appears identical to a run without purging, when starting from two states that appear identical.
lemma unwinding-implies-view-partitioned-ind:
shows view-partitioned-ind
proof-
\{ 
  fix execs execs2 s t n u 
  have equivalent-states s t u \land purged-relation u execs execs2 \implies equivalent-states (run n s execs) (run n t execs2) u 
  proof (induct n s execs arbitrary: t u execs2 rule: run.induct)
  case (1 s execs t u execs2)
  show ?case by auto
  next
  case (2 n execs t u execs2)
  show ?case by simp
  next
  case (3 n s execs t u execs2)
  assume interrupt-s \(\vdash\) interrupt (Suc n)
  assume IH \(\vdash\) equivalent-states (Some (cswitch (Suc n) s)) t u \land purged-relation u execs execs2 \implies equivalent-states (run n (Some (cswitch (Suc n) s)) execs) (run n t execs2) u 
  { 
    fix t' 
    assume t = Some t'
    fix rs
    assume rs: run (Suc n) (Some s) execs = Some rs
    fix rt
    assume rt: run (Suc n) (Some t') execs2 = Some rt
    assume vpeq-s-t \(\forall \ v. ifp^** v u \implies vpeq v s t'\)
    assume current-s-t: current s = current t'
    assume purged-a-a2: purged-relation u execs execs2
    — The following terminology is used: states rs and rt (for: run-s and run-t) are the states after a run. States ns and nt (for: next-s and next-t) are the states after one step.
    — We prove two properties: the states rs and rt have equal active domains (current-rs-rt) and are vpeq for all domains v that may influence u (vpeq-rs-rt). Both are proven using the IH. To use the IH, we have to prove that the properties hold for the next step (in this case, 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-nt).
    from current-s-t cswitch-independent-of-state
    have current-ns-nt: current (cswitch (Suc n) s) = current (cswitch (Suc n) t') by blast
    from cswitch-consistency vpeq-s-t
    have vpeq-ns-nt: \(\forall \ v. ifp^** v u \implies vpeq v (cswitch (Suc n) s) (cswitch (Suc n) t')\) 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-ss-rt: current rs = current rt using rs rt by(auto) 
    { 
      fix v
      assume ia: ifp^** v u
      from current-ss-rt and this have equivalent-states (Some rs) (Some rt) u by auto
    } 
    thus ?case by(simp add:option.splits,cases t,simp+)
  } 
ext 
  case (4 n execs s t u execs2)
**assume** not-interrupt: ¬interrupt (Suc n)
**assume** thread-empty-s: thread-empty(execs (current s))
**assume** IH: (∀s t u execs2. equivalent-states (Some s) t u ∧ purged-relation u execs execs2 → equivalent-states (run n (Some s) execs) (run n t execs2) u)

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

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

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

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

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

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

\[
\begin{align*}
\text{have equivalent-states (run n (Some s) execs) (run n (Some (step (next-state t' execs2) (next-action t' execs2))) (next-execs t' execs2)) u using rs rt by auto} \\
\text{from t-not-empty t-prec vpeq-s-nt this thread-empty-s not-interrupt} \\
\text{show ?thesis using rs rt by auto}
\end{align*}
\]

next

\[
\begin{align*}
\text{case t-not-prec} \\
\text{thus ?thesis using rt t-not-empty not-interrupt by(auto)}
\end{align*}
\]

qed

\[
\begin{align*}
\text{fix v} \\
\text{assume ia : ifp^** v u} \\
\text{have vpeq v rs rt using \[ vpeq s t u \] unfolding purged-relation-def by auto}
\end{align*}
\]

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

next

\[
\begin{align*}
\text{case t-not-empty} \\
\text{show ?thesis using (cases precondition (next-state t execs2) (next-action t execs2) rule :case-split[case-names t-prec t-not-prec])}
\end{align*}
\]

next-execs t

\[
\begin{align*}
\text{case t-prec} \\
\text{from t-not-empty current-next-state and vpeq-s-t-u and thread-empty-s and purged-a-a2 and current-s-t } \\
\text{have not-ifp-curr-t: ~ifp^** (current (next-state t' execs2))) u unfolding purged-relation-def by(auto)} \\
\text{from t-prec current-next-state locally-respects-next-state this and vpeq-s-t and locally-respects and vpeq-s-nt} \\
\text{have vpeq-s-nt (\'v. ifp^** v u \rightarrow vpeq v s (step (next-state t' execs2) (next-action t' execs2))) by(auto)} \\
\text{from purged-a-na2 this current-s-nt not-ifp-curr-t current-next-state } \\
\text{H[\{\text{where t=Some (step (next-state t' execs2)) (next-action t' execs2)) and u=u and ?execs2.0=next-execs t' execs2}\]} \\
\text{have equivalent-states (run n (Some s) execs) (run n (Some (step (next-state t' execs2) (next-action t' execs2))) u using rs rt by(auto)} \\
\text{from ia t-not-empty t-prec vpeq-s-nt this thread-empty-s not-interrupt} \\
\text{show ?thesis using rs rt by(auto)}
\end{align*}
\]

next

\[
\begin{align*}
\text{case t-not-prec} \\
\text{thus ?thesis using rt t-not-empty not-interrupt by(auto)}
\end{align*}
\]

qed

\[
\begin{align*}
\text{from current-rs-rt and this have equivalent-states (Some rs) (Some rt) u by(auto)}
\end{align*}
\]

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

next

\[
\begin{align*}
\text{case (5 n execs s t u execs2)} \\
\text{assume not-interrupt: ~interrupt (Suc n)} \\
\text{assume thread-not-empty-s: ~thread-empty(execs (current s))} \\
\text{assume not-pred-s: ~precondition (next-state s execs) (next-action s execs)}
\end{align*}
\]

— Whenever the precondition does not hold, the entire theorem flattens to True and everything holds vacuously.
hence run (∀ n) (Some s) execs = None using not-interrupt thread-not-empty-s by simp
thus ?case by(simp add:option.split)
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: (∀ n u execs2,
  equivalent-states (Some (step (next-state s execs) (next-action s execs))) t u ∧
  purged-relation u (next-execs s execs) execs2
  equivalent-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 vpeq-s-t: ∀ v. ifp^∗∗ (current s) u vpeq v s t'
  assume current-s-t: current s = current t'
  assume purged-a-a2: purged-relation u execs execs2

  — The following terminology is used: states rs and rt (for: run-s and run-t) are the states after a run. States ns
  and nt (for: next-s and next-t) are the states after one step.
  — We prove two properties: the states rs and rt have equal active domains (current-rs-rt) and are vpeq for all
  domains v that may influence u (vpeq-rs-rt). Both are proven using the IH. To use the IH, we have to prove that
  the properties hold for the next step (in this case, 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.
  from ifp-reflexive and vpeq-s-t have vpeq-s-t-u vpeq u s t' by auto
  from step-atomicity and current-s-t current-next-state
  have current-ns-nt: current (step (next-state s execs) (next-action s execs)) = current (step (next-state t'
  execs2)) (next-action t' execs2))
  unfolding step-def
  by (cases next-action s execs,cases next-action t' execs2,simp,simp,cases next-action t' execs2,simp,simp)
  from vpeq-s-t have vpeq-curr-s-t: ifp^∗∗ (current s) u vpeq (current s) s t' by auto
  from prec-s involved-ifp[THEN spec,THEN spec,where x1=next-state s execs and s=next-action s execs]
  vpeq-s-t have vpeq-involved: ifp^∗∗ (current s) u(vseq ∧ d ∈ involved (next-action s execs) . vpeq d s t')
  using current-next-state
  unfolding involved-def precondition-def B-def
  by(cases next-action s execs,simp,auto,metis converse-rtranclp-into-rtranclp)
  from current-s-t next-execs-consistent vpeq-curr-s-t vpeq-involved
  have next-execs-t: ifp^∗∗ (current s) u vseq next-execs t' execs = next-execs s execs
  unfolding next-execs-def
  by(auto)
  from current-s-t purged-a-a2 thread-not-empty-s next-action-consistent[THEN spec,THEN spec,where x1=s
  and x=t'] vpeq-curr-s-t vpeq-involved
  have next-action-s-t ifp^∗∗ (current s) u vseq next-action t' execs2 = next-action s execs
  by(unfold next-action-def,unfold purged-relation-def,auto)
  from purged-a-a2 current-s-t next-execs-consistent[THEN spec,THEN spec,THEN spec,where x2=s and x1=t'
  and x=execs]
  vpeq-curr-s-t vpeq-involved
have purged-na-na2: purged-relation u (next-exec s execs) (next-exec s' execs2)
unfolding next-exec-def purged-relation-def
by (auto)
from purged-a-a2 and purged-relation-def and thread-not-empty-s and current-s-t have thread-not-empty-t:
ifp^∗∗ (current s) u → ¬thread-empty(execs2 (current t')) by auto
from step-atomicity current-s-t current-next-state have current-s-t: current (step (next-state s execs) (next-action s execs)) = current t'
unfolding step-def
by (cases next-action s execs,auto)
from step-atomicity and current-s-t have current-s-nt: current s = current (step t' (next-action t' execs2))
unfolding step-def
by (cases next-action t' execs2,auto)
from purged-a-a2 have purged-na-na: ¬ifp^∗∗ (current s) u → purged-relation u (next-exec s execs) execs2
by (unfold next-exec-def,unfold purged-relation-def,auto)

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

have current-rs-rt: current rs = current rt
proof (cases ifp^∗∗ (current s) u rule : case-split[cases-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[cases-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-exec-s 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
IH [where u=u and ?execs2.0 = (next-exec s t' execs2) and t = Some (step (next-state t' execs2) (next-action
t' execs2))]
current-ns-nt purged-na-na2
have equivalent-states (run n (Some (step (next-state s execs) (next-action s execs))) (next-exec s execs))
(run n (Some (step (next-state t' execs2) (next-action t' execs2))) (next-exec t' execs2)) u
by auto
from prec-t thread-not-empty-t prec-s and this and not-interrupt and thread-not-empty-s and next-action-s-t
show ?thesis using rs rt by auto
next
case 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[cases-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[cases-names t-prec
t-not-prec])
case t-prec
from curr-not-ifp-u t-prec IH [where u=u and ?execs2.0 = (next-exec s t' execs2) and t = Some (step
(next-state \( t' \) \texttt{execs2}) \((\text{next-action} \ t' \ \texttt{execs2})\)])
\begin{align*}
\text{current-ns-nt} & \text{next-execs-t} \text{purged-na-na2} \ vpeq-ns-nt-not-ifp-u \ \text{current-s-t} \ vpeq-s-t \ \text{prec-s} \ \text{purged-a-a2} \\
\text{have} & \text{equivalent-states} \ \text{(run n} (\text{some (step (next-state s execs) (next-action s execs))} \text{)} \text{(next-execs s execs)}) \\
\text{by} & \text{auto} \\
\text{from} & \text{this} \ t\text{-prec} \ \text{curr-not-ifp-u} \ t\text{-empty} \ \text{prec-s} \ \text{not-interrupt} \ \text{thread-not-empty-s} \ \text{show} \ ?\text{thesis} \ \text{using} \ rs \ rt \ \text{by} \ \text{auto} \\
\text{next} & \text{case} \ t\text{-not-prec} \\
\text{from} & \text{t\text{-not-prec} t\text{-empty} not-interrupt} \ \text{show} \ ?\text{thesis} \ \text{using} \ rt \ \text{by} \ \text{simp} \\
\text{qed} & \\
\text{next} & \text{case} \ t\text{-empty} \\
\text{from} & \text{curr-not-ifp-u} \ \text{and} \ \text{prec-s} \ \text{and} \ vpeq-s-t \ \text{and} \ \text{locally-respects} \ \text{and} \ vpeq-ns-t \ \text{current-next-state} \ \text{locally-respects-next-state} \\
\text{have} & \text{vpeq-ns-t} \ (\forall \ v. \ \text{ifp^** v u → vpeq v (step (next-state s execs) (next-action s execs)) \( t' \))} \\
\text{by} & \text{blast} \\
\text{from} & \text{curr-not-ifp-u IH} \ [\text{where} \ t=\text{Some \( t' \)} \ \text{and} \ u=\text{u and} \ ?\text{execs2.0=execs2} \] \text{and} \ \text{current-ns-nt} \ \text{and} \ \text{next-execs-t} \\
\text{and} & \text{purged-na-a and} \ vpeq-ns-t \ \text{and} \ \text{this} \\
\text{have} & \text{equivalent-states} \ \text{(run n} (\text{some (step (next-state s execs) (next-action s execs))} \text{)} \text{(next-execs s execs)}) \\
\text{by} & \text{auto} \\
\text{from} & \text{this} \ \text{not-interrupt} \ \text{thread-not-empty-s} \ \text{t-empty} \ \text{prec-s} \ \text{show} \ ?\text{thesis} \ \text{using} \ rs \ rt \ \text{by} \ \text{auto} \\
\text{qed} & \\
\text{next} & \text{case} \ \text{t-prec} \\
\text{have} & \text{thread-not-empty-t: ¬thread-empty(\texttt{execs2 (current \( t' \))}} \text{using} \ \text{thread-not-empty-t curr-ifp-u} \ \text{by} \ \text{auto} \\
\text{from} & \text{current-ns-nt} \ \text{next-execs-t} \ \text{next-action-s-t} \ \text{purged-a-a2} \\
\text{curr-ifp-u} \ \text{t-prec} & \ \text{prec-s} \ vpeq-ns-nt-ifp-u \ [\text{where} \ a=(\text{next-action s execs})] \ \text{vpeq-s-t} \ \text{current-s-t} \\
\text{have} & \text{equivalent-states} \ \text{(some (step (next-state s execs) (next-action s execs))} \text{)} \text{(some (step (next-state t\' execs2)) (next-action t\' execs2))} \\
\text{unfolding} & \text{purged-relation-def next-state-def} \\
\text{by} & \text{auto} \\
\text{from} & \text{this} \\
\text{IH} \ [\text{where} \ u=\text{u and} \ ?\text{execs2.0=(next-execs t\' execs2)} \] \text{and} \ \text{t=Some (step (next-state t\' execs2) (next-action t\' execs2))} \\
\text{current-ns-nt} & \ \text{purged-na-na2} \\
\text{have} & \text{equivalent-states} \ \text{(run n} (\text{some (step (next-state s execs) (next-action s execs))} \text{)} \text{(next-execs s execs)}) \\
\text{by} & \text{auto} \\
\text{from} & \text{ia} \ \text{curr-ifp-u} \ \text{t-prec} \ \text{thread-not-empty-t} \ \text{prec-s} \ \text{and} \ \text{this and} \ \text{not-interrupt} \ \text{and} \ \text{thread-not-empty-s} \ \text{and} \ \text{next-action-s-t} \\
\text{show} & \ ?\text{thesis} \ \text{using} \ rs \ rt \ \text{by} \ \text{auto}
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: \( \forall u. u \in \text{execs} . \text{purged-relation u execs (purge execs u)} \)
by (unfold purged-relation-def, unfold purge-def, auto)

{
  fix execs s t n u
  assume 1: equivalent-states s t u
  from this view-partitioned-inductive purged-relation
  have equivalent-states (run n s execs) (run n t (purge execs u)) u
  unfolding view-partitioned-ind-def by auto
  from this ifp-reflexive
  have equivalent-states (run n s execs) (run n t (purge execs u)) u
  unfolding view-partitioned-def by auto
  have run n s execs ∥ run n t (purge execs u) ⇀ (λrs rt. vpeq u rs rt ∧ current rs = current rt)
  using r-into-rtranclp unfolding B-def
  unfolding view-partitioned-def by auto
  have run n (Some s0) execs ⇀ (λs-f. run n (Some s0) (purge execs (current s-f)) ⇀ (λ{s-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 s= 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
  qed

3.2.2 Security for indirectly interfering domains

Consider the following security policy over three domains A, B and C: A ∼ B ∼ C, but A ∉ 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 $\text{ipurge}$ functions. The first $\text{purges}$ all domains $d$ that are $\text{intermediary}$ 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** $\text{intermediary} :: 'd\text{-}t \Rightarrow 'd\text{-}t \Rightarrow \text{bool}$

**Where** $\text{intermediary} \; d \; u \equiv \exists \; v . \; \text{ifp}^{\ast \ast} \; v \; d \; \land \; \text{ifp} \; d \; u \; \land \; \neg \text{ifp} \; v \; u \; \land \; d \; \neq \; u$

**Primrec** $\text{remove-gateway-communications} :: 'd\text{-}t \Rightarrow 'd\text{-}t \Rightarrow 'd\text{-}t \Rightarrow 'd\text{-}t$ \[
\text{remove-gateway-communications} \; u \; [] = [] \\
\text{if} \; \text{remove-gateway-communications} \; u \; \text{asesq} \; \text{exec} \Rightarrow \text{null} \; \text{else} \; \text{asesq} \; \text{exec} \]

**Definition** $\text{ipurge-l} ::$

\[
\text{ipurge-l} \; \text{execs} \; u \equiv \lambda \; d . \; \text{if} \; \text{intermediary} \; d \; u \; \text{then} \; [] \\
\text{else if} \; d \; = \; u \; \text{then} \\
\text{remove-gateway-communications} \; u \; \text{execs} \; u \\
\text{else} \; \text{execs} \; d
\]

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

**Abbreviation** $\text{ind-source} :: 'd\text{-}t \Rightarrow 'd\text{-}t \Rightarrow \text{bool}$

**Where** $\text{ind-source} \; d \; u \equiv \text{ifp}^{\ast \ast} \; d \; u \; \land \; \neg \text{ifp} \; d \; u$

**Definition** $\text{ipurge-r} ::$

\[
\text{ipurge-r} \; \text{execs} \; u \equiv \lambda \; d . \; \text{if} \; \text{intermediary} \; d \; u \; \text{then} \; [] \\
\text{else if} \; \text{ind-source} \; d \; u \; \text{then} \\
\text{SOME} \; \alpha \; . \; \text{realistic-execution} \; \alpha \\
\text{else if} \; d \; = \; u \; \text{then} \\
\text{remove-gateway-communications} \; u \; \text{execs} \; u \\
\text{else} \\
\text{execs} \; d
\]

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

**Definition** $\text{NI-indirect-sources} :: \text{bool}$

**Where** $\text{NI-indirect-sources} \equiv \forall \; a \; n . \; \text{run} \; n \; \text{(Some} \; s0) \; \text{execs} \; \rightarrow \\
(\lambda \; s \cdot . \; \text{run} \; n \; \text{(Some} \; s0) \; \text{ipurge-l} \; \text{execs} \; \text{(current} \; s-f) \; \text{)} \; \| \\
\text{run} \; n \; \text{(Some} \; s0) \; \text{ipurge-r} \; \text{execs} \; \text{(current} \; s-f) \; ) \; \rightarrow \\
(\lambda \; s \cdot . \; \text{output-f} \; s \cdot \; a \; = \; \text{output-f} \; s \cdot \; a))$

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

This allows us to define security over intransitive policies.

**Definition** $\text{isecure} :: \text{bool}$

**Where** $\text{isecure} \equiv \text{NI-indirect-sources} \; \land \; \text{NI-unrelated}$

**Abbreviation** $\text{iequivalent-states} :: '\text{state-t option} \; \Rightarrow '\text{state-t option} \; \Rightarrow 'd\text{-}t \Rightarrow \text{bool}$
where \( iequivalent-states\ s\ t\ u\ \equiv\ s\ \parallel\ t\ \xrightarrow{} (\lambda\ s\ t\ .\ (\forall\ v\ .\ ifp\ v\ u\ \land\ \neg\ intermediary\ v\ u\ \xrightarrow{} vpeq\ v\ s\ t)\ \land\ current\ s\ =\ current\ t)\)

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

definition iview-partitioned\ :: bool
where \( iview-partitioned\ \equiv\ \forall\ execs\ ms\ mt\ n\ u\ .\ iequivalent-states\ ms\ mt\ u\ \xrightarrow{} (run\ n\ ms\ (ipurge-l\ execs\ u))\ \parallel\ run\ n\ mt\ (ipurge-r\ execs\ u)\ \xrightarrow{} (\lambda\ rs\ rt\ .\ vpeq\ u\ rs\ rt\ \land\ current\ rs\ =\ current\ rt))\)

definition ipurged-relation1\ :: dom-t \Rightarrow dom-t \Rightarrow action-t\ \Rightarrow dom-t \Rightarrow action-t\ \Rightarrow bool
where \( ipurged-relation1\ u\ execs1\ execs2\ \equiv\ \forall\ d\ .\ ifp\ d\ u\ \land\ intermediary\ d\ u\ \xrightarrow{} execs1\ d\ =\ execs2\ d\ \land\ (intermediary\ d\ u\ \xrightarrow{} execs1\ d\ =\ []))\)

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

lemma vpeq-involved-domains:
assumes ifp-curr: ifp (current s) u
and not-intermediary-curr: ~intermediary (current s) u
and no-gateway-comm: does-not-communicate-with-gateway u execs
and vpeq-s-t: \( \forall\ v\ .\ ifp\ v\ u\ \land\ ~intermediary\ v\ u\ \xrightarrow{} vpeq\ v\ s\ t'\)
and prec-s: precondition (next-state s execs) (next-action s execs)
shows \( \forall\ d\ \in\ involved\ (next-action\ s\ execs)\ .\ vpeq\ d\ s\ t'\)
proof:
  \{ fix \ v
  assume involved: \ v\ \in\ involved\ (next-action\ s\ execs)
  from this prec-s involved-ifp[THEN spec,THEN spec,where x1=next-state s execs and x=next-action s execs]
  have ifp-v-curr: ifp v (current s)
  using current-next-state
  unfolding involved-def precondition-def B-def
  by (cases next-action s execs.auto)
  have vpeq v s t'
  proof:
  \{ assume ifp v u \land ~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-w current s = u
  using rtranclp-trans r-into-rtranclp
  by (metis intermediary-def)
  from curr-is-u next-action-from-execs[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 \models \neg ifp v u
  from ifp-curr not-intermediary-curr ifp-v-curr intermediary-v
  have False unfolding intermediary-def by auto
  hence vpeq v s t' by auto
}
ultimately
show vpeq v s t' unfolding intermediary-def by auto
qed
}
thus \?thesis by auto
qed

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

lemma ipurge-l-removes-gateway-communications:
shows does-not-communicate-with-gateway u (ipurge-l execs u)
proof-
{
  fix aseq u execs a v
  assume 1 : aseq \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 \implies v \notin involved (Some a)
      by (induct exec, auto)
  }
  from 1 2 3 this show \?thesis by metis
  qed
}
from this
show \?thesis
unfolding does-not-communicate-with-gateway-def ipurge-l-def actions-in-execution-def
by auto
qed

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

lemma iunwinding_implies_view_partitioned1:
shows iview-partitioned
proof-
{
  fix u execs execs2 s t n
  have does-not-communicate-with-gateway u execs \land iequivalent-states s t u \land ipurged-relation1 u execs execs2
    \implies iequivalent-states (run n s execs) (run n t execs2) u
  proof (induct n s execs arbitrary: t u execs2 rule: run.induct)
    case (1 s execs t u execs2)
      show \?case by auto
    next
    case (2 n execs t u execs2)
      show \?case by simp
    next
    case (3 n s execs t u execs2)
  }
assume interrupt-s: interrupt (Suc n)
assume IH: (\(\forall t u execs2. does-not-communicate-with-gateway u execs \land
  inequivalent-states (Some (cswitch (Suc n) s)) t u \land
  ipurged-relation1 u execs execs2 \rightarrow
  inequivalent-states (run n (Some (cswitch (Suc n) s)) execs) (run n t execs2) u) \}
  \begin{itemize}
    \item fix t' = 'state-t
    \item assume t = Some t'
  \end{itemize}
fix rs
assume rs: run (Suc n) (Some s) execs = Some rs
fix rt
assume rt: run (Suc n) (Some t') execs2 = Some rt

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

from current-s-t cswitch-independent-of-state
have current-ns-nt: current (cswitch (Suc n) s) = current (cswitch (Suc n) t')
by blast
from cswitch-consistency vpeq-s-t
have vpeq-ns-nt: \(\forall v . ifp v u \land \neg \text{intermediary} v u \rightarrow
  vpeq v (cswitch (Suc n) s) (cswitch (Suc n) t') \)
by auto
from no-gateway-comm current-ns-nt vpeq-ns-nt interrupt-s vpeq-reflexve current-s-t purged-a-a2 IH[where
u=u and t=Some (cswitch (Suc n) t') and ?execs2.0=execs2] \begin{itemize}
  \item have current-rs-rt: current rs = current rt using rs rt by(auto) \}
  \item fix v
  \item assume ia: ifp v u \land \neg \text{intermediary} v u
  \item from no-gateway-comm interrupt-s current-ns-nt vpeq-ns-nt vpeq-reflexive ia current-s-t purged-a-a2 IH[where
u=u and t=Some (cswitch (Suc n) t') and ?execs2.0=execs2] \begin{itemize}
  \item have vpeq v rs rt using rs rt by(auto) \}
  \item from current-rs-rt and this have inequivalent-states (Some rs) (Some rt) u by auto \}
  \item thus \(\forall t' \) case by(simp add:option.splits,cases t,simp+)
next
case (\(\forall n execs s t u execs2\)
assume not-interrupt: \neg interrupt (Suc n)
assume thread-empty-s: thread-empty(execs (current s))

assume IH: (\(\forall t u execs2. does-not-communicate-with-gateway u execs \land
  inequivalent-states (Some s) t u \land
  ipurged-relation1 u execs execs2 \rightarrow
  inequivalent-states (run n (Some s) execs) (run n t execs2) u) \}
  \begin{itemize}
    \item fix t'
  \end{itemize}
assume t: t = Some t'
fix rs
assume rs: run (Suc n) (Some s) execs = Some rs
fix rt
assume rt: run (Suc n) (Some t') execs2 = Some rt

assume no-gateway-comm: does-not-communicate-with-gateway u execs
assume vpeq-s-t: \(\forall v . ifp v u \land \neg \text{intermediary} v u \rightarrow
  vpeq v s t' \)
assume current-s-t: current s = current t'
assume purged-a-a2: ipurged-relation1 u execs execs2
from ifp-reflexive vpeq-s-t have vpeq-u-s-t: vpeq u s t' unfolding intermediary-def by auto
from step-atomicity current-next-state current-s-t have current-s-t: current s = current (step (next-state t' execs2) (next-action t' execs2)) unfolding step-def by (cases next-action s execs,cases next-action t' execs2,simp,simp,cases next-action t' execs2,simp,simp)
from vpeq-s-t have vpeq-curr-s-t: ifp (current s) u ∧ ¬intermediary (current s) u → 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 have iequivalent-states (run (Suc n) (Some s) execs) (run (Suc n) (Some t') execs2) u
proof (cases thread-empty-s) have ?thesis using rs rt by(auto)
from this not-interrupt False next-execs t 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: ¬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: ind-source (current t') u ∨ unrelated (current t') u unfolding ipurged-relationI-def intermediary-def by auto
{ fix v
  assume ifp-v: ifp v u
  assume v-not-intermediary: ¬intermediary v u

  from I ifp-v v-not-intermediary have not-ifp-curr-v: ¬ifp (current t') v unfolding intermediary-def by auto
  from not-ifp-curr-v prec-t locally-respects[THEN spec,THEN spec,THEN spec,where x1=next-state t' execs2 and x=v and x2=the (next-action t' execs2)]
current-next-state vpeq-reflexive have vpeq v (next-state t' execs2) (step (next-state t' execs2) (next-action t' execs2)) unfolding step-def precondition-def B-def by (cases next-action t' execs2 auto)
from this vpeq-transitive not-ifp-curr-v locally-respects-next-state have vpeq-t-nt: 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-nt 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: ∀ v . ifp v u ∧ ¬intermediary v u → 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: ipurged-relationI u execs (next-execs t' execs2) unfolding ipurged-relation1-def next-execs-def by(auto)
from vpeq-ns-nt no-gateway-comm and IH[where t=Some (step (next-state t' execs2) (next-action t' execs2)) and ?execs2.0=(next-execs t' execs2) and u=u]
and current-s-nt purged-a-na2
have eq-ns-nt: inequivalent-states (run n (Some s) execs)
  (run n (Some (step (next-state t' execs2) (next-action t' execs2))) (next-execs t'))

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-nts by simp
  thus ?case by(simp add-option.splits,cases t,simp+)
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 prev-s: precondition (next-state s execs) (next-action s execs)
  assume IH: (\u. \u execs2, does-not-communicate-with-gateway u (next-execs s execs) \u inequivalent-states (Some (step (next-state s execs) (next-action s execs))) (next-execs s execs) execs2 \u inequivalent-states
                  (run n (Some (step (next-state s execs) (next-action s execs))) (next-execs s execs)) (next-execs 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: \u. ifp u \u \u intermediary v u \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 \u intermediary (current s) u \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 \u intermediary (current s) u \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
from current-s-t next-execs-consistent[THEN spec,THEN spec,THEN spec,where x2=s and x1=t′ and s=execs]
  vpeq-curr-s-t vpeq-involved
  have next-execs-t: ifp (current s) u ∧ ¬intermediary (current s) u → next-execs s execs = next-execs s execs
  by (auto simp add: next-execs-def)
from current-s-t and purged-a-a2 and thread-not-empty-s next-action-consistent[THEN spec,THEN spec,where x1=s and x=t′] vpeq-curr-s-t vpeq-involved
  have next-action-s-t: ifp (current s) u ∧ ¬intermediary (current s) u → next-action t′ execs2 = next-action s execs
  by (unfold next-action-def, unfold ipurged-relation1-def, auto)
from purged-a-a2 and thread-not-empty-s and current-s-t
  have next-state s execs = s from current-s-t next-execs-consistent
  and iequivalent-states have no-gateway-comm-na
    proof
      assume precs: precondition (next-state s execs) a ∧ precondition (next-state t′ execs) a
      assume ifp-curr: ifp (current s) u ∧ ¬intermediary (current s) u from ifp-curr precs
      next-state-consistent[THEN spec,THEN spec,where x1=s and x=t′] vpeq-curr-s-t vpeq-s-t
      current-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 ∀ v . ifp v u ∧ ¬intermediary v u → vpeq v (step (next-state s execs) a) (step (next-state t′ execs) a)
      unfolding step-def precondition-def B-def
      by (cases a, auto)
qed
have no-gateway-comm-na: does-not-communicate-with-gateway u (next-execs s execs)
  proof
  { fix a
    assume a ∈ actions-in-execution (next-execs s execs u)
    from this no-gateway-comm[unfolded does-not-communicate-with-gateway-def, THEN spec, where x=a]
      next-execs-subset[THEN spec,THEN spec,THEN spec,where x2=s and x1=execs and x0=u]
      have ∀ v . intermediary v u → v ∈ involved (Some a)
      unfolding actions-in-execution-def
      by (auto)
  }
thus ?thesis unfolding does-not-communicate-with-gateway-def by auto
qed
have iequivalent-states (run (Suc n) (Some s) execs) (run (Suc n) (Some t′) execs2) u
  proof (cases ifp (current s) u ∧ ¬intermediary (current s) u rule : case-split [case-names T F])
case T
  show ?thesis
  proof (cases thread-empty (execs2 (current t′)) rule : case-split [case-names T2 F2])
case F2
  show ?thesis
  proof (cases precondition (next-state t′ execs2) (next-action t′ execs2) rule : case-split [case-names T3 F3])
case T3
  from T purged-a-a2 current-s-t
     next-execs-consistent[THEN spec,THEN spec,where x1=s and x=t′] vpeq-curr-s-t vpeq-involved
     have purged-na-na2: ipurged-relation1 u (next-execs s execs) (next-execs t′ execs2)
unfolding ipurged-relation1-def next-execs-def
by auto
from IH[where t=Some (step (next-state t' execs2) (next-action t' execs2)) and ?execs2.0=next-execst'
execs2 and w=u]
purged-na-na2 current-ns-nt vpeq-ns-nt-1[where a=(next-action s execs)] T T3 prec-s
next-action-s-t eq-execsexecs current-s-t no-gateway-comm-na
have eq-ns-nt inequivalent-states (run n (Some (step (next-state s execs) (next-action s execs))) (next-execsexecs))

(far 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 & ~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 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 ∨ unrelated (current s) u ∨ intermediary (current s) u
unfolding intermediary-def
by auto
from purged-a-a2 and thread-not-empty-s
have 2: ~intermediary (current s) u unfolding ipurged-relation1-def by auto

let ?nt = if thread-empty(execs2 (current t')) then t' else step (next-state t' execs2) (next-action t' execs2)
let ?na2 = if thread-empty(execs2 (current t')) then execs2 else next-execsexecs2

have prec-t: ~thread-empty(execs2 (current t')) ===> precondition (next-state t' execs2) (next-action t'
execs2)
proof−
assume thread-not-empty-t: ~thread-empty(execs2 (current t'))
{
assume not-prec-t: ~precondition (next-state t' execs2) (next-action t' execs2)
hence 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** \( \text{ifp}\ v: \text{ifp}\ v\ u \)

**assume** \( v\text{-not-intermediary}: \neg\text{intermediary} v\ u \)

**have** \( \text{not-ifp-curr-v}: \neg\text{ifp}(\text{current}\ s)\ v \)

**proof**

**assume** \( \text{ifp-curr-v}: \text{ifp}(\text{current}\ s)\ v \)

thus \( \text{False} \)

**proof**–

\( \begin{array}{l}
\quad \text{assume } \text{ind-source}(\text{current}\ s)\ u \\
\quad \text{from } \text{this } \text{ifp-curr-v} \text{ ifp-v unfolding intermediary-def by auto} \\
\quad \text{from } \text{this } v\text{-not-intermediary } \text{have } \text{False unfolding intermediary-def by auto} \\
\end{array} \)

moreover

\( \begin{array}{l}
\quad \text{assume } \text{unrelated}: \text{unrelated}(\text{current}\ s)\ u \\
\quad \text{from } \text{this } \text{ifp-v} \text{ ifp-curr-v have False using rtranclp-trans r-into-rtranclp by metis} \\
\end{array} \)

ultimately show ?thesis using 1 2 by auto

qed

qed

from \( \text{this current-next-state}[\text{THEN spec.THEN spec, where } x1=s\ and\ x=execs] \text{ prec-s locally-respects}[\text{THEN spec.THEN spec, where } x=\text{next-state s execs}] \text{ vpeq-reflexive} \)

**have** \( \text{vpeq v (next-state s execs) (step (next-state s execs) (next-action s execs))} \)

**unfolding** step-def precondition-def B-def

by (cases next-action s execs,auto)

from \( \text{not-ifp-curr-v this locally-respects-next-state vpeq-transitive} \)

**have** \( \text{vpeq-s/ns: vpeq v s (step (next-state s execs) (next-action s execs))} \)

by blast

from \( \text{not-ifp-curr-v current-s-t current-next-state}[\text{THEN spec.THEN spec, where } x1=t’\ and\ x=execs2]) \text{ prec-t locally-respects}[\text{THEN spec.THEN spec,where } x=\text{next-state t’ execs2}] \)

**F** \( \text{vpeq-reflexive} \)

**have** \( \text{0}: \neg\text{thread-empty (execs2 (current t’)) } \rightarrow \text{vpeq v (next-state t’ execs2) (step (next-state t’ execs2) (next-action t’ execs2))} \)

**unfolding** step-def precondition-def B-def

by (cases next-action t’ execs2,auto)

from \( \text{0 not-ifp-curr-v current-s-t locally-respects-next-state}[\text{THEN spec.THEN spec.THEN spec,where } x2=t’\ and\ x1=v\ and\ x=execs2] \)

**vpeq-transitive**

**have** \( \text{vpeq-t-nt: \neg thread-empty (execs2 (current t’)) } \rightarrow \text{vpeq v t’ (step (next-state t’ execs2) (next-action t’ execs2))} \) by metis

from \( \text{this vpeq-reflexive} \)

**have** \( \text{vpeq-t-nt: vpeq v t’ ?nt} \)

by auto

from \( \text{vpeq-s-t ifp-v v-not-intermediary} \)

**have** \( \text{vpeq v s t’ by auto} \)

from \( \text{this vpeq-s-ns vpeq-t-nt vpeq-transitive vpeq-symmetric vpeq-reflexive} \)

**have** \( \text{vpeq v (step (next-state s execs) (next-action s execs)) ?nt} \)

by (metis (opaque-lifting, no-types))

\} \)

hence \( \text{vpeq-ns-nt: } \forall\ v.\ \text{ifp}\ v\ u\ \neg\text{intermediary} v\ u \rightarrow \text{vpeq v (step (next-state s execs) (next-action s execs)) ?nt} \) by auto

from \( \text{vpeq-s-t 2 F purged-a-a2 current-s-t thread-not-empty-s have purged-na-na2: ipurged-relation1 u (next-execs s execs) ?na2} \)

**unfolding** purged-relation1-def next-execs-def intermediary-def by(auto)

from \( \text{current-ns-nt current-ns-t current-next-state have current-ns-nt} : \)
current \((\text{step} \ (\text{next-state} \ s \ \text{execs}) \ (\text{next-action} \ s \ \text{execs})) = \text{current} \ ?\nt
\) by auto
from \(\text{prec-s vpeq-ns-nt no-gateway-comm-na}
\) and \(\text{IH[where} \ t=\text{Some} \ ?\nt \ and \ ?\execs2.0=\?na2 \ and \ u=u]\)
and \(\text{current-ns-nt purged-na-na2}
\) have \(\text{eq-ns-nt} \equiv \text{iequivalent-states} \ (run \ n \ (\text{Some} \ (\text{step} \ (\text{next-state} \ s \ \text{execs}) \ (\text{next-action} \ s \ \text{execs})) \ (\text{next-exec}s \ s \ \text{execs}))
\)
\((\text{run} \ n \ (\text{Some} \ ?\nt) \ ?\na2) \ u \ \text{by} \ \text{auto}
\)
from \(\text{this not-interrupt thread-not-empty-s prec-t prec-s}
\) have \(\text{current-rs-rt} \equiv \text{current} \ rs = \text{current} \ rt \ \text{using} \ \text{rs} \ rt \ \text{by} \ (\text{cases thread-empty} \ \text{execs2} \ \text{execs} \ \text{current} \ t') \\),simp,simp\)
\{
\}
fix \(v \\)
assume \(i\text{a} : \text{ifp} \ v \ u \ \land \neg \text{intermediary} \ v \ u
\)
from \(\text{this eq-ns-nt not-interrupt thread-not-empty-s prec-t}
\) have \(\text{vpeq} \ v \ rs \ rt
\) using \(\text{rs} \ rt \ \text{by} \ (\text{cases thread-empty} \ \text{execs2} \ \text{execs} \ \text{current} \ t') \\),simp,simp\)
\}
from \(\text{current-rs-rt} \ and \ this \ show \ \?\text{thesis} \ \text{using} \ \text{rs} \ rt \ \text{by} \ \text{auto}
\)
qd
\}
thus \(\?\text{case by}(\text{simp add:option.splits,cases t,simp+})
\)
qed
\}
hence \(\text{iview-partitioned-inductive} : \\forall \ u \ s \ t \ \text{execs} \ \text{execs2} \ n. \ \text{does-not-communicate-with-gateway} \ u \ \text{execs} \ \land \ \text{iequivalent-states} \ s \ t \ u \ \land \ \text{ipurged-relation1} \ u \ \text{execs} \ \text{execs2} \ \longrightarrow \ \text{iequivalent-states} \ (\text{run} \ n \ s \ \text{execs}) \ (\text{run} \ n \ t \ \text{execs2}) \ u
\)
by blast
have \(\text{ipurged-relation} : \\forall \ u \ \text{execs} . \ \text{ipurged-relation1} \ u \ (\text{ipurge-l} \ \text{execs} \ u) \ (\text{ipurge-r} \ \text{execs} \ u)
\)
by(\text{unfold \text{ipurged-relation1-def},unfold \text{ipurge-l-def},unfold \text{ipurge-r-def},auto})
\{
\}
fix \(\text{execs} \ s \ t \ n \ u \\)
assume \(\text{I} : \text{iequivalent-states} \ s \ t \ u
\)
from \(\text{ifp-reflexive}
\) have \(\text{dir-source} : \\forall \ u . \ \text{ifp} \ u \ u \ \land \neg \text{intermediary} \ u \ u
\) unfolding \text{intermediary-def} by auto
from \(\text{ipurge-l-removes-gateway-communications}
\) have \(\text{does-not-communicate-with-gateway} \ u \ (\text{ipurge-l} \ \text{execs} \ u)
\)
by auto
from \(\text{I} \ \text{this \ iview-partitioned-inductive \ ipurged-relation}
\) have \(\text{iequivalent-states} \ (\text{run} \ n \ s \ (\text{ipurge-l} \ \text{execs} \ u)) \ (\text{run} \ n \ t \ (\text{ipurge-r} \ \text{execs} \ u)) \ u \ \text{by} \ \text{auto}
\)
from \(\text{this \ dir-source}
\) have \(\text{run} \ n \ s \ (\text{ipurge-l} \ \text{execs} \ u) \parallel \text{run} \ n \ t \ (\text{ipurge-r} \ \text{execs} \ u) \ \rightarrow \ (\lambda \text{rs} \ rt. \ vpeq \ u \ rs \ rt \ \land \ \text{current} \ rs = \ \text{current} \ rt)
\)
using \(\text{r-into-rtranclp unfolding} \ B-def
\) by(\text{cases \ run} \ n \ s \ (\text{ipurge-l} \ \text{execs} \ u)\),simp,cases \ run \ n \ t \ (\text{ipurge-r} \ \text{execs} \ u)\),simp,auto\)
\}
thus \(\?\text{thesis} \ \text{unfolding} \ \text{iview-partitioned-def} \ \text{Let-def} \ \text{by} \ \text{auto}
\)
qd

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

**Definition** \(\text{mcurrents} \equiv \text{'}state-t \ option \Rightarrow \text{'}state-t \ option \Rightarrow \text{bool}
\)
where \(\text{mcurrents} \ m1 \ m2 \equiv m1 \parallel m2 \rightarrow (\lambda s \ t. \ \text{current} \ s = \ \text{current} \ t)
\)

Proof that switching/interrupts are purely time-based and happen independent of the actions done by the domains. As all theorems in this locale, it holds vacuously whenever one of the states is None, i.e., whenever at some point a precondition does not hold.
lemma current-independent-of-domain-actions:
assumes current-s-t: mcurrents s t
shows mcurrents (run n s execs) (run n t execs2)
proof-
{
  fix n s execs t execs2
  have mcurrents s t --- mcurrents (run n s execs) (run n t execs2)
  proof (induct n s execs arbitrary: t execs2 rule: run.induct)
    case (1 s execs t execs2)
      from this show ?case using current-s-t unfolding B-def by auto
    next
    case (2 n execs t execs2)
      show ?case unfolding mcurrents-def by(auto)
    next
    case (3 n s execs t execs2)
      assume interrupt: interrupt (Suc n)
      assume IH: (\(\forall t \in \text{execs2} \cdot \text{mcurrents} (\text{Some} (\text{cswitch} (\text{Suc} n) s)) t \longrightarrow \text{mcurrents} (\text{run} n (\text{Some} (\text{cswitch} (\text{Suc} n) s)) s) \text{ execs} \) (run n t execs2))
      {
        fix t'
        assume t: t = (Some t')
        assume curr: mcurrents (Some s) t
        from t curr cswitch-independent-of-state\[\text{THEN spec,}\ \text{THEN spec,}\ \text{THEN spec,}\ \text{where xI=x} = s\] have current-as-nt: current (cswitch (Suc n) s) = current (cswitch (Suc n) t')
        unfolding mcurrents-def by simp
        from current-ns-nt IH[\text{where t=Some (cswitch (Suc n) t')} \text{ and } \text{execs2.0=execs2}] have mcurrents-ns-nt: mcurrents (run n (Some (cswitch (Suc n) s)) execs) (run n (Some (cswitch (Suc n) t')) execs2)
        unfolding mcurrents-def by(auto)
        from mcurrents-ns-nt interrupt t
        have mcurrents (run (Suc n) (Some s) execs) (run (Suc n) t execs2)
        unfolding mcurrents-def B2-def B-def by(cases run n (Some (cswitch (Suc n) s)) execs, cases run (Suc n) t execs2,auto)
      }
  thus ?case unfolding mcurrents-def B2-def by(cases t,auto)
  next
  case (4 n execs s t execs2)
  assume not-interrupt: ~interrupt (Suc n)
  assume thread-empty-s: thread-empty(execs (current s))
  assume IH: (\(\forall t \in \text{execs2} \cdot \text{mcurrents} (\text{Some} s) t \longrightarrow \text{mcurrents} (\text{run} n (\text{Some} s) \text{ execs}) \) (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[\text{where } \text{execs2.0=execs2 and } t=\text{Some t}'] have mcurrents (run (Suc n) (Some s) execs) (run (Suc n) t execs2)
        by auto
    }
  }
  moreover
  {
    assume not-prec-t: ~thread-empty(execs2 (current t')) \land ~precondition (next-state t' execs2) \land (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
\{
  \text{assume } \text{step-}\neg\text{-thread-empty}(\text{execs}_2 (\text{current } t')) \land \text{precondition} (\text{next-state } t' \text{ execs}_2) (\text{next-action } t' \text{ execs}_2)
\}
\]
\text{have } \text{mcurrents} (\text{Some } s) (\text{Some } (\text{step} (\text{next-state } t' \text{ execs}_2) (\text{next-action } t' \text{ execs}_2)))
\]
\text{using } \text{step-atomicity } \text{curr } t \text{ current-next-state unfolding } \text{mcurrents-def}
\]
\text{unfolding } \text{step-def}
\]
\text{by } (\text{cases next-action } t' \text{ execs}_2, \text{auto})
\]
\text{from } t \text{ step-}\neg\text{-interrupt } \text{thread-empty-s this } \text{IH}[\text{where } ?\text{execs}_2.0=\text{next-execs } t' \text{ execs}_2 \text{ and } t=\text{Some (step (next-state } t' \text{ execs}_2) (\text{next-action } t' \text{ execs}_2))]
\]
\text{have } \text{mcurrents} (\text{run } (\text{Suc } n) (\text{Some } s) \text{ execs}) (\text{run } (\text{Suc } n) t \text{ execs}_2)
\]
\text{by } \text{auto}
\}
\]
ultimately \text{have } \text{mcurrents} (\text{run } (\text{Suc } n) (\text{Some } s) \text{ execs}) (\text{run } (\text{Suc } n) t \text{ execs}_2) \text{ by blast}
\}
\]
thus \text{case unfolding } \text{mcurrents-def } B2\text{-def by}(\text{cases } t, \text{auto})
\]
next
case (5 \text{n execs } s t \text{ execs}_2)
\]
\text{assume } \text{not-interrupt-s: } \neg\text{-interrupt } (\text{Suc } n)
\]
\text{assume } \text{thread-not-empty-s: } \neg\text{-thread-empty}(\text{execs } (\text{current } s))
\]
\text{assume } \text{prec-s: } \text{precondition } (\text{next-state } s \text{ execs}) (\text{next-action } s \text{ execs})
\]
\text{assume } \text{IH}: (\land \text{execs}_2).
\]
\text{mcurrents } (\text{Some } (\text{step} (\text{next-state } s \text{ execs}) (\text{next-action } s \text{ execs}))) t \rightarrow
\]
\text{mcurrents } (\text{run } n (\text{Some } (\text{step} (\text{next-state } s \text{ execs}) (\text{next-action } s \text{ execs}))) (\text{next-execs } s \text{ execs})) (\text{run } n t \text{ execs}_2))
\}
\}
\text{fix } t'
\]
\text{assume } t: t = (\text{Some } t')
\]
\text{assume } \text{curr: } \text{mcurrents } (\text{Some } s) t
\}
\]
\text{assume } \text{thread-empty-t: } \neg\text{-thread-empty}(\text{execs}_2 (\text{current } t'))
\]
\text{have } \text{mcurrents } (\text{Some } (\text{step} (\text{next-state } s \text{ execs}) (\text{next-action } s \text{ execs}))) (\text{Some } t')
\]
\text{using } \text{step-atomicity } \text{curr } t \text{ current-next-state unfolding } \text{mcurrents-def}
\]
\text{unfolding } \text{step-def}
\]
\text{by } (\text{cases next-action } s \text{ execs}, \text{auto})
\]
\text{from } t \text{ curr not-interrupt } \text{thread-not-empty-s prec-s thread-empty-t this } \text{IH}[\text{where } ?\text{execs}_2.0=\text{execs}_2 \text{ and } t=\text{Some } t']
\]
\text{have } \text{mcurrents} (\text{run } (\text{Suc } n) (\text{Some } s) \text{ execs}) (\text{run } (\text{Suc } n) t \text{ execs}_2)
\]
\text{by } \text{auto}
\}
\]
moreover
\{
  \text{assume } \text{not-prec-t: } \neg\text{-thread-empty}(\text{execs}_2 (\text{current } t')) \land \neg\text{precondition } (\text{next-state } t' \text{ execs}_2) (\text{next-action } t' \text{ execs}_2)
\}
\]
\text{from } t \text{ this not-interrupt}
\]
\text{have } \text{mcurrents} (\text{run } (\text{Suc } n) (\text{Some } s) \text{ execs}) (\text{run } (\text{Suc } n) t \text{ execs}_2)
\]
\text{unfolding } \text{mcurrents-def } B2\text{-def by } (\text{auto})
\}
\]
moreover
{ assume step-t: ¬thread-empty(execs2 (current t′)) ∧ precondition (next-state t′ execs2) (next-action t′ execs2)
    have mcurrents (Some (step (next-state s execs) (next-action s execs))) (Some (step (next-state t′ execs2) (next-action t′ execs2)))
      using step-atomicity curr t current-next-state unfolding mcurrents-def
      unfolding step-def
      by (cases next-action s execs, simp, cases next-action t′ execs2, simp, simp, cases next-action t′ execs2, simp, simp)
      from current-next-state t step-1 curr not-interrupt thread-not-empty-s prec-s this IH [where ?execs2.0=next-execs t′ execs2 and t=Some (step (next-state t′ execs2) (next-action t′ execs2))]
      have mcurrents (run (Suc n) (Some s) execs) (run (Suc n) t execs2)
        by auto
    } ultimately have mcurrents (run (Suc n) (Some s) execs) (run (Suc n) t execs2) by blast
    thus ?thesis using current-s-t by auto qed

theorem unwinding-implies-NI-indirect-sources:
shows NI-indirect-sources
proof−
{ fix execs a n
  from iunwinding-implies-view-partitioned1
  have vp: iview-partitioned by blast
  from vp and vpeq-reflexive
  have I: ∀ u . run n (Some s0) (ipurge-l execs u) ∥ run n (Some s0) (ipurge-r execs u) → (λrs rt. vpeq u rs rt ∧ current rs = current rt)
    unfolding iview-partitioned-def by auto
  have run n (Some s0) execs → (λs-f. run n (Some s0) (ipurge-l execs (current s-f)) ∥
    run n (Some s0) (ipurge-r execs (current s-f)) → (λs-l s-r. output-f s-l a = output-f s-r a))
  proof(cases run n (Some s0) execs)
    case None thus ?thesis unfolding B-def by simp
    next
    case (Some s-f)
    thus ?thesis
    proof(cases run n (Some s0) (ipurge-l execs (current s-f)))
    case None
    from Some this show ?thesis unfolding B-def by simp
    next
    case (Some s-ipurge-l)
    show ?thesis
    proof(cases run n (Some s0) (ipurge-r execs (current s-f)))
    case None
    from ‹run n (Some s0) execs = Some s-f› Some this show ?thesis unfolding B-def by simp
    next
    case (Some s-ipurge-r)
    from cswitch-independent-of-state
    ‹run n (Some s0) execs = Some s-f› ‹run n (Some s0) (ipurge-l execs (current s-f)) = Some s-ipurge-l›
    current-independent-of-domain-actions[where n=n and s=Some s0 and t=Some s0 and execs=execs and
    ?execs2.0=(ipurge-l execs (current s-f))]
  
EURO-MILS
have 2: current s-ipurge-l = current s-f
unfolding mcurrents-def B-def by auto
from ‹run n (Some s0) execs = Some s-f› ‹run n (Some s0) (ipurge-l execs (current s-f)) = Some s-ipurge-l›
have vpeq (current s-f) s-ipurge-l s-ipurge-r ∧ current s-ipurge-l = current s-ipurge-r
unfolding B-def by auto
from this 2 have output-f s-ipurge-l a = output-f s-ipurge-r a
using output-consistent by auto
from ‹run n (Some s0) execs = Some s-f› ‹run n (Some s0) (ipurge-l execs (current s-f)) = Some s-ipurge-l›
this Some
show ?thesis unfolding B-def by auto
qed
qed
qed}
thus ?thesis unfolding NI-indirect-sources-def by auto
qed

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

3.3 ISK (Interruptible Separation Kernel)
theory ISK
imports SK
begin

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

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

\[ \gamma = [COPY_INIT, COPY_CHECK, COPY_COPY] \]

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

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

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

- “AS_precondition s d COPY_INIT”
since COPY_INIT is the start of an action sequence.

- “AS_precondition (step s COPY_INIT) d COPY_CHECK”
since (COPY_INIT, COPY_CHECK) is a sub sequence.

- “AS_precondition (step s COPY_CHECK) d COPY_COPY”
since (COPY_CHECK, COPY_COPY) is a sub sequence.
Additionally, the precondition for domain \( d \) must be consistent when a context switch occurs, or when ever some other domain \( d' \) performs an action.

Locale Interruptible_Separation_Kernel refines locale Separation_Kernel in two ways. First, there is a definition of realistic executions. A realistic trace consists of action sequences from \( \text{AS-set} \).

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.

```ini
locale Interruptible-Separation-Kernel = Separation-Kernel kstep output-f s0 current cswitch interrupt kprecondition realistic-execution control kinvolved ifp vpeq
for kstep :: 'state-t \Rightarrow 'action-t \Rightarrow 'state-t
and output-f :: 'state-t \Rightarrow 'action-t \Rightarrow 'output-t
and s0 :: 'state-t
and current :: 'state-t =\rightarrow 'dom-t — Returns the currently active domain
and cswitch :: 'time-t \Rightarrow 'state-t \Rightarrow 'state-t — Switches the current domain
and interrupt :: 'time-t \Rightarrow bool — Returns t iff an interrupt occurs in the given state at the given time
and kprecondition :: 'state-t \Rightarrow 'action-t \Rightarrow bool — Returns t if an precondition holds that relates the current action to the state
and realistic-execution :: 'action-t execution \Rightarrow bool — In this locale, this function is completely unconstrained.
and control :: 'state-t \Rightarrow 'dom-t \Rightarrow 'action-t execution \Rightarrow (('action-t option) \times 'action-t execution \times 'state-t)
and kinvolved :: 'action-t \Rightarrow 'dom-t set
and ifp :: 'dom-t \Rightarrow 'dom-t \Rightarrow bool
and vpeq :: 'dom-t \Rightarrow 'state-t \Rightarrow 'state-t \Rightarrow bool
+
fixes AS-set :: ('action-t list) set — Returns a set of valid action sequences, i.e., the attack surface
and invariant :: 'state-t \Rightarrow bool
and AS-precondition :: 'state-t \Rightarrow 'dom-t \Rightarrow 'action-t \Rightarrow bool
and aborting :: 'state-t \Rightarrow 'dom-t \Rightarrow 'action-t \Rightarrow bool
and waiting :: 'state-t \Rightarrow 'dom-t \Rightarrow 'action-t \Rightarrow bool
assumes empty-in-AS-set: [] \in AS-set
and invariant-s0 invariant s0
and invariant-after-cswitch \( \forall \ s \ n. \ invariant \ s \ \rightarrow \ \text{invariant} \ (\text{cswitch} \ n \ s) \)
and preconditions-after-cswitch: \( \forall \ s \ d \ n \ a. \ \text{AS-precondition} \ s \ d \ a \ \rightarrow \ \text{AS-precondition} \ (\text{cswitch} \ n \ s) \ d \ a \)
and AS-pre-first-action: \( \forall \ s \ d \ a \ sseq \cdot \ \text{invariant} \ s \ \land \ \text{aseq} \in \text{AS-set} \land \ \text{aseq} \neq [] \ \rightarrow \ \text{AS-precondition} \ s \ d \ (hd \ aseq) \)
and AS-pre-after-step: \( \forall \ s \ a \ a'. \ (3 \ \text{aseq} \in \text{AS-set} . \ \text{is-sub-seq} \ a \ a' \text{aseq}) \ \land \ \text{invariant} \ s \ \land \ \text{AS-precondition} \ s \ (current \ s) \ a \ \land \ \text{~aborting} \ s \ (current \ s) \ a \ \land \ \text{~waiting} \ s \ (current \ s) \ a \ \land \ \text{AS-precondition} \ (kstep \ s \ a) \ (current \ s) \ a' \land \ \text{AS-pre-dom-independent:} \ \forall \ s \ d \ a \ a'. \ \text{current} \ s \ n \ d \ \land \ \text{AS-precondition} \ s \ d \ a \ \rightarrow \ \text{AS-precondition} \ (kstep \ s \ a') \ d \ a \land \ \text{spec-of-invariant:} \ \forall \ s \ a \cdot \ \text{invariant} \ s \ \rightarrow \ \text{invariant} \ (kstep \ s \ a) \)
and kprecondition-def: kprecondition s a \equiv \text{invariant} \ s \ \land \ \text{AS-precondition} \ s \ (current \ s) \ a
and realistic-execution-def: realistic-execution aseq \equiv \set \ aseq \subseteq \text{AS-set}
and control-spec: \\forall \ s \ d \ aseqs . \ \text{case control} \ s \ d \ aseqs \ of \ (a.aseqs',s') \Rightarrow (\text{thread-empty} \ aseqs \ \land \ (a.aseqs') = (None,[])) \ \lor \ \rightarrow \ \text{Nothing happens}
```
Lemma next-execution-is-realistic-partial
precondition-ind s execs invariably true.

invariant and 2.) more refined preconditions for the current action, we have to know that these two are
∧

aseq

where

realistic-executions-ind

definition

∧

realistic-AS-partial

AS-set.

not necessarily an action sequence from AS_set, but it is
(i.e., they must be from AS_set). The first action sequence, however, is being executed and is therefore
∧

realistic executions. All action sequences in the tail of the executions must be complete action sequences
∧

action. We generalize to do induction. Predicate realistic_executions_ind is the inductive version of
therefore not be proven directly: a realistic execution is not necessarily realistic after performing one
∧

ensures, e.g, that a COPY_CHECK is always preceded by a COPY_INIT.

This means that the execution of each domain contains action sequences that are from AS_set. This
∧

interrupt

where

run-total

0-implies-Suc

not0-implies-Suc

/\divides.alt0

¬

¬

 interrupts

where

run-total

spec-of-waiting

and

next-state-precondition

and

next-action-after-cswitch

and

next-action-after-next-state

and

next-action-after-next-step

and

next-state-invariant

and

spec-of-waiting

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-actions s execs)
using not0-implies-Suc by (metis prod-case3,auto)
termination by lexicographic-order

The major part of the proofs in this locale consist of proving that function run_total is equivalent to
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
therefore not be proven directly: a realistic execution is not necessarily realistic after performing one
action. We generalize to do induction. Predicate realistic_executions_ind is the inductive version of
realistic_executions. All action sequences in the tail of the executions must be complete action sequences
(i.e., they must be from AS_set). The first action sequence, however, is being executed and is therefore
not necessarily an action sequence from AS_set, but it is the last part of some action sequence from
AS_set.

definition realistic-AS-partial :: 'action-t list ⇒ bool
where realistic-AS-partial aseq ∋ ∃ n aseq' . n ≤ length aseq' ∧ aseq' ∈ AS-set ∧ aseq = last n aseq'
definition realistic-executions-ind :: ('dom-t ⇒ 'action-t execution) ⇒ bool
where realistic-executions-ind execs ∋ ∀ d . (case execs d of [] ⇒ True | (aseq#aseq) ⇒ realistic-AS-partial aseq ∧ set aseqs ≤ AS-set)

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

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

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

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

- **Assumptions** realistic-exec: realistic-executions execs and invariant: invariant s
- **Shows** strict-equal (run n (Some s) execs) (run-total n s execs)

**Proof**

```plaintext

have ?execs' ≠ [] unfolding next-exec-def by auto
from realistic([unfolded realistic-executions-ind-def, THEN spec, where x=d] d-is-curr thread-not-empty
obtain n aseq' where witness: n ≤ length aseq' ∧ aseq' ∈ AS-set ∧ hd(execs d) = lastn n aseq'
unfolding realistic-AS-partial-def by (cases execs d,auto)
from d-is-curr this subset as-defs have ?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-exec-def by auto
  from realistic([unfolded realistic-executions-ind-def, THEN spec, where x=d] d-is-curr set-tl-is-subset[where x=execs'])
  have subset: set (tl ?execs') ⊆ AS-set
  unfolding Let-def realistic-AS-partial-def
  by (cases execs d,auto)
  from na-def c-aborting d-is-curr
  have ?execs' ≠ [] unfolding next-exec-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'))
  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.
show case unfolding strict-equal-def by auto
next
case (3 n s execs sa)
  assume interrupt: interrupt (Suc n)
  assume IH: (∧sa. strict-equal (Some (cswitch (Suc n) s)) sa ∧ realistic-executions-ind execs ∧ precondind 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: precondind sa execs
    have inv-nsa: precondind (cswitch (Suc n) sa) execs
    proof-
    {
      fix d
      have fst (control (cswitch (Suc n) sa) d (execs d)) → AS-precond (cswitch (Suc n) sa) d
        using next-action-after-cswitch inv-sa unfolded precondition-ind-def ,THEN conjunct2 ,THEN spec
        where x=d]
        precondition-after-cswitch unfolding Let-def B-def precondition-ind-def
        by(cases fst (control (cswitch (Suc n) sa) d (execs d)).auto)
    }
    thus ?thesis using inv-sa invariant-after-cswitch unfolding precondition-ind-def by auto
  qed
  from equal-s-sa realistic inv-nsa 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: (∧sa. strict-equal (Some s) sa ∧ realistic-executions-ind execs ∧ precondind sa execs ⟷
    strict-equal (run n (Some (cswitch (Suc n) s)) execs) (run-total n sa execs))
  have current-s-sa: strict-equal (Some s) sa ⟷ current s = current sa unfolding strict-equal-def by auto
  {
    assume equal-s-sa: strict-equal (Some s) sa
    assume realistic: realistic-executions-ind execs
    assume inv-sa: precondind 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: ¬precond (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: precondind sa execs
  }
from equal-s-sa have s-sa \( s = sa \) unfolding strict-equal-def by auto
from inv-sa have
text
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 \( \Rightarrow \) 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 \( \colon \) 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: \( \neg \)interrupt (Suc n)
assume thread-not-empty: \( \neg \)thread-empty (execs (current s))
assume prec: precondition (next-state s execs) (next-action s execs)
assume IH: /\sa. (\some (\step (next-state s execs) (next-action s execs))) sa \wedge
realistic-executions-ind (next-exec s execs) \wedge precondition-ind sa (next-execs s execs) \Rightarrow
strict-equal (run n (\some (\step (next-state s execs) (next-action s execs))) (next-exec s execs) (run-total n sa (next-execs s 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 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-exec 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 [] \Rightarrow True | aseq \# aseqs \Rightarrow realistic-AS-partial aseq \wedge set aseqs \subseteq AS-set
proof(cases ?na d,simp,rename-tac aseq aseqs,simp,cases d = current s)
case False
fix aseq aseqs
assume next-exec s execs d = aseq \# aseqs
from False this realistic[unfolded realistic-executions-ind-def,THEN spec,where x=d]
show realistic-AS-partial aseq \wedge set aseqs \subseteq AS-set
unfolding next-exec-def by simp
next
case True
  fix aseq aseqs
  assume na-def: next-exec s execs d = aseq # aseqs
  from next-execution-is-realistic-partial na-def True realistic thread-not-empty
  show realistic-AS-partial aseq ∧ set aseqs ≤ AS-set by blast
  qed

thus thesis unfolding realistic-executions-ind-def by auto
qed
have invariant-na: precondition-ind ?ns ?na
proof-
  from spec-of-invariant inv-sa next-state-invariant s-sa have inv-ns: invariant ?ns
  unfolding precondition-ind-def step-def
  by (cases next-action sa execs auto)
  have ∨ d. fst (control ?ns d (?na d)) → AS-precondition ?ns d
  proof-
  { fix d
    { let ?a′ = fst (control ?ns d (?na d))
      assume snd-action-not-none: ?a′ ≠ None
      have AS-precondition ?ns d (the ?a′)
      proof (cases d = current s)
      case True
        { have ?thesis
          proof (cases ?a)
          case (Some a)

          — Assuming that the current domain executes some action a, and assuming that the action a′ after that is not None (statement snd-action-not-none), we prove that the precondition is inductive, i.e., it will hold for a′. Two cases arise: either action a is delayed (case waiting) or not (case executing).
          show ?thesis
          proof (cases ?na d = execs (current s)) rule:case-split[case-names waiting executing]
            case executing — The kernel is executing two consecutive actions a and a′. We show that [a,a′] is a subsequence in some action in AS-set. The PO’s ensure that the precondition is inductive.
            from executing True Some control-spec[THEN spec,THEN spec,THEN spec,where x2=s and x1=d and x=execs d]
            have a-def: a = hd (execs (current s)) ∧ ?na d = (tl (hd (execs (current s)))) # (tl (execs (current s)))
            unfolding next-exec-def next-exec-def Let-def
            by(auto)
            from a-def True snd-action-not-none control-spec[THEN spec,THEN spec,THEN spec,where x2=?ns and x1=d and x=?na d]
            second-elt-is-hd-tl[where x= hd (execs (current s)) and a=hd(tl(hd (execs (current s)))) and x′=tl (tl(hd (execs (current s))))]
            have na-def: the ?a′ = (hd (execs (current s)))!1
            unfolding next-exec-def
            by(auto)
            from Some realistic[unfolded realistic-executions-ind-def,THEN spec,where x=d] True thread-not-empty
            obtain n aseq′ where witness: n ≤ length aseq′ ∧ aseq′ ∈ AS-set ∧ hd(execs d) = lastn n aseq′
            unfolding realistic-AS-partial-def by (cases execs d,auto)
            from True executing length-lt-2-implies-tl-empty[where x=hd (execs (current s))]
            Some control-spec[THEN spec,THEN spec,THEN spec,where x2=s and x1=d and x=execs d]
            snd-action-not-none control-spec[THEN spec,THEN spec,THEN spec,where x2=?ns and x1=d and x=?na d]
            have in-action-sequence: length (hd (execs (current s))) ≥ 2
            unfolding next-exec-def next-exec-def
by auto
from this witness consecutive-is-sub-seq[ where \( a = a \) and \( b = ?a' \) and \( n = n \) and \( y = aseq' \) and \( x = tl \) (\( \text{hd} \ (\text{execs} \ (\text{current} \ s)) \)) ]
\( a \)-def \( na \)-def \( \text{True} \) in-action-sequence
\( x \)-is-hd-snd-tl[ where \( x = \text{hd} \ (\text{execs} \ (\text{current} \ s)) \) ]
have 1: \( \exists \ aseq' \in \text{AS-set} \ . \ \text{is-sub-seq} \ a \ (\text{the} \ ?a') \ aseq'
by (auto)
from True Some inv-sa[unfolded precondition-ind-def,THEN conjunct2,THEN spec,where \( x = \text{current} \ s \)] s-sa
have 2: \( \text{AS-precondition} \ s \ (\text{current} \ s) \ a \)
unfolding strict-equal-def next-action-def B-def by auto
from executing True Some control-spec[THEN spec,THEN spec,THEN spec,where \( x = \text{execs} \ d \)]
have not-aborting: \( \neg \)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 \( x = \text{execs} \ d \)]
have not-waiting: \( \neg \)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 \( x = \text{execs} \ d \) ]
\( \text{AS-prec-after-step} \ (\text{THEN spec,THEN spec,THEN spec,where} \ x = \text{next-state} \ s \ \text{execs} \ and \ x \ = d \) \ 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 \( x = \text{execs} \ d \) ] True waiting control-spec[THEN spec,THEN spec,THEN spec,where \( x = \text{execs} \ d \) ]
have \( a \)-def: \( ?n \ d = \text{execs} \ (\text{current} \ s) \land \text{next-state} \ s \ \text{execs} \ = s \land \text{waiting} \ s \ d \ (\text{the} \ ?a) \)
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 \( x = \text{execs} \ d \)]
have \( na \)-def: the \( ?a' = \text{hd} \ (\text{execs} \ (\text{current} \ s))) \)
unfolding next-action-def next-execs-def
by (auto)
from spec-of-waiting \( a \)-def True
have no-step: \( \text{step} \ s \ ?a = s \ unfolding \ step-def \ by \ (\text{cases next-action} \ s \ \text{execs,auto}) \)
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: \( \text{AS-precondition} \ s \ (\text{current} \ s) \ (\text{the} \ ?a') \)
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 x2=?ns and x1=d and x=?na d]
   control-spec[THEN spec,THEN spec,THEN spec,where x2=s and x1=d and x=execs d]
   have na-def: the ?a' = hd (tl (execs (current s))) ∧ ?na d = tl (execs (current s))
   unfolding next-action-def next-exec-def
   by(auto)
from True None snd-action-not-none control-spec[THEN spec,THEN spec,THEN spec,where x2=?ns and x1=d and x=?na d]
   this
   have 1: tl (execs (current s)) ≠ [] ∧ hd (tl (execs (current s))) ≠ []
   by auto
from this realistic unfold realistic-executions-ind-def,THEN spec,where x=d] True thread-not-empty
   have hd (tl (execs (current s))) ∈ AS-set
   by (cases execs d,auto)
from True snd-action-not-none this
   inv-ns this na-def 1
   AS-prec-first-action[THEN spec,THEN spec,THEN spec,where x2=?ns and x1= current s and x=?na
   (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 ?na d = execs d
   unfolding next-exec-def by auto
from this False current-next-state next-action-after-step
   have ?a' = fst (control (next-state s execs d) (next-exec s execs d))
   unfolding next-action-def by auto
from inv-sa[unfolded pre-condition-ind-def,THEN spec,where x=d] True thread-not-empty
   have AS-precondition s d (the ?a')
   unfolding pre-condition-ind-def next-action-def B-def by (cases fst (control sa d (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=next-state s execs
   and x2=d and x=the ?a and x1=the ?a']
   show ?thesis
   unfolding step-def
   by (cases next-action s execs,auto)
qed
)

hence fst (control ?ns d (?na d)) → AS-precondition ?ns d unfolding B-def
   by (cases fst (control ?ns d (?na d)),auto)
)
thus ?thesis by auto
qed
from this inv-ns show ?thesis
   unfolding pre-condition-ind-def B-def Let-def
by (auto)  
qed  
from equal-ns-nsa realistic-na invariant-na s-sa IH[ where sa=?ns] 
have equal-ns-nt: strict-equal (run n (Some ?ns) ?na) (run-total n (step (next-state sa execs) (next-action sa execs))) (next-exec sa execs)) by (auto) 
}  
from this current-s-sa thread-not-empty not-interrupt prec show ?case by auto  
qed  

have 1: strict-equal (Some s) s unfolding strict-equal-def by simp  
have 2: realistic-executions-inds execs  
proof--  
{  
  fix d  
  have case execs d of [] ⇒ True | aseq ≠ aseqs ⇒ realistic-AS-partial aseq ∧ set aseqs ≤ AS-set  
  proof(cases execs d,simp)  
  case (Cons aseq aseqs)  
  from Cons realistic-exec[unfolded realistic-executions-def,THEN spec,where x=d]  
  have 0: length aseq ≤ length aseq ∧ aseq ∈ AS-set ∧ aseq = lastn (length aseq) aseq  
  unfolding lastn-def realistic-execution-def by auto  
  hence 1: realistic-AS-partial aseq unfolding realistic-AS-partial-def by auto  
  from Cons realistic-exec[unfolded realistic-executions-def,THEN spec,where x=d]  
  have 2: set aseqs ≤ AS-set  
  unfolding realistic-execution-def by auto  
  from Cons 1 2 show ?thesis by auto  
  qed  
}  
thus ?thesis unfolding realistic-executions-inds-def by auto  
qed  
have 3: precondition-ind s execs  
proof--  
{  
  fix d  
  {  
    assume not-empty: fst (control s d (execs d)) ≠ None  
    from not-empty realistic-exec[unfolded realistic-executions-def,THEN spec,where x=d]  
    have current-aseq-is-realistic: hd (execs d) ∈ AS-set  
    using control-spec[THEN spec,THEN spec,THEN spec,where x=execs d and x1=d and x2=s]  
    unfolding realistic-execution-def by(cases execs d,auto)  
    from not-empty current-aseq-is-realistic invariant AS-prec-first-action[THEN spec,THEN spec,THEN spec,where x2=s and x1=d and x=hd (execs d)]  
    have AS-precondition s d (the (fst (control s d (execs d))))  
    using control-spec[THEN spec,THEN spec,THEN spec,where x=execs d and x1=d and x2=s]  
    by auto  
  }  
  hence fst (control s d (execs d)) → AS-precondition s d  
  unfolding B-def  
  by (cases fst (control s d (execs d)),auto)  
}  
from this invariant show ?thesis unfolding precondition-ind-def by auto  
qed  
from thm-inductive 1 2 3 show ?thesis by auto  
qed  

Theorem unwinding_implies_isecure gives security for all realistic executions. For unrealistic exe-
lemmas, it holds vacuously and therefore does not tell us anything. In order to prove security for this refinement (i.e., for function run_total), we have to prove that purging yields realistic runs.

**lemma realistic-purge:**

- shows \( \forall \, \text{execs d . realistic-executions execs} \rightarrow \text{realistic-executions (purge execs d)} \)
- proof:
  - fix execs d
  - assume realistic-executions execs
  - hence realistic-executions (purge execs d)
    - using some\( [\text{where } P=\text{realistic-execution and } x=\text{execs d}] \)
    - unfolding realistic-executions-def purge-def by(simp)
  - thus ?thesis by auto
- qed

**lemma remove-gateway-comm-subset:**

- shows \( \text{set (remove-gateway-communications d exec)} \subseteq \text{set exec \cup \{}\emptyset\} \)
- by(induct exec,auto)

**lemma realistic-ipurge-l:**

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

**lemma realistic-ipurge-r:**

- shows \( \forall \, \text{execs d . realistic-executions execs} \rightarrow \text{realistic-executions (ipurge-r execs d)} \)
- proof:
  - fix execs d
  - assume 1: realistic-executions execs
    - from empty-in-AS-set remove-gateway-comm-subset\( [\text{where } d=d \text{ and } \text{exec=execs d}] \)
    - have realistic-executions (ipurge-r execs d)
    - using some\( [\text{where } P=\lambda \ x . \text{realistic-execution x and } x=\text{execs d}] \)
    - unfolding realistic-executions-def ipurge-r-def by(auto)
  - thus ?thesis by auto
- qed

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

**definition NI-unrelated-total:**

- where NI-unrelated-total
  - \( \equiv \forall \, \text{execs a n . realistic-executions execs} \rightarrow \) (let \( s-f = \text{run-total n s0 execs in} \)
    - output-f (run-total n s0 (purge execs (current s-f))) a
    - current s-f = current (run-total n s0 (purge execs (current s-f))))

**definition NI-indirect-sources-total:**

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

definition isecure-total:\(\text{bool}\

where isecure-total \equiv \text{NI-unrelated-total} \land \text{NI-indirect-sources-total}

theorem unwinding-implies-isecure-total:
shows isecure-total
proof
  from unwinding-implies-secure have secure-partial: \(\text{NI-unrelated}\) unfolding isecure-def by blast
  from unwinding-implies-secure have isecure-partial: \(\text{NI-indirect-sources}\) unfolding isecure-def by blast

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

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

have \(\text{NI-indirect-sources-total} \land \text{NI-indirect-sources-total}\)
proof
  { 
    fix execs a n
    assume realistic: realistic-executions execs
    from invariant-s0 realistic run-total-equals-run[where n=n and s=s0 and execs=execs] 
    have 1: strict-equal (run n (Some s0) execs) (run-total n s0 execs) by auto
  }
have let s-f = run-total n s0 execs in output-f (run-total n s0 (ipurge-l execs (current s-f))) a = output-f (run-total n s0 (ipurge-r execs (current s-f))) a
proof (cases run n (Some s0) execs)
case None
  thus ?thesis using 1 unfolding NI-unrelated-total-def strict-equal-def by auto
next
case (Some s-f)
  from realistic-ipurge-l invariant-s0 realistic run-total-equals-run[where n=n and s=s0 and execs=ipurge-l execs (current s-f)]
  have 2: strict-equal (run n (Some s0) (ipurge-l execs (current s-f))) (run-total n s0 (ipurge-l execs (current s-f))) by auto
  from realistic-ipurge-r invariant-s0 realistic run-total-equals-run[where n=n and s=s0 and execs=ipurge-r execs (current s-f)]
  have 3: strict-equal (run n (Some s0) (ipurge-r execs (current s-f))) (run-total n s0 (ipurge-r execs (current s-f))) by auto
show ?thesis proof(cases run n (Some s0) (ipurge-l execs (current s-f)))
case None
  from 2 None show ?thesis using 2 unfolding NI-unrelated-total-def strict-equal-def by auto
next
case (Some s-ipurge-l)
  show ?thesis
  proof(cases run n (Some s0) (ipurge-r execs (current s-f)))
case None
  from 3 None show ?thesis using 2 unfolding NI-unrelated-total-def strict-equal-def by auto
next
case (Some s-ipurge-r)
  from run n (Some s0) execs = Some s-f, run n (Some s0) (ipurge-l execs (current s-f)) = Some s-ipurge-l, Some 1 2 3 isecure1-partial[unfolded NI-indirect-sources-def,THEN spec,THEN spec,THEN spec.where x=n and x2=execs]
  show ?thesis
  unfolding strict-equal-def NI-unrelated-def
  by(simp add: Let-def B-def B2-def)
  qed
  qed
  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

defs kstep :: 'state-t ⇒ 'action-t ⇒ 'state-t — Executes one atomic kernel action

and output-t :: 'state-t ⇒ 'action-t ⇒ 'output-t — Returns the observable behavior

and s0 :: 'state-t — The initial state

and current :: 'state-t ⇒ 'dom-t — Returns the currently active domain

and cswitch :: 'time-t ⇒ 'state-t ⇒ 'state-t — Performs a context switch

and interrupt :: 'time-t ⇒ bool — Returns true if an interrupt occurs in the given state at the given time

and kinvolved :: 'action-t ⇒ 'dom-t set — Returns the set of domains that are involved in the given action

and ifp :: 'dom-t ⇒ 'state-t ⇒ bool — The security policy.

and vpeq :: 'dom-t ⇒ 'state-t ⇒ 'state-t ⇒ bool — View partitioning equivalence

and AS-set :: ('action-t list) set — Returns a set of valid action sequences, i.e., the attack surface

and invariant :: 'state-t ⇒ bool — Returns an inductive state-invariant

and AS-precondition :: 'state-t ⇒ 'dom-t ⇒ 'action-t ⇒ bool — Returns the preconditions under which the given action can be executed.

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

and waiting :: 'state-t ⇒ 'dom-t ⇒ 'action-t ⇒ 'state-t ⇒ bool — Returns true iff execution of the given action is delayed.

and set-error-code :: 'state-t ⇒ 'action-t ⇒ 'state-t — Sets an error code when actions are aborted.

assumes vpeq-transitive: ∀ a b c u. (vpeq u a b ∧ vpeq u b c) —> vpeq u a c

and vpeq-symmetric: ∀ a b c u. vpeq u a b —> vpeq u b a

and vpeq-reflexive: ∀ a u. vpeq u a a

and ifp-reflexive: ∀ u : ifp u u

and weakly-step-consistent: ∀ s t u a. vpeq u s t ∧ vpeq (current s) s t ∧ invariant s ∧ AS-precondition s (current s) a ∧ invariant t ∧ AS-precondition t (current t) a ∧ current s = current t —> vpeq u (kstep s a) (kstep t a)

and locally-respects: ∀ a s u. ¬ifp (current s) u ∧ invariant s ∧ AS-precondition s (current s) a —> vpeq u s (kstep s a)

and output-consistent: ∀ a s t. vpeq (current s) s t ∧ current s = current t —> (output-t s a) = (output-t t a)

and step-atomicity: ∀ a s t. current s = current t —> current (cswitch n s) = current (cswitch n t)

and cswitch-consistency: ∀ u s t n. vpeq u s t —> vpeq u (cswitch n s) (cswitch n t)

and empty-in-AS-set: [] ∈ AS-set

and invariant-∅ invariant s0

and invariant-after-cswitch: ∀ s n. invariant s —> invariant (cswitch n s)

and precondition-after-cswitch: ∀ s d n a. AS-precondition s d a —> AS-precondition (cswitch n s) d a

and AS-prec-first-action: ∀ s d a seq. invariant s ∧ seq ∈ AS-set ∧ seq ≠ [] —> AS-precondition s d (hd seq)

and AS-prec-after-step: ∀ s a a'. (∃ seq ∈ AS-set . is-sub-seq a a' seq) ∧ invariant s ∧ AS-precondition s (current s) a ∧ ~waiting s (current s) a —> AS-precondition (kstep s a) (current s) a'

and AS-prec-dom-independent: ∀ s d a a'. current s ≠ d ∧ AS-precondition s d a —> AS-precondition (kstep s a') d a

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

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

and aborting-error-update: ∀ s d a a'. current s ≠ d ∧ aborting s d a —> aborting (set-error-code s a') d a

and aborting-after-step: ∀ s a d . current s ≠ d —> aborting (kstep s a) d = aborting s d

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

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

and waiting-error-update: ∀ s d a a'. current s ≠ d ∧ waiting s d a —> waiting (set-error-code s a') d a

and waiting-consistent: ∀ s t u a. current s ≠ s t ∧ waiting s d t —> waiting (set-error-code s a') d a

and precond-after-set-error-code: ∀ s d a a'. AS-precondition s d a ∧ aborting s (current s) a' —> AS-precondition (set-error-code s a') d a
and invariant-after-set-error-code: \( \forall \ s\ a. \text{invariant} \ s \rightarrow \text{invariant} (\text{set-error-code} s\ a) \)
and involved-ifp: \( \forall \ s\ a. \forall \ d \in (\text{kinvolved} a) . \text{AS-precondition} s (\text{current} s) \ a \rightarrow \text{ifp} d (\text{current} s) \)

\begin{verbatim}
begin
\end{verbatim}

### 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`.

\begin{verbatim}
function CISK-control :: 'state-t \Rightarrow 'dom-t \Rightarrow 'action-t execution \Rightarrow ('action-t option \times 'action-t execution \times '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 ([][a#as]#execs') = (None,as'#execs',s) — The current action sequence has been finished. Skip to the next sequence
      | CISK-control s d ((a#as) #execs') = (if aborting s d a then
                                       (None, execs', set-error-code s a)
                                 else if waiting s d a then
                                       (Some a, (a#as)#execs',s)
                                 else
                                       (Some a, as'#execs',s)) — Executing an action sequence

by pat-completeness auto
termination by lexicographic-order

Function `run` defines the execution semantics. This function is presented in [31] by pseudo code (see Algorithm 1). Before defining the run function, we define accessor functions for the control mechanism.

Functions `next_action`, `next_execs` and `next_state` correspond to “control.a”, “control.x” and “control.s” in [31].

\begin{verbatim}
abbreviation next-action::'state-t \Rightarrow ('dom-t \Rightarrow 'action-t execution) \Rightarrow 'action-t option
where next-action = Kernel.next-action current CISK-control
abbreviation next-exec::'state-t \Rightarrow ('dom-t \Rightarrow 'action-t execution) \Rightarrow ('dom-t \Rightarrow 'action-t execution)
where next-exec = Kernel.next-exec current CISK-control
abbreviation next-state::'state-t \Rightarrow ('dom-t \Rightarrow 'action-t execution) \Rightarrow 'state-t
where next-state = Kernel.next-state current CISK-control

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

abbreviation thread-empty::'action-t execution \Rightarrow bool
where thread-empty exec = [] \lor exec = [[]]

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

\begin{verbatim}
function run :: time-t \Rightarrow 'state-t \Rightarrow ('dom-t \Rightarrow 'action-t execution) \Rightarrow 'state-t
where run 0 s execs = s
    | interrupt (Suc n) \implies run (Suc n) s execs = run n (cswitch (Suc n) s) execs
    | ~interrupt (Suc n) \implies thread-empty(execs (current s)) \implies run (Suc n) s execs = run n s execs
    | interrupt (Suc n) \implies ~thread-empty(execs (current s)) \implies
      run (Suc n) s execs = (let control-a = next-action s execs;
                                control-s = next-state s execs;
                                control-x = next-exec s execs in
                                case control-a of None \Rightarrow run n control-s control-x
                                | (Some a) \Rightarrow run n (kstep control-s a) control-x)

using not0-implies-Suc by (metis prod-cases3.auto)
termination by lexicographic-order
\end{verbatim}
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 kstep

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

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

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

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 difference is the control function. In ISK, this function is a generic function called control, in CISK it is interpreted in function CISK_control. It is proven that function CISK_control satisfies all the proof obligations concerning generic function control. In other words, CISK_control is proven to be an interpretation of control. Therefore, all theorems on run_total apply to the run function of CISK as well.

lemma next-action-consistent:
shows ∀ 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 \( \forall s t \text{ execs} . \ vpeq \ (\text{current s}) \ s t \land (\forall d \in \text{involved} \ (\text{next-action s execs}) \ . \ vpeq \ d \ s t) \land \text{current s} = \text{current t} \rightarrow \text{fst} \ (\text{snd} \ (\text{CISK-control} s \ (\text{current s}) \ (\text{execs} \ (\text{current s})))) = \text{fst} \ (\text{snd} \ (\text{CISK-control} t \ (\text{current s}) \ (\text{execs} \ (\text{current s})))) \)
proof-
{ 
  fix s t \text{ execs} 
  assume vpeq: vpeq \ (\text{current s}) \ s t 
  assume vpeq-involved: \( \forall d \in \text{involved} \ (\text{next-action s execs}) \ . \ vpeq d \ s t \) 
  assume current-s-t: \text{current s} = \text{current t} 
  from aborting-consistent current-s-t vpeq 
  have \( \text{aborting} \ t \ (\text{current s}) = \text{aborting} \ s \ (\text{current s}) \) by auto 
  from \( \text{aborting-consistent} \ \text{current-s-t} \ \text{vpeq} \) \[ \text{THEN spec,THEN spec,THEN spec,THEN spec,where} \]
  unfolding Kernel.next-action-def Kernel.involved-def 
  by(cases \( s, (\text{current s}), \text{execs} \ (\text{current s}) \)) rule: CISK-control.cases,auto split: if-split-asm 
} 
thus \( ?\text{thesis} \) by auto 
qed

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

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

lemma locally-respects-next-state:
shows \( \forall s u \text{ execs}. \ \neg \text{ifp} \ (\text{current s}) \ u \rightarrow \text{vpeq} u s \ (\text{next-state} \ s \ \text{execs}) \)
(proof-
{ 

fix \( s \) \( u \) \( \text{execs} \)
assume \( \neg \text{ifp} \) \((\text{current} s)\) \( u \)
hence \( \text{vpeq} \) \((s)\) \((\text{next-state} s)\) \((\text{execs})\)

unfolding Kernel.next-state-def
using vpeq-reflexive set-error-locally-respects
by \((\text{cases} (s,(\text{current} s)\),\text{execs} \((\text{current} s)\))\) rule: CISK-control.cases,auto

thus ?thesis by auto
qed

**Lemma CISK-control-spec:**

shows \( \forall \) \( s \) \( d \) aseqs.

case CISK-control \( s \) \( d \) aseqs of
\( \_ \) \( a \), \( \_ \) \( a \) \( \_ \) \( \Rightarrow \)

thread-empty aseqs \& (a, aseqs') = (None, []) \lor

aseqs \# [] \& \( \_ \) \( \text{hd} \) aseqs \# [] \& \( \text{waiting} \) s' \( \text{d} \) (the a) \& \( \text{waiting} \) s' \( \text{d} \) (the a) \& (a, aseqs') = (Some (\( \text{hd} \) (\( \text{hd} \) aseqs))), t \( (\text{hd} \) aseqs) \# t \( l \) aseqs) \lor

aseqs \# [] \& \( \_ \) \( \text{hd} \) aseqs \# [] \& \( \text{waiting} \) s' \( \text{d} \) (the a) \& (a, aseqs', s') = (Some (\( \text{hd} \) (\( \text{hd} \) aseqs))), aseqs, s) \lor (a, aseqs') = (None, t \( l \) aseqs)

proof
{
fix \( s \) \( d \) aseqs

have case CISK-control \( s \) \( d \) aseqs of
\( \_ \) \( a \), \( \_ \) \( a \) \( \_ \) \( \Rightarrow \)

thread-empty aseqs \& (a, aseqs') = (None, []) \lor

aseqs \# [] \& \( \_ \) \( \text{hd} \) aseqs \# [] \& \( \text{aborting} \) s' \( \text{d} \) (the a) \& \( \text{waiting} \) s' \( \text{d} \) (the a) \& (a, aseqs') = (Some (\( \text{hd} \) (\( \text{hd} \) aseqs))), t \( (\text{hd} \) aseqs) \# t \( l \) aseqs) \lor

aseqs \# [] \& \( \_ \) \( \text{hd} \) aseqs \# [] \& \( \text{waiting} \) s' \( \text{d} \) (the a) \& (a, aseqs', s') = (Some (\( \text{hd} \) (\( \text{hd} \) aseqs))), aseqs, s) \lor (a, aseqs') = (None, t \( l \) aseqs)

by \((\text{cases} (s,d,aseqs))\) rule: CISK-control.cases,auto

thus ?thesis by auto
qed

**Lemma next-action-after-cswitch:**

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

proof
{
fix \( s \) \( n \) \( d \) aseqs

have \( \text{fst} \) (CISK-control (cswitch \((n \) \( s \)) \( d \) aseqs)) = \( \text{fst} \) (CISK-control \( s \) \( d \) aseqs)

using aborting-switch-independent waiting-switch-independent
by \((\text{cases} (s,d,aseqs))\) rule: CISK-control.cases,auto

thus ?thesis by auto
qed

**Lemma next-action-after-next-state:**

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

proof
{
fix \( s \) \( \text{execs} \) \( d \) aseqs

assume I: current \( s \) \( \neq \) \( d \)

have \( \text{fst} \) (CISK-control (next-state \( s \) \( \text{execs} \) \( d \)) aseqs) = \( \text{None} \) \( \lor \) \( \text{fst} \) (CISK-control (next-state \( s \) \( \text{execs} \) \( d \)) aseqs) = \( \text{fst} \) (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 d a execs
  assume AS-precondition s d a
  hence AS-precondition (next-state s execs) d a
  unfolding Kernel.next-state-def
  using precondition-after-set-error-code
  by (cases (s, (current s), execs (current s)) rule: CISK-control.cases, auto)
}
thus ?thesis by auto
qed

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

lemma next-action-from-exec:
shows ∀ s execs . next-action s execs ↭ (λ a . a ∈ actions-in-execution (execs (current s)))
proof--
{ fix s execs
\{ 
fix a
assume 1: next-action s execs = Some a
from 1 have a ∈ actions-in-execution (execs (current s))
unfolding Kernel.next-action-def actions-in-execution-def
by \{ cases (s,(current s).execs (current s)) rule: CISK-control.cases.auto split: if-split-asn \} 
\} 
\hence next-action s execs = (\land a. a ∈ actions-in-execution (execs (current s)))
\unfolding B-def
by (cases next-action s execs.auto)
\} 
\thus \text{?thesis unfolding B-def by (auto)}
\qed

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

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

show ∀ s t execs.
vpeq (current s) s t∧ (∀ d ∈ involved (next-action s execs) . vpeq d s t) ∧ current s = current t →
fst (snd (CISK-control s (current s) (execs (current s)))) = fst (snd (CISK-control t (current s) (execs (current s))))

using next-execs-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 . invariant s → invariant (cswitch n s)
using invariant-after-cswitch by blast
show ∀ s d n a. AS-precondition s d a → AS-precondition (cswitch n s) d a
using precondition-after-cswitch by blast
show invariant s0
using invariant-s0 by blast
show ∀ s d aseq. invariant s ∧ aseq ∈ AS-set ∧ aseq ≠ [] → AS-precondition s d (hd aseq)
using AS-prec-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-prec-after-step by blast
show ∀ s d a a′. current s + d ∧ AS-precondition s d a → AS-precondition (kstep s a′) d a
using AS-prec-dom-independent by blast
show ∀ s a . invariant s → invariant (kstep s a)
using spec-of-invariant by blast
show ∀ s a . kprecondition s a ∈ kprecondition s a

by auto

show ∀ aseq. realistic-execution aseq ≡ set aseq ⊆ AS-set
unfolding realistic-execution-def by auto

show ∀ s a. ∀ d ∈ involved a. kprecondition s (the a) → ifp d (current s)
using involved-ifp unfolding Kernel.involved-def by (auto split: option.splits)
show ∀ s execs. next-action s execs → (λa. 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-execs-subset by blast

show ∀ s d aseqs.
case CISK-control s d aseqs of
(a, aseqs’, s’) ⇒
thread-empty aseqs ∧ (a, aseqs’) = (None, []) ∀

aseq ∉ [] ∧ hd aseqs ∉ [] ∧ aborting s′ d (the a) ∧ ¬ waiting s′ d (the a) ∧ (a, aseqs’) = (Some (hd (hd aseqs))), tl (hd aseqs) # tl aseqs) ∨
aseq ∉ [] ∧ 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-next-state by auto
show \( \forall s \ a \ d\ a\mbox{seqs} \ \mbox{current} \ s /\!\!/ \slash .left s /\!\!/ \slash .right d /\!\!/ \slash .left d /\!\!/ \slash .right \to \left( \mbox{CISK-control} (\mbox{step} s \ a) \ d \ a\mbox{seqs} \right) = \left( \mbox{CISK-control} s \ d \ a\mbox{seqs} \right) \)
using \( \mbox{next-action-after-step} \) by \( \mbox{auto} \)
show \( \forall s \ d \ a\mbox{execs} \ \mbox{AS-precondition} \ s \ d \ a \to \left( \mbox{AS-precondition} (\mbox{next-state} \ s \ \mbox{execs}) \ d \ a \right) \)
using \( \mbox{next-state-precondition} \) by \( \mbox{auto} \)
show \( \forall s \ d \ a\mbox{execs} \ \mbox{invariant} \ (\ \mbox{current} \ s /\!\!/ \slash .left s /\!\!/ \slash .right) \to \left( \mbox{invariant} (\mbox{next-state} \ s \ \mbox{execs}) \right) \)
using \( \mbox{next-state-invariant} \) by \( \mbox{auto} \)
show \( \forall s \ a \ \mbox{waiting} \ s \ (\ \mbox{current} \ s /\!\!/ \slash .left s /\!\!/ \slash .right) \to \left( \mbox{kstep} s \ a = s \right) \)
using \( \mbox{spec-of-waiting} \) by \( \mbox{blast} \)
\( \qed \)

\[ \text{CISK-control kinvolved ifp vpeq AS-set invariant AS-precondition} \]

\[ \text{aborting waiting - interrupt} \]

\[ \text{have run-equals-run-total:} \]
\[ \bigwedge n s \ \mbox{execs} \ . \ \text{run} n s \ \mbox{execs} \equiv \text{Interruptible-Separation-Kernel.run-total kstep current cswitch interrupt} \]

\[ \text{CISK-control n s execs} \]
\[ \text{proof-} \]
\[ \text{fix n s execs} \]
\[ \text{show run n s execs} \equiv \text{Interruptible-Separation-Kernel.run-total kstep current cswitch interrupt CISK-control n s execs} \]
using \( \text{interpreted int.step-def} \) by \( \text{run-total-induct}, \text{auto split: option.splits} \)
\( \qed \)

from \( \text{interpreted} \)

\[ \text{have \( \sigma \): Interruptible-Separation-Kernel.isecure-total kstep output-f s0 current cswitch interrupt realistic-execution} \]

\[ \text{CISK-control kinvolved ifp} \]
by \( \text{metis int.unwinding-implies-isecure-total} \)

from \( \text{0 run-equals-run-total} \)

\[ \text{have 1: NI-unrelated} \]
by \( \text{(metis realistic-executions-def int.isecure-total-def int.realistic-executions-def int.NI-unrelated-total-def NI-unrelated-def)} \)

from \( \text{0 run-equals-run-total} \)

\[ \text{have 2: NI-indirect-sources} \]
by \( \text{(metis realistic-executions-def int.NI-indirect-sources-total-def int.isecure-total-def int.realistic-executions-def NI-indirect-sources-def)} \)

from \( \text{1 2} \)
\[ \text{show ?thesis unfolding isecure-def by auto} \]
\( \qed \)

end

end

4 Instantiation by a separation kernel with concrete actions

theory \( \text{Step-configuration} \)
imports \( \text{Main} \)
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 it at a first glance, for example the L4 microkernel API only provided IPC as communication primitive [16]. In particular, the concrete actions illustrate how an application of the CISK framework
can be used to separate policy enforcement from other computations unrelated to policy enforcement.

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

Lastly, as a limited manipulation of an access control policy is often needed, we also provide an invariant for having a dynamic access control policy whose maximal closure is bounded by the static per-partition access control policy. That the dynamic access control policy is a subset of a static access control policy is expressed by the invariant sp_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 hierachically 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.

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

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

consts partition :: thread-id-t ⇒ partition-id-t
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-object 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
\[ \begin{aligned}
\forall & \text{sp-spec-subj-subj b a} \\
\forall & (\exists c \ y . \text{sp-spec-subj-subj a c} \\
& \land \text{sp-spec-subj-obj c (PAGE y) WRITE} \\
& \land \text{sp-spec-subj-obj b (PAGE y) READ}) \\
\lor (a = b)
\end{aligned} \]

Show that the policies specified in Section 4.2.1 can be derived from the configuration and their definitions.

**Lemma** correct:

**Shows** policy-axioms sp-spec-subj-obj sp-spec-subj-subj ifp

**Proof** (unfold-locales)

**Show** sp-spec-file-provider:

\[ \forall p1 p2 f m1 m2 . \]
\[ \text{sp-spec-subj-obj p1 (FILEP f \ m1 \land} \]
\[ \text{sp-spec-subj-obj p2 (FILEP f \ m2 \rightarrow sp-spec-subj-subj p1 p2} \]

**Unfolding** sp-spec-subj-subj-def by auto

**Show** sp-spec-no-wronly-pages:

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

**Unfolding** sp-spec-subj-obj-def by auto

**Show** ifp-reflexive:

\[ \forall p . \text{ifp p p} \]

**Unfolding** ifp-def by auto

**Show** ifp-compatible-with-sp-spec:

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

**Unfolding** ifp-def by auto

**Show** ifp-compatible-with-ipc:

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

**Unfolding** ifp-def by auto

**Qed**

**End**

**Type-synonym** sp-subj-subj-t = partition-id-t \Rightarrow partition-id-t \Rightarrow bool

**Type-synonym** sp-subj-obj-t = partition-id-t \Rightarrow obj-id-t \Rightarrow mode-t \Rightarrow 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** \[ \forall p . \text{Policy.ifp p p} \]

**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).
**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).

cons

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 := \ 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 ≡
\[s \land sp-impl-subj-subj = \lambda q q'. (p = q \land p' = q') \lor sp-impl-subj-subj s q q'\]

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)

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 partner s =
\[s \land ev-counter := Suc (ev-counter (thread s partner)) \]

definition atomic-step-ev-wait-one :: thread-id-t ⇒ state-t ⇒ state-t where
atomic-step-ev-wait-one tid s =
\[s \land ev-counter := (ev-counter (thread s tid) - 1) \]

definition atomic-step-ev-wait-all :: thread-id-t ⇒ state-t ⇒ state-t where
atomic-step-ev-wait-all tid s =
\[s \land ev-counter := 0 \]

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 \textit{EV-SIGNAL-PREP} stage if the precondition for the calling thread does not hold.

\textbf{definition} aborting :: state-t \Rightarrow thread-id-t \Rightarrow int-point-t \Rightarrow bool
where aborting s tid a \equiv \text{case a of SK-IPC dir PREP partner page} \Rightarrow
\neg \text{ipc-precondition tid dir partner page s}
| \text{SK-EV-SIGNAL EV-SIGNAL-PREP partner} \Rightarrow
\neg \text{ev-signal-precondition tid partner s}
| \Rightarrow \text{False}

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

\textbf{definition} waiting :: state-t \Rightarrow thread-id-t \Rightarrow int-point-t \Rightarrow bool
where waiting s tid a \equiv
\text{case a of SK-IPC dir WAIT partner page} \Rightarrow
\neg \text{ipc-precondition partner (opposite-ipc-direction dir) tid (SOME page'. True) s}
| \text{SK-EV-WAIT EV-PREP - \Rightarrow False}
| \text{SK-EV-WAIT EV-WAIT - \Rightarrow ev-counter (thread s tid) = 0}
| \text{SK-EV-WAIT EV-FINISH - \Rightarrow False}
| \Rightarrow \text{False}

\textbf{The atomic step function.} In the definition of \textit{atomic-step} the arguments to an interrupt point are not taken from the thread state – the argument given to \textit{atomic-step} could have an arbitrary value. So, seen in isolation, \textit{atomic-step} allows more transitions than actually occur in the separation kernel. However, the CISK framework (1) restricts the atomic step function by the \textit{waiting} and \textit{aborting} functions as well (2) the set of realistic traces as attack sequences \textit{rAS-set} (Section 4.8). An additional condition is that (3) the dynamic policy used in \textit{aborting} is a subset of the static policy. This is ensured by the invariant \textit{sp-subset}.

\textbf{definition} atomic-step :: state-t \Rightarrow int-point-t \Rightarrow state-t where
atomic-step s ipt \equiv
\text{case ipt of}
| \text{SK-IPC dir stage partner page} \Rightarrow
\text{atomic-step-ipc (current s) dir stage partner page s}
| \text{SK-EV-WAIT EV-PREP consume} \Rightarrow s
| \text{SK-EV-WAIT EV-WAIT consume} \Rightarrow s
| \text{SK-EV-WAIT EV-FINISH consume} \Rightarrow
\text{case consume of}
| \text{EV-CONSUME-ONE} \Rightarrow \text{atomic-step-ev-wait-one (current s) s}
| \text{EV-CONSUME-ALL} \Rightarrow \text{atomic-step-ev-wait-all (current s) s}
| \text{SK-EV-SIGNAL EV-SIGNAL-PREP partner} \Rightarrow s
| \text{SK-EV-SIGNAL EV-SIGNAL-FINISH partner} \Rightarrow
\text{atomic-step-ev-signal (current s) partner s}
| \text{NONE} \Rightarrow s
\end

\textbf{4.4 Preconditions and invariants for the atomic step}

\textbf{theory} Step-invariants
\textbf{imports} Step
\textbf{begin}

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

\textbf{definition} sp-subset s \equiv
∀ p1 p2 . sp-impl-subj-subj s p1 p2 → Policy.sp-spec-subj-subj p1 p2) ∧ (∀ p1 p2 m. sp-impl-subj-obj s p1 p2 m → Policy.sp-spec-subj-obj p1 p2 m)

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

definition atomic-step-precondition :: state-t ⇒ thread-id-t ⇒ int-point-t ⇒ bool
where
atomic-step-precondition s tid ipt ≡
case ipt of
  SK-IPC dir WAIT partner page ⇒
    — the thread managed it past PREP stage
  ipc-precondition tid dir partner page s
  | SK-IPC dir (BUF page') partner page ⇒
    — both the calling thread and its communication partner managed it past PREP and WAIT stages
  ipc-precondition tid dir partner page s ∧ ipc-precondition partner (opposite-ipc-direction dir) tid page' s
  | SK-EV-SIGNAL EV-SIGNAL-FINISH partner ⇒
  ev-signal-precondition tid partner s
  | × ⇒
    — No precondition for other interrupt points.
    True

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

definition atomic-step-invariant :: state-t ⇒ bool
where
atomic-step-invariant s ≡
sp-subset s

4.4.1 Atomic steps of SK_IPC preserve invariants

lemma set-object-value-invariant:
  shows atomic-step-invariant s = atomic-step-invariant (set-object-value ob va s)
proof –
  show ?thesis
  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

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 \( ?\text{thesis} \) unfolding atomic-step-def
  using assms atomic-i-pc-preserves-invariants
  by simp

next case (SK-EV-WAIT ev-wait-stage consume)
  then show \( ?\text{thesis} \)
  proof (cases consume)
    case EV-CONSUME-ALL
    then show \( ?\text{thesis} \) 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 \( ?\text{thesis} \) unfolding atomic-step-def
  using SK-EV-WAIT assms atomic-ev-wait-one-preserves-invariants
  by (simp split: ev-wait-stage-t.splits)

next case SK-EV-SIGNAL
  then show \( ?\text{thesis} \) unfolding atomic-step-def
  using assms atomic-ev-signal-preserves-invariants
  by (simp add: ev-signal-stage-t.splits)

next case NONE
  then show \( ?\text{thesis} \) unfolding atomic-step-def
  using assms
  by auto

qed

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

theorem cswitch-preserves-invariants:
  fixes s :: state-t
  and new-current :: thread-id-t
  assumes atomic-step-invariant s
  shows atomic-step-invariant (s \( \{ \text{current} := \text{new-current} \} \))

proof –
  let \( ?s1 = s \{ \text{current} := \text{new-current} \} \)
  have sp-subset s = sp-subset ?s1
    unfolding sp-subset-def by auto
  from assms this show \( ?\text{thesis} \)
  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 \( ?\text{thesis} \)
  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
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 \text{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 \text{open()} and observing success or failure.

\begin{definition}
\text{vpeq-obj} :: \text{partition-id-t} \Rightarrow \text{state-t} \Rightarrow \text{state-t} \Rightarrow \text{bool} \where
\forall \text{obj-id}. \text{Policy.sp-spec-subj-obj u obj-id} \Rightarrow \text{READ} \rightarrow (\text{obj s}) \Rightarrow (\text{obj t}) \Rightarrow \text{obj-id}
\end{definition}

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

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

\begin{definition}
\text{vpeq-local} :: \text{partition-id-t} \Rightarrow \text{state-t} \Rightarrow \text{state-t} \Rightarrow \text{bool} \where
\forall \text{tid}. (\text{partition tid}) \Rightarrow \text{u} \Rightarrow (\text{thread s tid}) \Rightarrow (\text{thread t tid})
\end{definition}

\begin{definition}
\text{vpeq u s t} \equiv \text{vpeq-obj u s t} \land \text{vpeq-subj-subj u s t} \land \text{vpeq-subj-obj u s t} \land \text{vpeq-local u s t}
\end{definition}

4.5.1 Elementary properties

\begin{lemma}
\text{vpeq-rel}:
\begin{itemize}
\item \text{vpeq-refl}: \text{vpeq u s s}
\item \text{vpeq-sym}: \text{vpeq u s t} \Rightarrow \text{vpeq u t s}
\item \text{vpeq-trans}: \text{vpeq u s t} \Rightarrow \text{vpeq u s2 s3} \Rightarrow \text{vpeq u s1 s3}
\end{itemize}
\end{lemma}

\begin{lemma}
\text{set-object-value-ign}:
\begin{itemize}
\item \text{eq-obs} \Rightarrow \text{Policy.sp-spec-subj-obj u x READ}
\item \text{vpeq u s (set-object-value x y s)}
\end{itemize}
\end{lemma}

\begin{proof}
\begin{itemize}
\item \text{vpeq-def} \Rightarrow \text{vpeq-obj-def} \Rightarrow \text{vpeq-subj-subj-def} \Rightarrow \text{vpeq-subj-obj-def} \Rightarrow \text{vpeq-local-def}
\item \text{auto}
\end{itemize}
\end{proof}

Auxiliary equivalence relation.
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** noc ~ 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
unfolding atomic-step-ipc-def set-object-value-def
unfolding vpeq-def vpeq-obj-def vpeq-subj-obj-def vpeq-subj-subj-def vpeq-local-def
by auto
qed
qed

lemma ev-signal-respects-policy:
assumes no ~ 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) \notin u
unfolding Policy.ifp-def atomic-step-precondition-def sp-subset-def
by auto
with prec have 1:(partition partner) \notin u
unfolding atomic-step-precondition-def ev-signal-precondition-def
by (auto simp add: ev-signal-stage-1.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 prec: atomic-step-precondition s tid ipt
and ipt-case: ipt = SK-EV-WAIT ev-wait-stage EV-CONSUME-ALL
shows vpeq u s (atomic-step-ev-wait-all tid s)
proof –
from assms have 1: (partition tid) \notin u
unfolding Policy.ifp-def
by simp
then have 2:vpeq-local u s (atomic-step-ev-wait-all tid s)
unfolding vpeq-local-def atomic-step-ev-wait-all-def
by simp
have 3:vpeq-obj u s (atomic-step-ev-wait-all tid s)
unfolding vpeq-obj-def atomic-step-ev-wait-all-def
by simp
have 4:vpeq-subj-subj u s (atomic-step-ev-wait-all tid s)
unfolding vpeq-subj-subj-def atomic-step-ev-wait-all-def
by simp
have 5:vpeq-subj-obj u s (atomic-step-ev-wait-all tid s)
unfolding vpeq-subj-obj-def atomic-step-ev-wait-all-def
by simp
with 2 3 4 5 show ?thesis
unfolding vpeq-def
by simp
qed

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 noPedido Policy.ifp (partition tid) u
and inv: atomic-step-invariant s
and prec: atomic-step-precondition s tid ipt
and ipt-case: ipt = SK-EV-WAIT ev-wait-stage EV-CONSUME-ONE
shows vpeq u s (atomic-step-ev-wait-one tid s)
proof –
from assms have 1:(partition tid) # u
unfolding Policy.ifp-def
by simp
then have 2:vpeq-local u s (atomic-step-ev-wait-one tid s)
unfolding vpeq-local-def atomic-step-ev-wait-one-def
by simp
have 3:vpeq-obj u s (atomic-step-ev-wait-one tid s)
unfolding vpeq-obj-def atomic-step-ev-wait-one-def
by simp
have 4:vpeq-subj-subj u s (atomic-step-ev-wait-one tid s)
unfolding vpeq-subj-subj-def atomic-step-ev-wait-one-def
by simp
have 5:vpeq-subj-obj u s (atomic-step-ev-wait-one tid s)
unfolding vpeq-subj-obj-def atomic-step-ev-wait-one-def
by simp
with 2 3 4 5 show ?thesis
unfolding vpeq-def
by simp
qed
4.7 Weak step consistency

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

The notion of weak step consistency is common usage. We augment it by assuming that the atomic-step-invariant holds (see [31]).

4.7.1 Weak step consistency of auxiliary functions

lemma ipc-precondition-weakly-step-consistent:
  assumes eq-tid : vpeq (partition tid) s1 s2
  and inv1 : atomic-step-invariant s1
  and inv2 : atomic-step-invariant s2
  shows ipc-precondition tid dir partner page s1 = ipc-precondition tid dir partner page s2
  proof
  let ?sender = case dir of SEND ⇒ tid / divides.alt0
  RECV ⇒ partner
  let ?receiver = case dir of SEND ⇒ partner / divides.alt0
  RECV ⇒ tid
  let ?local-access-mode = case dir of SEND ⇒ READ / divides.alt0
  RECV ⇒ WRITE
  let ?A = sp-impl-subj-subj s1 (partition ?sender) (partition ?receiver)
  = sp-impl-subj-subj s2 (partition ?sender) (partition ?receiver)
  let ?B = sp-impl-subj-obj s1 (partition tid) (PAGE page) ?local-access-mode
  = sp-impl-subj-obj s2 (partition tid) (PAGE page) ?local-access-mode
  have A : ?A
  proof (cases Policy.sp-spec-subj-subj (partition ?sender) (partition ?receiver))
    case True
    thus ?A
    using eq-tid unfolding vpeq-def vpeq-subj-subj-def
    by (simp split: ipc-direction-t.splits)
  next case False
  have sp-subset s1 and sp-subset s2
  using inv1 inv2 unfolding atomic-step-invariant-def sp-subset-def by auto
  hence ¬ sp-impl-subj-subj s1 (partition ?sender) (partition ?receiver)
  and ¬ sp-impl-subj-subj s2 (partition ?sender) (partition ?receiver)
  using False unfolding sp-subset-def by auto
  thus ?A by auto
  qed
  have B : ?B
  proof (cases Policy.sp-spec-subj-obj (partition tid) (PAGE page) ?local-access-mode)
    case True
    thus ?B
    using eq-tid unfolding vpeq-def vpeq-subj-obj-def
    by (simp split: ipc-direction-t.splits)
  next case False
  have sp-subset s1 and sp-subset s2
  using inv1 inv2 unfolding atomic-step-invariant-def sp-subset-def by auto
  hence ¬ sp-impl-subj-obj s1 (partition tid) (PAGE page) ?local-access-mode
  and ¬ sp-impl-subj-obj s2 (partition tid) (PAGE page) ?local-access-mode
  using False unfolding sp-subset-def by auto
  thus ?B by auto

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

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

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 \( (\text{partition tid}) \) \( s1 \) \( s2 \)
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
\( (\text{atomic-step-ipc tid dir stage partner page s1}) \)
\( (\text{atomic-step-ipc tid dir stage partner page s2}) \)
proof –
have \( \forall \text{mypage . } [\text{dir} = \text{SEND}; \text{stage} = \text{BUF mypage}] \implies ?thesis \)
proof –
fix \( \text{mypage} \)
assume dir-send: \( \text{dir} = \text{SEND} \)
assume stage-buf: \( \text{stage} = \text{BUF mypage} \)
have Policy,\( sp\)-spec-subj-obj \( (\text{partition tid}) \) \( (\text{PAGE page}) \) \( \text{READ} \)
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 \( s1 \) \( (\text{PAGE page}) = \) obj \( s2 \) \( (\text{PAGE page}) \)
using eq-act unfolding vpeq-def vpeq-obj-def vpeq-local-def
by auto
thus vpeq u
\( (\text{atomic-step-ipc tid dir stage partner page s1}) \)
\( (\text{atomic-step-ipc tid dir stage partner page s2}) \)
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 \( \text{vpeq } u \left( \text{extend-subj-subj } f s1 \right) \left( \text{extend-subj-subj } f s2 \right) \)

proof −
let \(?g1 = \text{sp-impl-subj-subj } s1 \) and \(?g2 = \text{sp-impl-subj-subj } s2 \)
have \( \forall v . \text{Policy.sp-spec-subj-subj } u v \rightarrow \ ?g1 u v = \ ?g2 u v \)
and \( \forall v . \text{Policy.sp-spec-subj-subj } v u \rightarrow \ ?g1 v u = \ ?g2 v u \)
using assms unfolding vpeq-def vpeq-subj-subj-def by auto

hence \( \forall v . \text{Policy.sp-spec-subj-subj } u v \rightarrow \ ?g1 u v = \ ?g2 u v \)
and \( \forall v . \text{Policy.sp-spec-subj-subj } v u \rightarrow \ ?g1 v u = \ ?g2 v u \)

unfolding extend-f-def by auto
hence \( 1: \text{vpeq-subj-subj } u \left( \text{extend-subj-subj } f s1 \right) \left( \text{extend-subj-subj } f s2 \right) \)
unfolding vpeq-subj-subj-def extend-subj-subj-def by auto

hence \( 2: \text{vpeq-obj } u \left( \text{extend-subj-subj } f s1 \right) \left( \text{extend-subj-subj } f s2 \right) \)
using assms unfolding vpeq-def vpeq-subj-subj-def by fastforce

hence \( 3: \text{vpeq-local } u \left( \text{extend-subj-subj } f s1 \right) \left( \text{extend-subj-subj } f s2 \right) \)
using assms unfolding vpeq-def vpeq-local-def extend-subj-subj-def by fastforce

from \( 1 2 3 4 \) show \( \text{?thesis} \)
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 \( \text{Step.partition} \ tid \).

theorem atomic-step-weakly-step-consistent:
assumes eq-obs: \( \text{vpeq } u s1 s2 \)
and eq-act: \( \text{vpeq } \left( \text{partition } \left( \text{current } s1 \right) \right) s1 s2 \)
and inv1: \( \text{atomic-step-invariant } s1 \)
and inv2: \( \text{atomic-step-invariant } s2 \)
and prec1: \( \text{atomic-step-precondition } s1 \left( \text{current } s1 \right) \left( \text{ipt} \right) \)
and prec2: \( \text{atomic-step-precondition } s2 \left( \text{current } s2 \right) \left( \text{ipt} \right) \)
and eq-curr: \( \text{current } s1 = \text{current } s2 \)
shows \( \text{vpeq } u \left( \text{atomic-step } s1 \left( \text{ipt} \right) \right) \left( \text{atomic-step } s2 \left( \text{ipt} \right) \right) \)

proof −
show \( \text{?thesis} \)
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
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.

const initial-current :: thread-id-t
const initial-obj :: obj-id-t \rightarrow obj-t

definition s0 \in state-t where
  s0 \equiv \{ 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 = \lambda . \{ 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 rabs \in state-t => rstate-t (\dagger -) where rabs = Abs-rstate-t
definition rrep \in rstate-t => state-t (\dagger -) where rrep = Rep-rstate-t

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

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

A CISK action is identified with an interrupt point.

type-synonym \( raction-t = \text{int-point-t} \)

definition \( rcurrent :: rstate-t \Rightarrow \text{thread-id-t} \) where
\( rcurrent s = \text{current } \downarrow s \)

definition \( rstep :: rstate-t \Rightarrow raction-t \Rightarrow rstate-t \) where
\( rstep s a \equiv \uparrow (\text{atomic-step } (\downarrow s) a) \)

Each CISK domain is identified with a thread id.

type-synonym \( rdom-t = \text{thread-id-t} \)

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

datatype \( \text{visible-obj-t} = \text{VALUE obj-t} | \text{EXCEPTION} \)

type-synonym \( \text{routput-t} = \text{page-t} \Rightarrow \text{visible-obj-t} \)

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

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

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

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

Translate view-partitioning and interaction-allowed relations.

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

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

Context Switches

definition \( \text{rcswitch} :: \text{nat} \Rightarrow rstate-t \Rightarrow rstate-t \) where
\( \text{rcswitch } n s \equiv \uparrow (\text{if current := } (\text{SOME } t . \text{True}) [] \) \)

4.8.3 Possible action sequences

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

definition \( \text{is-SK-IPC} :: raction-t \ list \Rightarrow \text{bool} \) where
\( \text{is-SK-IPC } aseq \equiv \exists \ dir \ partner \ page . \)
\( \text{aseq } = [\text{SK-IPC } dir \text{ PREP partner page}, \text{SK-IPC } dir \text{ WAIT partner page}, \text{SK-IPC } dir \ (\text{BUF } (\text{SOME page’} . \text{True})) \text{ partner page]} \)
An \textit{SK-EV-WAIT} consists of three atomic actions, one for each of the stages \textit{EV-PREP}, \textit{EV-WAIT} and \textit{EV-FINISH} with the same parameters.

\textbf{definition} \textit{is-SK-EV-WAIT} :: raction-t list \Rightarrow \textit{bool}
\textbf{where} \textit{is-SK-EV-WAIT} aseq \equiv \exists \text{ consume} .
\begin{align*}
  \text{aseq} &= [\text{SK-EV-WAIT EV-PREP consume}, \\
  &\hspace{1cm} \text{SK-EV-WAIT EV-WAIT consume}, \\
  &\hspace{1cm} \text{SK-EV-WAIT EV-FINISH consume}]
\end{align*}

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

\textbf{definition} \textit{is-SK-EV-SIGNAL} :: raction-t list \Rightarrow \textit{bool}
\textbf{where} \textit{is-SK-EV-SIGNAL} aseq \equiv \exists \text{ partner} .
\begin{align*}
  \text{aseq} &= [\text{SK-EV-SIGNAL EV-SIGNAL-PREP partner}, \\
  &\hspace{1cm} \text{SK-EV-SIGNAL EV-SIGNAL-FINISH partner}]
\end{align*}

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

\textbf{definition} \textit{rAS-set} :: raction-t list set
\textbf{where} \textit{rAS-set} \equiv \{ \text{aseq} . \text{is-SK-IPC aseq} \lor \text{is-SK-EV-WAIT aseq} \lor \text{is-SK-EV-SIGNAL aseq} \} \cup \{[]\}

4.8.4 Control

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

\textbf{abbreviation} \textit{raborting}
\textbf{where} \textit{raborting} \text{ s} \equiv \text{aborting} (\downarrow \text{ s})

\textbf{abbreviation} \textit{rwaiting}
\textbf{where} \textit{rwaiting} \text{ s} \equiv \text{waiting} (\downarrow \text{ s})

\textbf{definition} \textit{rset-error-code} :: rstate-t \Rightarrow raction-t \Rightarrow rstate-t
\textbf{where} \textit{rset-error-code} \text{ s} a \equiv \text{ s}

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

\textbf{definition} \textit{rkinvolved} :: int-point-t \Rightarrow rdom-t set
\textbf{where} \textit{rkinvolved} \text{ a} \equiv \text{ case a of SK-IPC dir WAIT partner page } \Rightarrow \{ \text{partner}\} \\
| SK-EV-SIGNAL EV-SIGNAL-FINISH partner } \Rightarrow \{ \text{partner}\} \\
| \_ \Rightarrow \{\}

\textbf{abbreviation} \textit{rinvolved}
\textbf{where} \textit{rinvolved} \equiv \text{ Kernel.involved rkinvolved}

4.8.5 Discharging the proof obligations

\textbf{lemma} \textit{inst-vpeq-rel}:
\textbf{shows} rvpeq-refl: rvpeq u s s \\
and rvpeq-sym: rvpeq u s1 s2 \implies rvpeq u s2 s1 \\
and rvpeq-trans: [ [ rvpeq u s1 s2; rvpeq u s2 s3 ] ] \implies rvpeq u s1 s3
\textbf{unfolding} rvpeq-def using vpeq-rel by metis+

\textbf{lemma} \textit{inst-ifp-refl}:
\textbf{shows} \forall u . rifp u u
\textbf{unfolding} rifp-def using Policy-properties.ifp-reflexive by fast

\textbf{lemma} \textit{inst-step-atomicity} [simp]:
\textbf{shows} \forall s a . rcurrent (rstep s a) = rcurrent s
\textbf{unfolding} rstep-def rcurrent-def
by auto

lemma inst-weakly-step-consistent:
assumes rvpeq u s t
and rvpeq (rcurrent s) s t
and rcurrent s = rcurrent t
and rprecondition s (rcurrent s) a
and rprecondition t (rcurrent t) a
shows rvpeq u (rstep s a) (rstep t a)
using assms atomic-step-weakly-step-consistent rstate-invariant atomic-step-preserves-invariants
unfolding rcurrent-def rstep-def rvpeq-def rprecondition-def
by auto

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

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

thus \( \forall \ a \ s \ t. \ rvpeq \ ( rcurrent \ s) \ s \ t \land rcurrent \ s = rcurrent \ t \mapsto \ routput-f \ s \ a = routput-f \ t \ a \)
by auto
qed
thus \(?\)thesis using assms by auto
qed

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

lemma \texttt{inst-cswitch-consistency}:
assumes \( rvpeq \ u \ s \ t \)
shows \( rvpeq \ u \ ( rcswitch \ n \ s) \ ( rcswitch \ n \ t) \)
proof-
have 1: \( vpeq \ ( \text{partition} \ u) \ ( \downarrow \ s \downarrow \ ( rcswitch \ n \ s) \)
using \( rstate-invariant \ cswitch-consistency-and-respect cswitch-preserves-invariants \) unfolding \( rcswitch-def \)
by auto
have 2: \( vpeq \ ( \text{partition} \ u) \ ( \downarrow \ t \downarrow \ ( rcswitch \ n \ t) \)
using \( rstate-invariant \ cswitch-consistency-and-respect cswitch-preserves-invariants \) unfolding \( rcswitch-def \)
by auto
from 1 2 assms show ?thesis unfolding \( rvpeq-def \) using \( vpeq-rel \) by metis
qed

For the \texttt{PREP} stage (the first stage of the IPC action sequence) the precondition is True.

lemma \texttt{prec-first-IPC-action}:
assumes \( is-SK-IPC \ aseq \)
shows \( rprecondition \ s \ d \ ( \text{hd} \ aseq) \)
using assms unfolding \( is-SK-IPC-def rprecondition-def atomic-step-precondition-def \)
by auto

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

lemma \texttt{prec-first-EV-WAIT-action}:
assumes \( is-SK-EV-WAIT \ aseq \)
shows \( rprecondition \ s \ d \ ( \text{hd} \ aseq) \)
using assms unfolding \( is-SK-EV-WAIT-def rprecondition-def atomic-step-precondition-def \)
by auto

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

lemma \texttt{prec-first-EV-SIGNAL-action}:
assumes \( is-SK-EV-SIGNAL \ aseq \)
shows \( rprecondition \ s \ d \ ( \text{hd} \ aseq) \)
using assms unfolding \( is-SK-EV-SIGNAL-def rprecondition-def atomic-step-precondition-def \)
ev-signal-precondition-def
by auto
When not waiting or aborting, the precondition is “1-step inductive”, that is at all times the precondition holds initially (for the first step of an action sequence) and after doing one step.

**Lemma** prec-after-IPC-step:
**Assumes**
- \( \text{prec: } r\text{precondition } s \ (r\text{current } s) \ (\text{aseq} \ ! n) \)
- \( \text{and } n\text{-bound: } \text{Suc } n < \text{length } a\text{seq} \)
- \( \text{and } \text{IPC: } \text{is-SK-IPC } a\text{seq} \)
- \( \text{and not-aborting: } \neg r\text{aborting } s \ (r\text{current } s) \ (\text{aseq} \ ! n) \)
- \( \text{and not-waiting: } \neg r\text{waiting } s \ (r\text{current } s) \ (\text{aseq} \ ! n) \)

**Shows**
- \( r\text{precondition } (r\text{step } s \ (\text{aseq} \ ! n)) \ (r\text{current } s) \ (\text{aseq} \ ! \text{Suc } n) \)

**Proof**
- Fix dir partner page
  - Let \( ?\text{page}' = (\text{SOME } \text{page}'. \text{True}) \)
  - Assume IPC: \( a\text{seq} = [\text{SK-IPC dir PREP partner page}, \text{SK-IPC dir WAIT partner page}, \text{SK-IPC dir (BUF } ?\text{page}') \text{ partner page}] \)
  - Assume 0: \( n=0 \)
    - From 0 IPC \( \text{prec not-aborting} \)
      - Have \( \text{thesis} \)
        - By (auto)
  - Moreover
    - Assume 1: \( n=1 \)
    - From 1 IPC \( \text{prec not-waiting} \)
      - Have \( \text{thesis} \)
        - By (auto)
  - Moreover
    - From IPC
      - Have length aseq = 3
        - By auto
    - Ultimately
      - Have \( \text{thesis} \)
      - Using n-bound
        - By arith
  - Thus \( \text{thesis} \)
    - 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**
- \( \text{prec: } r\text{precondition } s \ (r\text{current } s) \ (\text{aseq} \ ! n) \)
- \( \text{and } n\text{-bound: } \text{Suc } n < \text{length } a\text{seq} \)
- \( \text{and IPC: } \text{is-SK-EV-WAIT } a\text{seq} \)
- \( \text{and not-aborting: } \neg r\text{aborting } s \ (r\text{current } s) \ (\text{aseq} \ ! n) \)
- \( \text{and not-waiting: } \neg r\text{waiting } s \ (r\text{current } s) \ (\text{aseq} \ ! n) \)

**Shows**
- \( r\text{precondition } (r\text{step } s \ (\text{aseq} \ ! n)) \ (r\text{current } s) \ (\text{aseq} \ ! \text{Suc } n) \)

**Proof**
- {
fix consume

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

{  
  assume 0: n=0
  from 0 WAIT prec not-aborting
  have ?thesis
  unfolding rprecondition-def atomic-step-precondition-def
  by(auto)
}

moreover
{  
  assume 1: n=1
  from 1 WAIT prec not-waiting
  have ?thesis
  unfolding rprecondition-def atomic-step-precondition-def
  by(auto)
}

moreover
from WAIT
have length aseq = 3
by auto

ultimately
have ?thesis
using n-bound
by arith

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

qed

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

lemma prec-after-EV-SIGNAL-step:
assumes prec: rprecondition s (rcurrent s) (aseq ! n)
and n-bound: Suc n < length aseq
and SIGNAL: is-SK-EV-SIGNAL aseq
and not-aborting: ¬raborting s (rcurrent s) (aseq ! n)
and not-waiting: ¬rrwaiting s (rcurrent s) (aseq ! n)
shows rprecondition (rstep s (aseq ! n)) (rcurrent s) (aseq ! Suc n)

proof-
{  
  fix partner
  assume SIGNAL1: aseq = [SK-EV-SIGNAL EV-SIGNAL-PREP partner,
                           SK-EV-SIGNAL EV-SIGNAL-FINISH partner]

  {  
    assume 0: n=0
    from 0 SIGNAL1 prec not-aborting
    have ?thesis
    unfolding rprecondition-def atomic-step-precondition-def ev-signal-precondition-def
    abortion-def rstep-def atomic-step-def
    by auto
  }

  moreover
  from SIGNAL1
  have length aseq = 2

  more...
by auto
ultimately
have thesis
using n-bound
by arith
}
thus thesis
using assms
unfolding is-SK-EV-SIGNAL-def
by auto
qed

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

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

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

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

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

lemma prec-dom-independent:
shows ∀ s d a a' . rcurrent s /slash.left d ∧ rprecondition s d a → rprecondition (rstep s a') d a
using atomic-step-preserves-invariants
rstate-invariant prec-IPC-dom-independent prec-ev-signal-dom-independent
prec-ev-wait-all-dom-independent prec-ev-wait-one-dom-independent
unfolding rcurrent-def rprecondition-def rstep-def atomic-step-def
by (auto split: int-point.t.splits ev-consume.t.splits ev-wait-stage.t.splits ev-signal-stage.t.splits)

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

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

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

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

\}

thus \(?thesis\) by auto

qed

lemma aborting-after-IPC-step:
assumes \( d1 \not\equiv d2 \)
shows aborting (atomic-step-ipc \( d1 \) dir stage partner page \( s \)) \( d2 \) \( a \) = aborting \( s \) \( d2 \) \( a \)
unfolding atomic-step-ipc-def aborting-def set-object-value-def ipc-precondition-def ev-signal-precondition-def
by(auto split: int-point-t.splits ipc-stage-t.splits ipc-direction-t.splits ev-signal-stage-t.splits)

lemma waiting-after-IPC-step:
assumes \( d1 \not\equiv d2 \)
shows waiting (atomic-step-ipc \( d1 \) dir stage partner page \( s \)) \( d2 \) \( a \) = waiting \( s \) \( d2 \) \( a \)
unfolding atomic-step-ipc-def waiting-def set-object-value-def ipc-precondition-def
by(auto split: int-point-t.splits ipc-stage-t.splits ipc-direction-t.splits ev-wait-stage-t.splits)

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

lemma aborting-dom-independent:
assumes \( r\text{current} \ s \not\equiv d \)
shows raborting (rstep \( s \) \( a \)) \( d \ a' \) = raborting \( s \) \( d \ a' \)
proof –
\have \( \\land \tid \ dir \ partner \ page \ s \cdot \text{ipc-precondition} \ tid \ dir \ partner \ page \ s = \text{ipc-precondition} \ tid \ dir \ partner \ page \ (\text{atomic-step} \ s \ a) \)
∧ ev\text{-signal-precondition} tid partner \( s \) = ev\text{-signal-precondition} tid partner \( (\text{atomic-step} \ s \ a) \)
proof –
\fix \ tid \ dir \ partner \ page \ s
\let \( ?s = \text{atomic-step} \ s \ a \)
have \( (\forall p q . \text{sp-impl-subj subj } s \ p \ q = \text{sp-impl-subj subj } ?s \ p \ q) \)
\& \( (\forall p x m . \text{sp-impl-subj obj } s \ p \ x \ m = \text{sp-impl-subj obj } ?s \ p \ x \ m) \)

unfolding atomic-step-def atomic-step-ipc-def
atomic-step-ev-signal-def set-object-value-def
by (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 ipc-precondition tid dir partner page s = ipc-precondition tid dir partner page (atomic-step s a)
\& ev-signal-precondition tid partner s = ev-signal-precondition tid partner (atomic-step s a)

unfolding ipc-precondition-def ev-signal-precondition-def by simp
qed

moreover have \( (\downarrow (\uparrow (\text{atomic-step } (\downarrow s)) b)) = \text{atomic-step } (\downarrow s) b \)
using rstate-invariant atomic-step-preserves-invariants rstate-up-down by auto

ultimately show \(?thesis

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

d31.1 – Formal Specification of a Generic Separation Kernel

Lemma ipc-precondition-of-partner-consistent:

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

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 rypeq-def rkinvolved-def
by auto

Lemma ev-signal-precondition-of-partner-consistent:

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

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 rypeq-def rkinvolved-def
by auto

Lemma waiting-consistent:

shows \( \forall s t u a . \text{rypeq (rcurrent s) s t} \wedge (\forall d \in rkinvolved a . \text{rypeq } d \ s \ t) \)
\wedge rypeq u s t

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

proof –
{ fix s t u a
assume rypeq (rcurrent s) s t
assume vpeq-involved: \( \forall d \in rkinvolved a . \text{rypeq } d \ s \ t \)
assume vpeq-u rypeq 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 rtype-def by (auto split: ipc-stage-t.splits)
next case SK-EV-WAIT
thus rwaiting s u a = rwaiting t u a

EURO-MILS D31.1 Page 86 of 94
using ev-signal-precondition-of-partner-consistent
vpeq-involved vpeq vpeq-u
unfolding waiting-def rkinvolved-def ev-signal-precondition-def
rveq-def vpeq-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 –
let ?sp = \lambdat1 t2 . Policy.sp-spec-subj-subj (partition t1) (partition t2)
have ?sp (current s) partner \lor ?sp partner (current s)
using assms unfolding ipc-precondition-def atomic-step-invariant-def sp-subset-def
by (cases dir, auto)
thus thesis
unfolding rifp-def using Policy-properties.ifp-compatible-with-sp-spec by auto
qed

lemma ev-signal-precondition-ensures-ifp:
assumes ev-signal-precondition (current s) partner s
and atomic-step-invariant s
shows rifp partner (current s)
proof –
let ?sp = \lambdat1 t2 . Policy.sp-spec-subj-subj (partition t1) (partition t2)
have ?sp (current s) partner \lor ?sp partner (current s)
using assms unfolding ev-signal-precondition-def atomic-step-invariant-def sp-subset-def
by (auto)
thus thesis
unfolding rifp-def using Policy-properties.ifp-compatible-with-sp-spec by auto
qed

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

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

lemma spec-of-waiting:
shows \( \forall s \ a. \ r\text{waiting } s (r\text{current } s) a \rightarrow r\text{step } s a = s \)
unfolding waiting-def rstep-def atomic-step-def atomic-step-IPC-def
atomic-step-ev-wait-all-def
atomic-step-ev-wait-one-def
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
\( (\uparrow \theta) \)
rcurrent
rcswitch
rKinvolved
rifp
rvpeq
rAS-set
rinvariant
rprecondition
raborting
rwating
rset-error-code

proof (unfold-locales)
— show that rvpeq is equivalence relation
show \( \forall a \ b \ c \ u. \ (rvpeq u a b \land rvpeq u b c) \rightarrow rvpeq u a c \)
and \( \forall a \ b \ u. \ rvpeq u a b \rightarrow rvpeq u b a \)
and \( \forall a \ u. \ rvpeq u a a \)
using inst-vpeq-rel by metis+
— show output consistency
show \( \forall a \ s \ t. \ rvpeq (rcurrent \ s) s t \land rcurrent \ s = rcurrent \ t \rightarrow routput-f \ s a = routput-f \ t a \)
using inst-output-consistency by metis
— show reflexivity of ifp
show \( \forall u . \ rifp u u \)
using inst-ifp-refl by metis
— show step consistency
show \( \forall s \ t u a. \ rvpeq u s t \land rvpeq (rcurrent \ s) s t \land rprecondition \ s (rcurrent \ s) a \land rprecondition \ t (rcurrent \ t) a \land rcurrent \ s = rcurrent \ t \rightarrow rvpeq u (rstep \ s a) (rstep \ t a) \)
using inst-weakly-step-consistent by blast
— show step atomicity
show \( \forall s a \cdot \text{rcurrent (rstep s a) = rcurrent s} \)
using inst-step-atomicity by metis

show \( \forall a s u. \sim \text{rfip (rcurrent s)} u \land \text{True} \land \text{precondition s (rcurrent s) a } \rightarrow \text{rvpeq u s (rstep s a)} \)
using inst-local-respect by blast
— show cswitch is independent of state

show \( \forall a s u. \neg \text{rifp (rcurrent s)} u \land \text{True} \land \text{precondition s (rcurrent s) a } \rightarrow \text{rvpeq u s (rcurrent s)} \)
using inst-cswitch-independent-of-state by metis

— Show the empt action sequence is in AS-set

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

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

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

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

show \( \forall s d aseq. \text{True } \land \text{aseq } \in \text{rAS-set } \land \text{aseq } \notin \[\] \rightarrow \text{precondition s d (hd aseq)} \)
using prec-first-IPC-action prec-first-EV-WAIT-action prec-first-EV-SIGNAL-action
unfolding rAS-set-def is-sub-seq-def by auto
— Steps of other domains do not influence the precondition

show \( \forall s d a a'. \text{precondition s d a } \rightarrow \text{precondition (rstep s a) d a'} \)
unfolding rAS-set-def is-sub-seq-def by auto
— The invariant

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

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

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

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

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

show \( \forall s t u a. \text{rvpeq (rcurrent s) s t } \land (\forall d \in \text{rkinvolved a} \cdot \text{rvpeq d s t}) \)
\land \text{rvpeq u s t} \)
D31.1 – Formal Specification of a Generic Separation Kernel

→ rwaiting s u a = rwaiting 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 → rifp d (rcurrent s)

using involved-ifp by blast

— An action that is waiting does not change the state

show ∀ s a. rwaiting 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’. rcurrent s ̸= d ∧ raborting s d a → raborting (rset-error-code s a’) d a

unfolding rset-error-code-def

by auto

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

unfolding rset-error-code-def

by auto

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

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’. 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’. 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-secure-CISK

by blast

end
5 Related Work

We consider various definitions of intransitive (I) noninterference (NI). This overview is by no means intended to be complete. We first prune the field by focusing on INI with as granularity the domains: if the security policy states the act “v → u”, this means domain v is permitted to flow any information it has at its disposal to u. We do not consider language-based approaches to noninterference [26], which allow finer granularity mechanisms (i.e., flowing just a subset of the available information). Secondly, several formal verification efforts have been conducted concerning properties similar and related to INI such as no-exfiltration and no-infiltration [9]. Heitmeyer et al. prove these properties for a separation kernel in a Common Criteria certification process [11] (which kernel and which EAL is not clear). Martin et al. proved separation properties over the MASK kernel [18] and Shapiro and Weber verified correctness of the EROS confinement mechanism [28]. Klein provides an excellent overview of 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 [2]. 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-security. Their critique on IP-secure, however, is not universally accepted [7]. Greve at al. provided the GWV framework developed in ACL2 [9]. Their definition is a non-inductive version of noninterference similar to Rushby’s step consistency. GWV has been used on various industrial systems. The exact relation between GWV and IP-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 := 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].

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-security [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 [3].

6 Conclusion

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

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

6.0.1 Acknowledgement.

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

FP7/2007 – 2013

under grant agreement number ICT-318353.

References


