**D31.1**

**Formal Specification of a Generic Separation Kernel**

<table>
<thead>
<tr>
<th>Project number:</th>
<th>318353</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project acronym:</td>
<td>EURO-MILS</td>
</tr>
<tr>
<td>Project title:</td>
<td>EURO-MILS: Secure European Virtualisation for Trustworthy Applications in Critical Domains</td>
</tr>
<tr>
<td>Start date of the project:</td>
<td>1st October, 2012</td>
</tr>
<tr>
<td>Duration:</td>
<td>36 months</td>
</tr>
<tr>
<td>Programme:</td>
<td>FP7/2007-2013</td>
</tr>
<tr>
<td>Deliverable type:</td>
<td>R</td>
</tr>
<tr>
<td>Deliverable reference number:</td>
<td>ICT-318353 / D31.1 / 0.0</td>
</tr>
<tr>
<td>Activity and Work package contributing to deliverable:</td>
<td>Activity 3 / WP 3.1</td>
</tr>
<tr>
<td>Due date:</td>
<td>September 2013 – M12</td>
</tr>
<tr>
<td>Actual submission date:</td>
<td>13th September, 2023</td>
</tr>
<tr>
<td>Responsible organisation:</td>
<td>Open University of The Netherlands</td>
</tr>
<tr>
<td>Editors:</td>
<td>Freek Verbeek, Julien Schmaltz</td>
</tr>
<tr>
<td>Dissemination level:</td>
<td>PU</td>
</tr>
<tr>
<td>Revision:</td>
<td>0.0 (r-2)</td>
</tr>
</tbody>
</table>

**Abstract:**

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

**Keywords:**

separation kernel with control, formal model, instantiation, IPC, events, Isabelle/HOL
Editors
Freek Verbeek, Julien Schmaltz (Open University of The Netherlands)

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

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

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

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

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

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

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

4 **Instantiation by a separation kernel with concrete actions** .................................................. 57
   4.1 Model of a separation kernel configuration .................... 58
      4.1.1 Type definitions ................................................ 58
      4.1.2 Configuration ................................................... 58
   4.2 Formulation of a subject-subject communication policy and an information flow policy, and showing both can be derived from subject-object configuration data .................. 59
      4.2.1 Specification ..................................................... 59
      4.2.2 Derivation ........................................................ 59
   4.3 Separation kernel state and atomic step function ............. 60
      4.3.1 Interrupt points ................................................ 60
      4.3.2 System state ..................................................... 61
      4.3.3 Atomic step ....................................................... 61
   4.4 Preconditions and invariants for the atomic step ............ 63
      4.4.1 Atomic steps of SK_IPC preserve invariants ............... 64
      4.4.2 Summary theorems on atomic step invariants .............. 65
   4.5 The view-partitioning equivalence relation .................... 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

theory Option-Binders
imports Main
begin

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

**definition** B :: 'a option ⇒ ('a ⇒ bool) ⇒ bool (infixl → 65)

**where** B m α ≡ case m of None ⇒ True | (Some a) ⇒ α a

**definition** B2 :: 'a option ⇒ 'a option ⇒ ('a ⇒ 'a ⇒ bool) ⇒ bool

**where** B2 m1 m2 α ≡ m1 → (λ a . m2 → (λ b . α a b))

**syntax** B2 :: ['a option, 'a option, ('a ⇒ 'a ⇒ bool)] ⇒ bool ((⊥ || → ⊥) [0, 0, 10] 10)

Some rewriting rules for the binders

**lemma** rewrite-B2-to-cases[simp]:
- shows B2 s t f = (case s of None ⇒ True | (Some s1) ⇒ (case t of None ⇒ True | (Some t1) ⇒ f s t1))
- unfolding B2-def B-def by(cases s,cases t,simp+)

**lemma** rewrite-B-None[simp]:
- shows None → α = True
- unfolding B-def by(auto)

**lemma** rewrite-B-m-True[simp]:
- shows m → (λ a . True) = True
- unfolding B-def by(cases m,simp+)

**lemma** rewrite-B2-cases:
- shows (case a of None ⇒ True | (Some s) ⇒ (case b of None ⇒ True | (Some t) ⇒ f s t))
  = (∀ s t . a = (Some s) ∧ b = (Some t) → f s t)
- by(cases a,simp,cases b,simp+)

**definition** strict-equal :: 'a option ⇒ 'a ⇒ bool

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

**end**

### 2.2 Theorems on lists

**theory** List-Theorems

**imports** Main

**begin**

**definition** lastn :: nat ⇒ 'a list ⇒ 'a list

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

**definition** is-sub-seq :: 'a ⇒ 'a ⇒ 'a list ⇒ bool

**where** is-sub-seq a b x ≡ ∃ n . Suc n < length x ∧ x||n = a ∧ x!(Suc n) = b

**definition** prefixes :: 'a list set ⇒ 'a list set

**where** prefixes s ≡ { x . ∃ n y . n > 0 ∧ y ∈ s ∧ 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 ≠ [] → length x ≥ 1 by(induct x,auto)

**lemma** take-but-one[simp]:
- shows x ≠ [] → lastn ((length x) - 1) x = tl x unfolding lastn-def

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

**lemma** Suc-m-minus-n[simp]:
- shows m ≥ n → Suc m - n = Suc (m - n) by auto
lemma lastn-one-less:
shows n > 0 ∧ n ≤ length x ∧ lastn n x = (a # y) → lastn (n - 1) x = y unfolding lastn-def
using drop-Suc[where n=length x - n and xs=x] drop-tl[where n=length x - n and xs=x]
by(auto)
lemma list-sub-implies-member:
shows ∀ a x. set (a # x) ⊆ Z → a ∈ Z by auto
lemma subset-smaller-list:
shows ∀ a x. set (a # x) ⊆ Z → set x ⊆ Z by auto
lemma second-elt-is-hd-tl:
shows tl x = (a # x') → a = x!1 by (cases x, auto)
lemma length-ge-2-implies-tl-not-empty:
shows length x ≥ 2 → tl x /= [] by (cases x, auto)
lemma length-lt-2-implies-tl-empty:
shows length x < 2 → tl x = [] by (cases x, auto)
lemma first-second-is-sub-seq:
shows length x ≥ 2 ⇒ is-sub-seq (hd x) (x!1) x
proof
assume length x ≥ 2
hence I: (Suc 0) < length x by auto
hence x!0 = hd x by(cases x,auto)
from this I show is-sub-seq (hd x) (x!1) x unfolding is-sub-seq-def by auto
qed
lemma hd-drop-is-nth:
shows n < length x ⇒ hd (drop n x) = x!n
proof (induct x arbitrary: n)
case Nil
  thus ?thesis by simp
next
case (Cons a x)
  have hd (drop n (a # x)) = (a # x)!n
  proof (cases n)
    case 0
      thus ?thesis by simp
    next
case (Suc m)
    from Suc Cons show ?thesis by auto
  qed
thus ?case by auto
qed
lemma def-of-hd:
shows y = a # x ⇒ hd y = a by simp
lemma def-of-tl:
shows y = a # x ⇒ tl y = x by simp
lemma drop-yields-results-implies-nbound:
shows drop n x ≠ [] ⇒ n < length x
by (induct x,auto)
lemma consecutive-is-sub-seq:
shows a # (b # x) = lastn n y ⇒ is-sub-seq a b y
proof
assume I: a # (b # x) = lastn n y
from I drop-Suc[where n=(length y) - n 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 \( \text{run} \) 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 take into account.

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

- time_t Time is modelled using natural numbers. Each atomic kernel action can be executed within one time unit.
short notations for using function control.

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

```plaintext
function run :: time-t ⇒ state-t option ⇒ (dom-t ⇒ action-t execution) ⇒ state-t option
where
run 0 s execs = s
| run (Suc n) None execs = None
| interrupt (Suc n) ⟹ run (Suc n) (Some s) execs = run n (Some (cswitch (Suc n) s)) execs
| ¬interrupt (Suc n) ⟹ thread-empty(execs (current s)) ⟹ run (Suc n) (Some s) execs = run n (Some s) execs
| ¬interrupt (Suc n) ⟹ ¬thread-empty(execs (current s)) ⟹ ¬precondition (next-state s execs) (next-action s execs) ⟹ run (Suc n) (Some s) execs = None
| ¬interrupt (Suc n) ⟹ ¬thread-empty(execs (current s)) ⟹ precondition (next-state s execs) (next-action s execs)

run (Suc n) (Some s) execs = run n (Some (step (next-state s execs) (next-action s execs))) (next-exec s execs)
using not0-implies-Suc by (metis option.exhaust prod-cases3,auto)
termination by lexicographic-order
end
```

### 3.2 SK (Separation Kernel)

theory SK

imports K

begin

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

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

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

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

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

**definition actions-in-execution:** 'action-t execution ⇒ 'action-t set

where actions-in-execution exec ≡ { a . ∃ aseq ∈ set exec . a ∈ set aseq }
locale Separation-Kernel = Kernel kstep output-f s0 current cswitch interrupt kprecondition realistic-execution control kinvolved

for kstep :: 'state-t ⇒ 'action-t ⇒ 'state-t
and output-f :: 'state-t ⇒ 'action-t ⇒ 'output-t
and s0 :: 'state-t
and current :: 'state-t ⇒ 'dom-t — Returns the currently active domain
and cswitch :: 'time-t ⇒ 'state-t ⇒ 'state-t — Switches the current domain
and interrupt :: 'time-t ⇒ bool — Returns true if an interrupt occurs in the given state at the given time
and kprecondition :: 'state-t ⇒ 'action-t ⇒ bool — Returns true if an precondition holds that relates the current action to the state
and realistic-execution :: 'action-t execution ⇒ bool — In this locale, this function is completely unconstrained.
and control :: 'state-t ⇒ 'dom-t ⇒ 'action-t execution ⇒ (('action-t option) × 'action-t execution × 'state-t)
and kinvolved :: 'action-t ⇒ 'dom-t set
+
fixes ifp :: 'dom-t ⇒ 'dom-t ⇒ bool
and vpeq :: 'dom-t ⇒ 'state-t ⇒ 'state-t ⇒ bool
assumes vpeq-transitive: ∀ a b c u. (vpeq u a b ∧ vpeq u b c) → vpeq u a c
and vpeq-symmetric: ∀ a b u. vpeq u a b → vpeq u b a
and vpeq-reflexive: ∀ a u. vpeq u a a
and ifp-reflexive: ∀ a. 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 u. ¬ifp (current s) u ∧ kprecondition s a → vpeq u s (kstep s a)
and output-consistent: ∀ a s t. vpeq (current s) s t ∧ current s = current t → (output-f s a) = (output-f t a)
and step-atomicity: ∀ s a 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 → fst (snd (control s (execs (current s)))) = fst (snd (control t (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: ∀ s a v. ∃ d ∈ (involved a) . kprecondition s (the a) → ifp d (current s)
and next-action-from-exec: ∀ s execs . next-action s execs → (∀ λ a . a ∈ actions-in-execution (execs s))
    ∧ next-execs-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.
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-f . run n (Some s0) (purge execs (current s-f)) →
    (λ s-f2 . 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. Invariently, 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. Invariently, 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 → (λ s t . (∀ 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 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 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 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) u

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

lemma vpeq-s-nt:
  assumes prec-t: precondition (next-state t execs2) (next-action t execs2)
  assumes not-ifp-curr-u: ¬ 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-refl v u

  from ifp-refl v u not-ifp-curr-u have unrelated: ¬ ifp-refl (current t) v using rtranclp-trans by metis
  from this current-next-state[THEN spec,THEN spec,where x1=t]
    locally-respects [THEN spec,THEN spec,THEN spec,where x1=next-state t execs2] vpeq-reflexive
    prec-s have vpeq v (next-state t execs2) (step (next-state t execs2) (next-action t execs2))
    unfolding step-def precondition-def B-def
    by (cases next-action t execs2,auto)
  from unrelated this locally-respects-next-state vpeq-transitive have vpeq v t (step (next-state t execs2) (next-action t execs2)) by blast
  from this and ifp-refl v u and vpeq-s-t and vpeq-symmetric and vpeq-transitive have vpeq v s (step (next-state t execs2) (next-action t execs2)) by metis
}
thus ?thesis by auto
qed

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

lemma vpeq-refl-fire:
  assumes prec-s: precondition (next-state s execs) (next-action s execs)
  assumes not-ifp-curr-u: ¬ ifp-refl (current s) u
  assumes vpeq-s-t: ∀ v . ifp-refl v u v t
  shows ∀ v . ifp-refl v u v t (step (next-state s execs) (next-action s execs))
proof -
{
  fix v
  assume ifp-refl v u

  from ifp-refl v u not-ifp-curr-u have unrelated: ¬ ifp-refl (current t) v using rtranclp-trans by metis
  from this current-next-state[THEN spec,THEN spec,where x1=t]
    locally-respects [THEN spec,THEN spec,THEN spec,where x1=next-state t execs2] vpeq-reflexive
    prec-s have vpeq v (next-state t execs2) (step (next-state t execs2) (next-action t execs2))
    unfolding step-def precondition-def B-def
    by (cases next-action s execs,auto)
  from unrelated this locally-respects-next-state vpeq-transitive have vpeq v s (step (next-state s execs) (next-action s execs)) by blast
  from this and ifp-refl v u and vpeq-s-t and vpeq-symmetric and vpeq-transitive have vpeq v s (step (next-state t execs) (next-action t execs)) by metis
}
thus ?thesis by auto
qed

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

lemma vpeq-refl-fire:
  assumes vpeq-s-t: ∀ v . ifp-refl v u v t'
  and current-s-t: current s = current t'
  shows precondition (next-state s execs) a ∧ precondition (next-state t' execs) a' (ifp-refl (current s) u) (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
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**: \( \text{vpeq-ns-nt-not-ifp-u} \)

**Assume** \( \text{purged-a-a2: purged-relation u execs execs2} \)

**And** \( \text{prec-s precondition (next-state s execs) (next-action s execs)} \)

**And** \( \text{current-s-t: current s = current t'} \)

**And** \( \text{vpeq-s-t: } \forall v. \text{ifp}^{\ast\ast} (current s) u \implies \text{vpeq} v s t' \)

**Shows** \( \neg \text{ifp}^{\ast\ast} (current s) u \land \text{precondition (next-state t' execs2)} \) \( \implies (\forall v. \text{ifp}^{\ast\ast} v u \implies \text{vpeq} v (\text{step (next-state s execs)} a) (\text{step (next-state t' execs2)} a)) \)

**Proof**:

\[
\begin{align*}
\text{assume} & \quad \text{not-ifp: } \neg \text{ifp}^{\ast\ast} (current s) u \\
\text{assume} & \quad \text{prec-t: precondition (next-state t' execs2) (next-action t' execs2)} \\
\text{fix} & \quad a' v \\
\text{assume} & \quad \text{ifp-v-uc: ifp}^{\ast\ast} v u \\
\text{from} & \quad \text{not-ifp and purged-a-a2 have } \neg \text{ifp}^{\ast\ast} (current s) u \text{ unfolding purged-relation-def by auto} \\
\text{from} & \quad \text{this and ifp-v-uc have not-ifp-curr-v: } \neg \text{ifp}^{\ast\ast} (current s) v \text{ using rtranclp-trans by metis} \\
\text{from} & \quad \text{this current-next-state[THEN spec,THEN spec,where } x1=s \text{ and } x2=\text{execs}2 \text{ prec-s vpeq-reflexive} \\
& \quad \text{locally-respects[THEN spec,THEN spec,THEN spec,where } x1=\text{next-state s execs and } x2=\text{the (next-action s execs)} \text{ and } x=v] \\
& \quad \text{have } \text{vpeq } v \text{ (next-state s execs) (step (next-state s execs) (next-action s execs))} \\
& \quad \text{unfolding step-def precondition-def B-def} \\
& \quad \text{by (cases next-action s execs,auto)} \\
\text{from} & \quad \text{not-ifp-curr-v this locally-respects-next-state vpeq-transitive} \\
& \quad \text{have } \text{vpeq-s-ns: vpeq v s (step (next-state s execs) (next-action s execs))} \\
& \quad \text{by blast} \\
\text{from} & \quad \text{not-ifp-curr-v current-s-t current-next-state[THEN spec,THEN spec,where } x1=t' \text{ and } x=\text{execs2}] \text{ prec-t} \\
& \quad \text{locally-respects[THEN spec,THEN spec,where } x=\text{next-state t' execs2} \text{ vpeq-reflexive} \\
& \quad \text{have } \theta: \text{vpeq v (next-state t' execs2) (step (next-state t' execs2) (next-action t' execs2))} \\
& \quad \text{unfolding step-def precondition-def B-def} \\
& \quad \text{by (cases next-action t' execs2,auto)} \\
\text{from} & \quad \text{not-ifp-curr-v current-s-t current-next-state have l: } \neg \text{ifp}^{\ast\ast} (current t') v \\
& \quad \text{using rtranclp-trans by auto} \\
& \quad \text{from } 0 \text{ 1 locally-respects-next-state vpeq-transitive} \\
& \quad \text{have } \text{vpeq-t-nt: vpeq v t' (step (next-state t' execs2) (next-action t' execs2))} \\
& \quad \text{by blast} \\
\text{from} & \quad \text{vpeq-s-ns and vpeq-t-nt and vpeq-s-t and ifp-v-uc and vpeq-symmetric and vpeq-transitive} \\
& \quad \text{have } \text{vpeq-s-ns: vpeq v (step (next-state s execs) (next-action s execs)) (step (next-state t' execs2) (next-action t' execs2))} \\
& \quad \text{by blast} \\
\text{thus } & \quad \text{thesis by auto} \\
\text{qed} 
\end{align*}
\]
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 : interrupt (Suc n) 
  assume IH : (\forall u execs2. 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-rs-rt). 
    from current-s-t cswitch-independent-of-state 
    have current-ns-nt: current (cswitch (Suc n) s) = current (cswitch (Suc n) t') by blast 
    from cswitch-consistency vpeq-s-t 
    have vpeq-ns-nt : \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-ns-rt: current rs = current rt using rs rt by(auto) 
    { 
      fix v 
      assume ia : ifp^∗∗ v u 
      from current-ns-nt vpeq-ns-nt ia interrupt-s vpeq-reflexive purged-a-a2 IH[where u=u and t=Some (cswitch (Suc n) t') and ?execs2.0=execs2] 
      have vpeq-rs-rt: vpeq v rs rt using rs rt by(auto) 
    } 
    from current-ns-rt and this have equivalent-states (Some rs) (Some rt) u by auto 
    } 
  thus ?case by(simp add:option.splits,cases t,simp+) 
  next 
  case (4 n execs s t u execs2) 
}
assume not-interrupt. ¬interrupt (Suc n)
assume thread-empty-s. thread-empty(execs (current s))
assume IH: (\forall u execs2. equivalent-states (Some s) t u \land purged-relation u execs execs2 \implies equivalent-states (run n (Some s) execs) (run n t execs2) u)
{
  fix t'
  assume t: t = Some t'
  fix rs
  assume rs: run (Suc n) (Some s) execs = Some rs
  fix rt
  assume rt: run (Suc n) (Some t') execs2 = Some rt

  assume vpeq-s-t: \forall v. ifp^v u 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 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 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 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).

  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 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 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 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).

  IH[where t=Some t' and u=u and ?execs2.0=?execs2]
  have equivalent-states (run n (Some s) execs) (run n (Some t') execs2) u using rs rt by (auto)
  from this not-interrupt t-empty thread-empty-s
  show ?thesis using rs rt by (auto)
  next

  case t-not-empty
  from t-not-empty current-next-state and vpeq-s-t-u and thread-empty-s and purged-a-a2 and current-s-t
  have not-ifp-curr-t. ¬ifp^v s t (next-state t' execs2) u unfolding purged-relation-def by auto
  show ?thesis
  proof (cases precondition (next-state t' execs2) (next-action t' execs2) rule :case-split[case-names t-prec t-not-prec])

  case t-prec
  from locally-respects-next-state current-next-state t-prec not-ifp-curr-t vpeq-s-t locally-respects vpeq-s-nt
  have vpeq-s-nt: \forall v. ifp^v s t (next-state t' execs2) (next-action t' execs2)) by auto
  from vpeq-s-nt purged-a-a2 this current-s-nt not-ifp-curr-t current-next-state
  IH[where t=Some (step (next-state t' execs2) (next-action t' execs2)) and u=u and ?execs2.0=?execs2]
\[ t' \text{execs2} \]

- \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}

next

\text{case t-not-prec}

\text{thus ?thesis using rt t-not-empty not-interrupt by(auto)}

qed

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

next

\text{case t-not-empty}

\text{show ?thesis}

\text{proof (cases precondition (next-state t' execs2) (next-action t' execs2) rule :case-split[case-names t-prec t-not-prec])}
\text{case t-prec}

\text{from 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}
\text{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: ('t 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{IH[where t=Some (step (next-state t' execs2) (next-action t' execs2)) and u=u and ?execs2.0=next-execs}
\text{t' execs2) (next-execs 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}

next

\text{case t-not-prec}

\text{thus ?thesis using rt t-not-empty not-interrupt by(auto)}

qed

\}

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

\}

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

next

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

- Whenever the precondition does not hold, the entire theorem flattens to True and everything holds vacuously.
hence run (Suc n) (Some s) execs = None using not-interrupt thread-not-empty-s by simp
thus ?case by(simp add?option.splits)
next
case (6 n execs s t u execs2)
assume not-interrupt: ~interrupt (Suc n)
assume thread-not-empty-s: ~thread-empty (execs (current s))
assume prec-s: precondition (next-state s execs) (next-action s execs)
assume IH: (∀ u execs2, 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) ) (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^∗∗ v u → vpeq v s t'
  assume current-s-t: current s = current t'
  assume purged-a-a2: purged-relation u execs execs2

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

  — We prove two properties: the states rs and rt have equal active domains (current-rs-rt) and are vpeq for all domains v that may influence u (vpeq-rs-rt). Both are proven using the IH. To use the IH, we have to prove that the properties hold for the next step (in this case, 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) t 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 → (∀ 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 autometis converse-rtranclp-into-rtranclp)
from current-s-t next-execs-consistent vpeq-curr-s-t vpeq-involved
  have next-execs-t: ifp^∗∗ (current s) u → next-execs t' execs = next-execs s execs
unfolding next-execs-def by(auto)
from current-s-t purged-a-a2 thread-not-empty-s next-action-consistent[THEN spec.THEN spec,where x1=s and x2=t]
  vpeq-curr-s-t vpeq-involved
have next-action-s-t: ifp^∗∗ (current s) u → next-action t' execs2 = next-action s execs
by (unfold next-action-def, unfold purged-relation-def auto)
from purged-a-a2 current-s-t next-execs-consistent[THEN spec.THEN spec,THEN spec,where x2=s and x1=t'
and x=execs]
  vpeq-curr-s-t vpeq-involved
have purged-na-na2: purged-relation u (next-exec s execs) (next-exec t 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:
t f p^\ast\ast\ (current s) u \rightarrow \neg \text{thread-empty}(execs2 (current t')) by auto
from step-atomicity current-s-t current-next-state have current-s-nt: 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: \neg \text{ifp}^\ast\ast\ (current s) u \rightarrow \text{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^\ast\ast\ (current s) u rule : case-split[case-names curr-ifp-u curr-not-ifp-u])
case curr-ifp-u
show ?thesis
proof (cases precondition (next-state t' execs2) (next-action t' execs2) rule : case-split[case-names prec-t prec-not-t])
case prec-t
have thread-not-empty-t: \neg \text{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 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[case-names t-empty t-not-empty])
case t-not-empty
show ?thesis
proof (cases precondition (next-state t' execs2) (next-action t' execs2) rule : case-split[case-names t-prec t-not-prec])
case t-prec
from curr-not-ifp-u t-prec IH[where u=u and ?execs2.0=(next-exec s execs2) and t=Some (step
\[(\text{next-state } t' \text{ execs2}) (\text{next-action } t' \text{ execs2})])
\]
\[\text{current-ns-nt next-execx-t purged-na-na2 vpeq-ns-nt-not-ifp-u current-s-t vpeq-s-t prec-s purged-a-a2 have equivalent-states (run n (Some (step (next-state s execs) (next-action s execs))) (next-execx s execs)) (run n (Some (step (next-state t' execs2) (next-action t' execs2))) (next-execx t' execs2))}\]
\[u \text{ by auto}\]
\[\text{from this } t\text{-prec curr-not-ifp-u t-not-empty prec-s not-interrupt thread-not-empty-s show } ?\text{thesis using } rs rt \text{ by auto}\]
\[\text{next case } t\text{-not-prec from } t\text{-not-prec } t\text{-not-empty not-interrupt show } ?\text{thesis using } rt \text{ by simp qed}\]
\[\text{next case } t\text{-empty from curr-not-ifp-u and prec-s and vpeq-s-t and locally-respects and vpeq-ns-t current-next-state locally-respects-next-state have vpeq-ns-t (} v \cdot \text{ ifp}^\ast v u \rightarrow \text{ vpeq } v \text{ (step (next-state s execs) (next-action s execs)) } t') by blast\]
\[\text{from curr-not-ifp-u IH[where } t=\text{Some } t' \text{ and } u=u \text{ and } ?\text{execs2.0=execs2] and current-ns-t and next-execx-t and purged-na-a-a and vpeq-ns-t and this have equivalent-states (run n (Some (step (next-state s execs) (next-action s execs))) (next-execx s execs)) (run n (Some (t') execs2) u \text{ by auto}\]
\[\text{from this } \text{not-interrupt thread-not-empty-s } t\text{-empty prec-s show } ?\text{thesis using } rs rt \text{ by auto qed}\]
\[\text{qed}\]
\[\{\text{fix } v\]
\[\text{assume } ia: \text{ ifp}^\ast v u\]
\[\text{have vpeq v rs rt proof (cases ifp}^\ast (current s) u \text{ rule } \text{case-split[case-names curr-ifp-u curr-not-ifp-u]}\]
\[\text{case curr-ifp-u show } ?\text{thesis proof (cases precondition (next-state t' execs2) (next-action t' execs2) rule } \text{case-split[case-names t-prec t-not-prec]}\]
\[\text{case t-prec have thread-not-empty-t: } \rightarrow \text{thread-empty}(execx2 (current t')) using thread-not-empty-t curr-ifp-u by auto from current-ns-nt next-execx-t next-action-s-t purged-a-a2 curr-ifp-u t-prec prec-s vpeq-ns-nt-ifp-u[where } a=(\text{next-action s execs}) \text{ vpeq-s-t current-s-t have equivalent-states (Some (step (next-state s execs) (next-action s execs))) (Some (step (next-state t' execs2) (next-action t' execs2))) u unfolding purged-relation-def next-state-def by auto from this IH[where } a=u \text{ and } ?\text{execs2.0=(next-execx t' execs2) and } t=\text{Some (step (next-state t' execs2) (next-action t' execs2))]}\]
\[\text{current-ns-nt purged-na-na2 have equivalent-states (run n (Some (step (next-state s execs) (next-action s execs))) (next-execx s execs)) (run n (Some (t') execs2) (next-action t' execs2))) (next-execx t' execs2)) u by auto from ia curr-ifp-u t-prec thread-not-empty-t prec-s and this and not-interrupt and thread-not-empty-s and next-action-s-t show } ?\text{thesis using } rs rt \text{ by auto}\]
\[\text{next}\]
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.
D31.1 – Formal Specification of a Generic Separation Kernel

{ 
  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 run n s execs ∥ run n t (purge execs u) ⇀ (λrs rt. vpeq u rs rt ∧ current rs = current rt)
  unfolding B-def
  by(cases run n s execs,simp,cases run n t (purge execs u),simp,auto)
}
thus ?thesis unfolding view-partitioned-def Let-def by auto
qed

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

```
theorem unwinding-implies-NI-unrelated:
shows NI-unrelated
proof－
{
  fix execs a n
  from unwinding-implies-view-partitioned
  have vp: view-partitioned by blast
  from vp and vpeq-reflexive
  have 1: ∀ u . (run n (Some s0) execs ∥ run n (Some s0) (purge execs u))
            ⇀ (λrs rt. vpeq u rs rt ∧ current rs = current rt))
  unfolding view-partitioned-def by auto
  have run n (Some s0) execs ↦ (λs-f. run n (Some s0) (purge execs (current s-f)) ↦ (λs-f2. output-f s-f a =
                                output-f s-f2 a ∧ current s-f = current s-f2))
  proof(cases run n (Some s0) execs)
  case None
    thus ?thesis unfolding B-def by simp
  next
  case (Some rs)
    thus ?thesis
  proof(cases run n (Some s0) (purge execs (current rs)))
  case None
    from Some this show ?thesis unfolding B-def by simp
  next
  case (Some rt)
    from «run n (Some s0) execs = Some rs> Some I[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 ipurge functions. The first purges all domains $d$ that are intermediary for some other domain $v$. An intermediary for $u$ is defined as a domain $d$ for which there exists an information flow from some domain $v$ to $u$ via $d$, but no direct information flow from $v$ to $u$ is allowed.

**definition intermediary :: 'dom-t ⇒ 'dom-t ⇒ bool**
**where** intermediary $d u \equiv \exists v . \text{ifp} v d u \land \neg \text{ifp} v u \land d \neq u$

**primrec remove-gateway-communications :: 'dom-t ⇒ 'action-t execution ⇒ 'action-t execution**
**where** remove-gateway-communications $u [] = []$

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

**definition ipurge-l ::**
\[
\text{('dom-t ⇒ 'action-t execution) ⇒ 'dom-t ⇒ ('dom-t ⇒ 'action-t execution) where}
\]
\[
\text{ipurge-l execs } u \equiv \lambda d . \text{if intermediary } d u \text{ then}
\]

\[
\[
\text{else if } d = u \text{ then}
\]
\[
\text{remove-gateway-communications } u \text{ (execs } u)\]

else execs $d$

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

**abbreviation ind-source :: 'dom-t ⇒ 'dom-t ⇒ bool**
**where** ind-source $d u \equiv \text{ifp} v d u \land \neg \text{ifp} v u \land d \neq u$

**definition ipurge-r ::**
\[
\text{('dom-t ⇒ 'action-t execution) ⇒ 'dom-t ⇒ ('dom-t ⇒ 'action-t execution) where}
\]
\[
\text{ipurge-r execs } u \equiv \lambda d . \text{if intermediary } d u \text{ then}
\]

\[
\[
\text{else if } \text{ind-source } d u \text{ then}
\]
\[
\text{SOME alpha . realistic-execution alpha}
\]

else if $d = u$ then

\[
\text{remove-gateway-communications } u \text{ (execs } u)\]

else execs $d$

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

**definition NI-indirect-sources ::bool**
**where** NI-indirect-sources $\equiv \forall \text{ execs } a n . \text{run } n (\text{Some } s0) \text{ execs } (\lambda s-l . \text{run } n (\text{Some } s0) \text{ (ipurge-l execs } (\text{current s-f})) \parallel \text{run } n (\text{Some } s0) \text{ (ipurge-r execs } (\text{current s-f})) \rightarrow
\]

\[
(\lambda s-l s-r . \text{output-f s-l a } = \text{output-f s-r a}))
\]

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

This allows us to define security over intransitive policies.

**definition insecure ::bool**
**where** insecure $\equiv$ NI-indirect-sources $\land$ NI-unrelated

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

**definition** `does-not-communicate-with-gateway`

where `does-not-communicate-with-gateway u execs ≡ (∀ a . a ∈ actions-in-execution (execs u) → (∀ v . intermediary v u → v ≠ involved (Some a)))``

**definition** `iview-partitioned :: bool` where `iview-partitioned ≡ ∀ execs ms mt n u . iequivalent-states ms mt u ↦ (run n ms (ipurge-l execs u)) ∥ run n mt (ipurge-r execs u) ↦ (λ rs rt . vpeq u rs rt ∧ current rs = current rt))``

**definition** `ipurged-relation1 :: dom-t ⇒ dom-t ⇒ action-t execution ⇒ dom-t ⇒ action-t execution ⇒ bool` where `ipurged-relation1 u execs1 execs2 ≡ (∀ d . ifp d u = execs1 d ∧ intermediary d u = 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 : ∀ v . ifp v u ∧ ¬intermediary v u → vpeq v s t` and `prec-s : precondition (next-state s execs) (next-action s execs)``

shows `∀ d ∈ involved (next-action s execs) . vpeq d s t` proof---

{ fix v assume involved : v ∈ 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 ∧ ¬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 [fp-curr not-intermediary-v have curr-is-u : current s = u using rtranclp-trans r-into-rtranclp by (metis intermediary-def)] from curr-is-u next-action-from-exec[THEN spec,THEN spec,where x=execs and x1=s] not-intermediary-v involved no-gateway-comm[unfolded does-not-communicate-with-gateway-def,THEN spec,where x=the (next-action s execs)] have False unfolding involved-def B-def by (cases next-action s execs,auto) hence vpeq v s t by auto }
moreover
{
assume intermediary-v: ¬ ifp v u
from ifp-curr not-intermediary-curr ifp-v-curr intermediary-v
  have False unfolding intermediary-def by auto
  hence vpeq v s t' by auto
}
ultimately
show vpeq v s t' unfolding intermediary-def by auto
qed
}
thus thesis by auto
qed

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

lemma ipurge-l-removes-gateway-communications:
shows does-not-communicate-with-gateway u (ipurge-l execs u)
proof-
{
  fix aseq u execs a v
  assume 1: aseq ∈ set (remove-gateway-communications u (execs u))
  assume 2: a ∈ set aseq
  assume 3: intermediary v u
  have 4: v ∉ involved (Some a)
  proof-
  {
    fix a: action t
    fix aseq u exec v
    have aseq ∈ set (remove-gateway-communications u exec) ∧ a ∈ set aseq ∧ intermediary v u →→ v ∉ 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 and uses the same convention for naming.

lemma iunwinding-implies-view-partitioned1:
shows iview-partitioned
proof-
{
  fix u execs execs2 s t n
  have does-not-communicate-with-gateway u execs ∧ iequivalent-states s t u ∧ ipurged-relation1 u execs execs2
    → 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: (\(\forall t \in \text{execs2}, \text{does-not-communicate-with-gateway} u \text{ execs} \land
\text{iequivalent-states (Some (cswitch (Suc n) s)) t u \land \text{ipurged-relation1} u \text{ execs} \text{ execs2} \rightarrow
\text{iequivalent-states (run n (Some (cswitch (Suc n) s)) execs) (run n t execs2) u})\)

\[
\begin{cases}
\text{fix } t' = \text{state-t} \\
\text{assume } t = \text{Some } t' \\
\text{fix } rs \\
\text{assume } rs: \text{run (Suc n) (Some s) execs = Some rs} \\
\text{fix } rt \\
\text{assume } rt: \text{run (Suc n) (Some t') execs2 = Some rt}
\end{cases}
\]

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

\[
\begin{cases}
\text{fix } t' \\
\text{assume t: t = Some t' } \\
\text{fix } rs \\
\text{assume rs: run (Suc n) (Some s) execs = Some rs} \\
\text{fix } rt \\
\text{assume rt: run (Suc n) (Some t') execs2 = Some rt}
\end{cases}
\]
assume no-gateway-comm: does-not-communicate-with-gateway u execs
assume vpeq-s-t: \(\forall v. \text{ifp v u} \land \neg \text{intermediary v u} \rightarrow vpeq v s t'\)
assume current-s-t: current s = current t'
assume purged-a-a2: ipurred-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 inequivalent-states (run (Suc n) (Some s) execs) (run (Suc n) (Some t') execs2) u
proof (cases thread-empty(execs2 (current t')))

case True
from purged-a-a2 and vpeq-s-t and current-s-t IH[where t=Some t' and u=u and ?execs2.0=?execs2] no-gateway-comm
have inequivalent-states (run n (Some s) execs) (run n (Some t') execs2) u using rs rt by(auto)
from this not-interrupt True thread-empty-s
show ?thesis using rs rt by(auto)

next
case False
have prec-t: precondition (next-state t' execs2) (next-action t' execs2)
proof
\
\{ 
\hspace*{1cm} assume not-prec-t: ¬precondition (next-state t' execs2) (next-action t' execs2) 
\hspace*{1cm} hence run (Suc n) (Some t') execs2 = None using not-interrupt False not-prec-t by (simp)
\hspace*{1cm} from this have False using rt by(simp add:option.splits)
\}
\hspace*{1cm} 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 ipurved-relation1-def intermediary-def by auto
\{ 
\hspace*{1cm} fix v 
\hspace*{1cm} assume ifp-v: ifp v u 
\hspace*{1cm} 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,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)
\}
\hspace*{1cm} 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: ipurved-relation1 u execs (next-execst t' execs2)
unfolding ipurved-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-execst t' execs2) and u=u]
and current-s-nt purged-a-na2
have eq-ns-nt: iequivalent-states (run n (Some s) execs)
(run n (Some (step (next-state t' execs2) (next-action t' execs2))) (next-execs t' execs2)) u by auto
from prec-t eq-ns-nt not-interrupt False thread-empty-s
show ?thesis using t rs rt by(auto)
qed
}
thus ?case by(simp add:option.splits,cases t simp+)
next
case (5 n execs s t u execs2)
assume not-interrupt: ¬interrupt (Suc n)
assume thread-not-empty-s: ¬thread-empty(execs (current s))
assume not-prec-s: ¬precondition (next-state s execs) (next-action s execs)
hence run (Suc n) (Some s) execs = None using not-interrupt thread-not-empty-s by simp
thus ?case by(simp add:option.splits)
next
case (6 n execs s t u execs2)
assume not-interrupt: ¬interrupt (Suc n)
assume thread-not-empty-s: ¬thread-empty(execs (current s))
assume prec-s: precondition (next-state s execs) (next-action s execs)
assume IH: (∀t u execs2. does-not-communicate-with-gateway u (next-execs s execs) ∧
iequivalent-states (Some (step (next-state s execs) (next-action s execs))) t u ∧
ipurged-relation1 u (next-execs s execs) execs2 →
iequivalent-states
(ran n (Some (step (next-state s execs) (next-action s execs))) (next-execs s execs))
(ran n t execs2) u)
{
fix t'
assume t: t = Some t'
fix rs
assume rs: run (Suc n) (Some s) execs = Some rs
fix rt
assume rt: run (Suc n) (Some t') execs2 = Some rt
assume no-gateway-comm: does-not-communicate-with-gateway u execs
assume vpeq-s-t: ∀ v. ifp u v ∧ ¬intermediary v u → vpeq v s t'
assume current-s-t: current s = current t'
assume purged-a-a2: ipurged-relation1 u execs execs2
from ifp-reflexive vpeq-s-t have vpeq-u-s-t: vpeq u s t' unfolding intermediary-def by auto
from step-atomicity and current-s-t current-next-state
have current-ns-nt: current (step (next-state s execs) (next-action s execs)) = current (step (next-state t'
execs2) (next-action t' execs2)) unfolding step-def
by (cases next-action s execs,cases next-action t' execs2,simp,simp,cases next-action t' execs2,simp,simp)
from step-atomicity current-next-state current-s-t have current-ns-t: current (step (next-state s execs) (next-action s execs)) = current t'
unfolding step-def
by (cases next-action s execs,auto)
from vpeq-s-t have vpeq-curr-s-t: ifp (current s) u ∧ ¬intermediary (current s) u → vpeq (current s) s t'
unfolding intermediary-def by auto
from current-s-t purged-a-a2
have eq-execs ifp (current s) u ∧ ¬intermediary (current s) u → execs (current s) = execs2 (current s)
by (auto simp add: ipurged-relation1-def)
from vpeq-involved-domains no-gateway-comm vpeq-s-t vpeq-involved-domains prec-s
have vpeq-involved: ifp (current s) u \land \sim \text{intermediary} (current s) u \longrightarrow (\forall \ d \in \text{involved} \ (\text{next-action} \ s \ \text{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 \neq t' \ and \ s \ = \ \text{execs}]

vpeq-curr-s-t vpeq-involved

have next-execs-t: ifp (current s) u \land \sim \text{intermediary} (current s) u \longrightarrow next-execs t' execs = next-execs s execs

by(auto simp add: next-execs-def)

from current-s-t \ and \ purified-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 \land \sim \text{intermediary} (current s) u \longrightarrow next-action t' execs2 = next-action s execs

by(unfold next-action-def,unfold ipurged-relation1-def,auto)

from purified-a-a2 \ and \ thread-not-empty-s \ and \ current-s-t

have thread-not-empty-t: ifp (current s) u \land \sim \text{intermediary} (current s) u \longrightarrow \sim \text{thread-empty}(execs2 (current t'))

unfolding ipurged-relation1-def by auto

have vpeq-ns-nt-1: \ (\forall \ a \ . \ \text{precondition} \ (\text{next-state} \ s \ \text{execs}) \ a \land \ \text{precondition} \ (\text{next-state} \ t' \ \text{execs}) \ a \longrightarrow ifp (current s) u \land \sim \text{intermediary} (current s) u \longrightarrow (\forall \ v \ . \ ifp \ v \ u \land \sim \text{intermediary} v \ u \longrightarrow vpeq v \ (\text{step} \ (\text{next-state} \ s \ \text{execs})) \ a) \ (\text{step} \ (\text{next-state} \ t' \ \text{execs}) \ a))

proof-

fix a

assume ifp-a: ifp (current s) u \land \sim \text{intermediary} (current s) u

from ifp-a

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 \ v \ u \land \sim \text{intermediary} v \ u \longrightarrow vpeq v \ (\text{step} \ (\text{next-state} \ s \ \text{execs})) \ a) \ (\text{step} \ (\text{next-state} \ t' \ \text{execs}) \ a)

unfolding step-def precondition-def B-def

by (cases a,auto)

dqe

have no-gateway-comm-na: \ \text{does-not-communicate-with-gateway} u \ (\text{next-execs} \ s \ \text{execs})

proof-

{ fix a

assume a \in \text{actions-in-execution} \ (\text{next-execs} \ \text{execs} \ u)

from this no-gateway-comm[unfolded \text{does-not-communicate-with-gateway-def},THEN spec, where \ x = a]

\text{next-execs-subset} \ (THEN spec, \text{THEN spec, where} \ x2 = s \ \text{and} \ x1 = \text{execs} \ and \ x0 = u]

have \ (\forall \ v \ . \ \text{intermediary} v \ u \longrightarrow v \notin \text{involved} \ (\text{Some} \ a))

unfolding actions-in-execution-def

by(auto)
}

dqe

have inequivalent-states \ (run \ (Suc \ n) \ (\text{Some} \ s) \ \text{execs}) \ (run \ (Suc \ n) \ (\text{Some} \ t') \ \text{execs}2) \ u

proof (cases ifp (current s) u \land \sim \text{intermediary} (current s) u \ \text{rule :case-split[case-names \ T F]})

case T

show \ ?thesis

proof (cases \text{thread-empty}(execs2 (current t')) \ \text{rule :case-split[case-names \ T2 F2])}

case F2

show \ ?thesis

proof (cases \text{precondition} \ (\text{next-state} \ t' \ \text{execs}2) \ (\text{next-action} \ t' \ \text{execs}2) \ \text{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: \ \text{ipurged-relation1} \ u \ (\text{next-execs} \ s \ \text{execs}) \ (\text{next-execs} \ t' \ \text{execs}2)
unfolding ipurged-relation1-def next-exec-def by auto
from HH[where t=Some (step (next-state t' execs2) (next-action t' execs2)) and ?execs2.0=next-exec t'
execs2 and u=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-exec 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-exec s execs))
(run n (Some (step (next-state t' execs2) (next-action t' execs2))) (next-exec t'
execs2)) u
unfolding next-state-def by (auto,metis)
from this not-interrupt thread-not-empty-s prec-s F2 T3 have current-rs-rt∶current rs = current rt using rs rt by auto
{ fix v
assume ia∶ifp v u ∧ ¬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
next
next
next
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-exec t' execs2
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 ifp-$v$: ifp $v$ $u$
assume $v$-not-intermediary: $\neg$intermediary $v$ $u$

have not-ifp-curr-$v$: $\neg$ifp (current $s$) $v$
proof
assume ifp-curr-$v$: ifp (current $s$) $v$
thus False
proof
{ assume ind-source (current $s$) $u$
  from this ifp-curr-$v$ ifp-$v$ have intermediary $v$ $u$ unfolding intermediary-def by auto
  from this $v$-not-intermediary have False unfolding intermediary-def by auto
}
moreover
{ assume unrelated: unrelated (current $s$) $u$
  from this ifp-$v$ ifp-curr-$v$ have False using rtranclp-trans r-into-rtranclp by metis
}
ultimately show ?thesis using 1 2 by auto
qed

from this current-next-state [THEN spec, THEN spec, where $x1 = s$ and $x = \text{execs}$] prec-$s$
locally-respects [THEN spec, THEN spec, where $x = \text{next-state s execs}$] vpeq-reflexive
have vpeq $v$ (next-state $s$ execs) (step (next-state $s$ execs) (next-action $s$ execs)) unfolding step-def precondition-def B-def
by (cases next-action $s$ execs,auto)
from not-ifp-curr-$v$ this locally-respects-next-state vpeq-transitive
have vpeq-$s$-$ns$: vpeq $v$ $s$ (step (next-state $s$ execs) (next-action $s$ execs)) by blast

from not-ifp-curr-$v$ current-$s$-$t$ current-next-state [THEN spec, THEN spec, where $x1 = t'$ and $x = \text{execs2}$] prec-$t$
locally-respects [THEN spec, THEN spec, where $x = \text{next-state t' execs2}$] $F$ vpeq-reflexive
have $\theta$: $\neg$ thread-empty ($\text{execs2}$ (current $t'$)) $\rightarrow$ vpeq $v$ (next-state $t'$ execs2) (step (next-state $t'$ execs2) (next-action $t'$ execs2)) unfolding step-def precondition-def B-def
by (cases next-action $t'$ execs2,auto)
from 0 not-ifp-curr-$v$ current-$s$-$t$ locally-respects-next-state [THEN spec, THEN spec, THEN spec, where $x2 = t'$ and $x1 = v$ and $x = \text{execs2}$] vpeq-transitive
have vpeq-$t$-$nt$: $\neg$ thread-empty ($\text{execs2}$ (current $t'$)) $\rightarrow$ vpeq $v$ $t'$ (step (next-state $t'$ execs2) (next-action $t'$ execs2)) by metis
from this vpeq-reflexive
have vpeq-$t$-$nt$: vpeq $v$ $t'$ ?nt
by auto
from vpeq-$s$-$t$ ifp-$v$ v-not-intermediary
have vpeq $v$ $s$ $t'$ by auto
from this vpeq-$s$-$ns$ vpeq-$t$-$nt$ vpeq-transitive vpeq-symmetric vpeq-reflexive
have vpeq $v$ (step (next-state $s$ execs) (next-action $s$ execs)) ?nt
by (metis (opaque-lifting, no-types))
}

hence vpeq-$ns$-$nt$: $\forall$ $v$. ifp $v$ $u$ $\wedge$ $\neg$intermediary $v$ $u$ $\rightarrow$ vpeq $v$ (step (next-state $s$ execs) (next-action $s$ execs)) ?nt by auto
from vpeq-$s$-$t$ $2$ $F$ purged-a-a2 current-$s$-$t$ thread-not-empty-$s$ have purged-na-na2: ipurged-relation1 $u$ (next-execs $s$ execs) ?na2
unfolding ipurged-relation1-def next-execs-def intermediary-def by(auto)
from current-$ns$-$nt$ current-$ns$-$t$ current-next-state have current-$ns$-$nt$:
current (step (next-state s execs) (next-action s execs)) = current ?nt
by auto
from prec-s vpeq-ns-nt no-gateway-comm-na
and IH[where t=Some ?nt and ?execs2.0= ?na2 and u=u]
and current-ns-nt purged-na-na2
have eq-ns-nt : inequivalent-states (run n (Some (step (next-state s execs) (next-action s execs))) (next-exec s execs))
by auto
from this not-interrupt thread-not-empty-s prec-t prec-s
have current-rs-rt : current rs = current rt using rs rt by (cases thread-empty (execs2 (current t'))),simp,simp)
{
fix v
assume ia : ifp v u ∧ ¬intermediary v u
from this eq-ns-nt not-interrupt thread-not-empty-s prec-t
have vpeq v rs rt
using rs rt by (cases thread-empty(execs2 (current t')),simp,simp)
}
from current-rs-rt and this show :thesis using rs rt by auto
qed
qed
}
thus case by(simp add:option.splits,cases t,simp+)
qed
}

hence iview-partitioned-inductive : ∀ u s t execs execs2 n. does-not-communicate-with-gateway u execs ∧ inequivalent-states s t u ∧ ipurged-relation1 u execs execs2 !→ inequivalent-states (run n s execs) (run n t execs2) u
by blast
have ipurged-relation : ∀ u execs . ipurged-relation1 u (ipurge-l execs u) (ipurge-r execs u)
by(unfold ipurged-relation1-def ,unfold ipurge-l-def,unfold ipurge-r-def ,auto)
{
fix execs s t n u
assume I : inequivalent-states s t u
from ifp-reflexive
have dir-source : ∀ u . ifp u u ∧ ¬intermediary u u unfolding intermediary-def by auto
from ipurge-l-removes-gateway-communications
have does-not-communicate-with-gateway u (ipurge-l execs u)
by auto
from I this iview-partitioned-inductive ipurged-relation
have inequivalent-states (run n s (ipurge-l execs u)) (run n t (ipurge-r execs u)) u by auto
from this dir-source
have run n s (ipurge-l execs u) || run n t (ipurge-r execs u) → (λrs rt. vpeq u rs rt ∧ current rs = current rt)
using r-into-rtranclp unfolding B-def
by(cases run n s (ipurge-l execs u).simp,cases run n t (ipurge-r execs u).simp,auto)
}
thus :thesis unfolding iview-partitioned-def Let-def by auto
qed


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

definition mcurrents :: state-t option ⇒ state-t option ⇒ bool
where mcurrents m1 m2 ≡ m1 || m2 → (λs t . current s = current t)

Proof that switching/interrupts are purely time-based and happen independent of the actions done by
the domains. As all theorems in this locale, it holds vacuously whenever one of the states is None, i.e.,
whenever at some point a precondition does not hold.
\textbf{lemma} current-independent-of-domain-actions:
\textbf{assumes} current-s-t: mcurrents s t
\textbf{shows} mcurrents \ ((run n s execs) \ (run n t execs2))
\textbf{proof}-
\begin{enumerate}
\item \textbf{fix} n s execs t execs2
\item \textbf{have} mcurrents s t \rightarrow mcurrents \ ((run n s execs) \ (run n t execs2))
\item \textbf{proof} \ ((\text{induct} n s execs \text{ arbitrary:} \ t \text{ execs2 rule:} \ run.induct)
\item \textbf{case} (1 s execs t execs2)
\item \textbf{from this} show ?case \textbf{using} current-s-t \textbf{unfolding} B-def \textbf{by} \textbf{auto}
\item \textbf{next}
\item \textbf{case} (2 n execs t execs2)
\item \textbf{show} ?case \textbf{unfolding} mcurrents-def \textbf{by}(\text{auto})
\item \textbf{next}
\item \textbf{case} (3 n s execs t execs2)
\item \textbf{assume} interrupt: interrupt (Suc n)
\item \textbf{assume} IH: \ ((\lambda t \text{ execs2}. \ mcurrents \ (\text{Some} \ (\text{cswitch} \ (Suc n) \ s)) \ t \rightarrow mcurrents \ (\text{run} n \ (\text{Some} \ (\text{cswitch} \ (Suc n) \ s)) \ s)) \ execs) \ (\text{run} n \ t \ \text{execs2}))
\item \textbf{fix} t'
\item \textbf{assume} t: t = (Some t')
\item \textbf{assume} curr: mcurrents (Some s) t
\item \textbf{from} \ t \text{ curr} \text{ cswitch-independent-of-state[THEN spec,THEN spec,THEN spec,where x1=s] have} current-ns-nt: current (\text{cswitch} \ (Suc n) \ s) = current (\text{cswitch} \ (Suc n) \ t')
\item \textbf{unfolding} mcurrents-def \textbf{by simp}
\item \textbf{from} current-ns-nt \text{IH[where t=Some (cswitch (Suc n) t') and ?execs2.0=execs2] have} mcurrents-ns-nt: mcurrents (\text{run} n \ (\text{Some} \ (\text{cswitch} \ (Suc n) \ s)) \ execs) \ (\text{run} n \ (\text{Some} \ (\text{cswitch} \ (Suc n) \ t')) \ execs2)
\item \textbf{unfolding} mcurrents-def \textbf{by}(\text{auto})
\item \textbf{from} mcurrents-ns-nt interrupt t
\item \textbf{have} mcurrents \ (\text{run} \ (Suc n) \ (\text{Some} s) \ execs) \ (\text{run} \ (Suc n) \ t \ \text{execs2})
\item \textbf{unfolding} mcurrents-def B2-def B-def \textbf{by}(\text{cases \ run} n \ (\text{Some} \ (\text{cswitch} \ (Suc n) \ s)) \ execs, \ \text{cases \ run} \ (Suc n) \ t \ \text{execs2,auto})
\item \textbf{thus} ?case \textbf{unfolding} mcurrents-def B2-def \textbf{by}(\text{cases t,auto})
\item \textbf{next}
\item \textbf{case} (4 n execs s t execs2)
\item \textbf{assume} not-interrupt: \neg\text{interrupt} \ (Suc n)
\item \textbf{assume} thread-empty-s: thread-empty(\text{execs} \ (\text{current} \ s))
\item \textbf{assume} IH: \ ((\lambda t \text{ execs2}. \ mcurrents \ (\text{Some} \ s) \ t \rightarrow mcurrents \ (\text{run} n \ (\text{Some} s) \ execs) \ (\text{run} n \ t \ \text{execs2})))
\item \textbf{fix} t'
\item \textbf{assume} t: t = (Some t')
\item \textbf{assume} curr: mcurrents (Some s) t
\item \textbf{from} \ t \text{ curr} \text{ thread-empty(\text{execs2} \ (\text{current} \ t'))}
\item \textbf{have} mcurrents \ (\text{run} \ (Suc n) \ (\text{Some} s) \ execs) \ (\text{run} \ (Suc n) \ t \ \text{execs2})
\item \textbf{by} \textbf{auto}
\item \textbf{moreover}
\item \textbf{assume} not-prec-t: \neg\text{thread-empty(\text{execs2} \ (\text{current} \ t'))} \land \neg\text{precondition} \ (\text{next-state} \ t' \ \text{execs2}) \ (\text{next-action} \ t' \ \text{execs2})
\item \textbf{from} \ t \text{ this \ not-interrupt}
\item \textbf{have} mcurrents \ (\text{run} \ (Suc n) \ (\text{Some} s) \ execs) \ (\text{run} \ (Suc n) \ t \ \text{execs2})
\item \textbf{unfolding} mcurrents-def \textbf{by} \textbf{(simp add: rewrite-B2-cases)
execs2

D31.1 – Formal Specification of a Generic Separation Kernel

...
{  
  \textbf{assume} \quad \text{step-t: } \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)
  
  \textbf{have} \quad \text{mcurrents} (\text{Some} (\text{step} (\text{next-state } s \text{ execs}) (\text{next-action } s \text{ execs}))) (\text{Some} (\text{step} (\text{next-state } t' \text{ execs}_2) (\text{next-action } t' \text{ execs}_2)))

  \textbf{using} \quad \text{step-atomicity curr } t \text{ current-next-state unfolding} \text{ mcurrents-def unfolding step-def}

  \textbf{by} \quad \text{(cases next-action } s \text{ execs, simp, cases next-action } t' \text{ execs}_2, simp, simp, cases next-action } t' \text{ execs}_2, simp, simp)

  \textbf{from} \quad \text{current-next-state } t \text{ step-t curr not-interrupt thread-not-empty-s prec-s this IH [where } \text{execs}_2.0=\text{next-exec s execs} \text{ and } \text{t=Some} (\text{step} (\text{next-state } t' \text{ execs}_2) (\text{next-action } t' \text{ execs}_2))

  \textbf{have} \quad \text{mcurrents} (\text{run } (\text{Suc } n) (\text{Some } s \text{ execs}) (\text{run } (\text{Suc } n) t \text{ execs}_2) \text{ by auto})

  \textbf{ultimately have} \quad \text{mcurrents} (\text{run } (\text{Suc } n) (\text{Some } s \text{ execs}) (\text{run } (\text{Suc } n) t \text{ execs}_2) \text{ by blast})

  \textbf{thus} \quad \text{?thesis using current-s-t by auto}
  
  \textbf{qed}

\textbf{theorem unwinding-implies-NI-indirect-sources:}

\textbf{shows} \quad \text{NI-indirect-sources}

\textbf{proof-}

\textbf{fix} \quad \text{execs } a \text{ n}

\textbf{from} \quad \text{iunwinding-implies-view-partitioned1}

\textbf{have} \quad \text{vp: iview-partitioned by blast}

\textbf{from} \quad \text{vp and vpeq-reflexive}

\textbf{have} \quad \text{l: } \forall \text{ u. } \text{run } n (\text{Some } s0) (\text{ipurge-l execs } u) \parallel \text{run } n (\text{Some } s0) (\text{ipurge-r execs } u) \rightarrow (\lambda rs rt. \text{vpeq } u \text{ rs rt } \land \text{current } rs = \text{current } rt)

\textbf{unfolding} \quad \text{iview-partitioned-def by auto}

\textbf{have} \quad \text{run } n (\text{Some } s0) \text{ execs } \rightarrow (\lambda s-f. \text{run } n (\text{Some } s0) (\text{ipurge-l execs } (\text{current } s-f)) \parallel \text{run } n (\text{Some } s0) (\text{ipurge-r execs } (\text{current } s-f)) \rightarrow (\lambda s-l s-r. \text{output-f } s-l a = \text{output-f } s-r a))

\textbf{proof(cases run } n (\text{Some } s0) \text{ execs)}

\textbf{case None}

\textbf{thus} \quad \text{?thesis unfolding B-def by simp}

\textbf{next}

\textbf{case } (\text{Some } s-f)

\textbf{thus} \quad \text{?thesis}

\textbf{proof(cases run } n (\text{Some } s0) (\text{ipurge-l execs } (\text{current } s-f))\})

\textbf{case None}

\textbf{from Some this show} \quad \text{?thesis unfolding B-def by simp}

\textbf{next}

\textbf{case } (\text{Some } s-\text{ipurge-l})

\textbf{show} \quad \text{?thesis}

\textbf{proof(cases run } n (\text{Some } s0) (\text{ipurge-r execs } (\text{current } s-f))\})

\textbf{case None}

\textbf{from} \quad \langle \text{run } n (\text{Some } s0) \text{ execs } = \text{Some } s-f \rangle \text{ Some this show} \quad \text{?thesis unfolding B-def by simp}

\textbf{next}

\textbf{case } (\text{Some } s-\text{ipurge-r})

\textbf{from cswitch-independent-of-state}

\textbf{\langle run } n (\text{Some } s0) \text{ execs } = \text{Some } s-f \rangle \quad \langle \text{run } n (\text{Some } s0) (\text{ipurge-l execs } (\text{current } s-f)) = \text{Some } s-\text{ipurge-l} \rangle

\textbf{current-independent-of-domain-actions[where } n=n \text{ and s=Some } s0 \text{ and } t=\text{Some } s0 \text{ and execs=execs and } ?\text{execs2.0=(ipurge-l execs } (\text{current } s-f))\]
have 2: current s-ipurge-l = current s-f

unfolding mcurrents-def B-def by auto
from \langle run n (Some s0) \rangle execs = Some s-f \triangleright run n (Some s0) (ipurge-l execs (current s-f)) = Some s-ipurge-l, Some s \triangleright THEN spec where x = current s-f \rangle
have vpeq (current s-f) s-ipurge-l s-ipurge-r \land current s-ipurge-l = current s-ipurge-r

unfolding B-def by auto
from this 2 have output-f s-ipurge-l a = output-f s-ipurge-r a
using output-consistent by auto
from \langle run n (Some s0) \rangle execs = Some s-f \triangleright 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

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 AS_set.

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

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

As for the state update, this is still completely unconstrained and generic as long as it respects the generic invariant and the precondition.

**locale** Interruptible-Separation-Kernel = Separation-Kernel kstep output-f $s0$ current cswitch interrupt kprecondition realistic-execution control kinvolved ifp vpeq

**for** kstep :: 'state-t => 'action-t => 'state-t
**and** output-f :: 'state-t => 'action-t => 'output-t
**and** $s0$ :: 'state-t
**and** current :: '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 if an interrupt occurs in the given state at the given time
**and** kprecondition :: 'state-t => 'action-t => bool — Returns t if an precondition holds that relates the current action to the state
**and** realistic-execution :: 'action-t execution => bool — In this locale, this function is completely unconstrained.

**control** :: 'state-t => 'dom-t => 'action-t execution => ('(action-t option) x 'action-t execution x 'state-t)
**and** kinvolved :: 'action-t => 'dom-t set
**and** ifp :: 'dom-t => 'dom-t => bool
**and** vpeq :: 'dom-t => 'state-t => 'state-t => bool

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

**assumes** empty-in-AS-set: [] ∈ AS-set
**and** invariant-s0 invariant $s0$
**and** invariant-after-cswitch ∀ s n . invariant s → invariant (cswitch n s)
**and** precondition-after-cswitch ∀ s d n a . AS-precondition s d a → AS-precondition (cswitch n s) d a
**and** AS-precondition-first-action: ∀ s d aseq . invariant s ∧ aseq ∈ AS-set ∧ aseq ≠ [] → AS-precondition s d (hd aseq)
**and** AS-precondition-after-step: ∀ s a $\alpha$ . (∃ aseq ∈ AS-set . is-sub-seq a $\alpha$ aseq) ∧ invariant s ∧ AS-precondition s (current s) a ∧ (is-aborting s (current s) a ∧ ~ waiting s (current s) a → AS-precondition (kstep s d a) (current s) $\alpha$
**and** AS-precondition-independent: ∀ s d a $\alpha$ . current s ≠ d ∧ AS-precondition s d a → AS-precondition (kstep s d a) $\alpha$
**and** spec-of-invariant: ∀ s a . invariant s → invariant (kstep s a)

**and** kprecondition-def: kprecondition s a ≡ invariant s ∧ AS-precondition s (current s) a
**and** realistic-execution-def: realistic-execution aseq ≡ set aseq ⊆ AS-set
**and** control-spec ∀ s d aseqs . case control s d aseqs of (a, aseqs', s') ⇒ (thread-empty aseqs ∧ (a, aseqs') = (None,[])) ∀ — Nothing happens
\[(\text{aseqs} \neq [] \land \text{hd aseqs} \neq (\&\text{ waiting } s')) \land \neg \text{ aborting } s' d (\text{the a}) \land \neg \text{ waiting } s' d (\text{the a}) \land (a.aseqs') = (\text{Some (hd (hd aseqs)), (tl (hd aseqs))} = (\text{tl (tl aseqs)})) \lor - \text{ Execute the first action of the current action sequence} \]

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

\[(a.aseqs') = (\text{None tl aseqs}) \]

\begin{itemize}
\item \text{next-action-after-cswitch:} \forall s d a seqs . \text{fst (cswitch n s d aseqs) = fst (control s d aseqs)}
\item \text{next-action-after-next-state:} \forall s execs d . \text{current s \neq d \rightarrow \text{fst (control (next-state s execs) d (execs d)) = None} \lor \text{fst (control (next-state s execs) d (execs d)) = fst (control s d (execs d))}}
\item \text{next-action-after-step:} \forall s a d aseqs . \text{current s \neq d \rightarrow \text{fst (control (step s a) d aseqs) = fst (control s d aseqs)}}
\item \text{next-state-precondition:} \forall s d a execs . \text{AS-precondition s d a \rightarrow AS-precondition (next-state s execs) d a}
\item \text{and next-state-invariant:} \forall s execs . \text{invariant s \rightarrow invariant (next-state s execs)}
\item \text{and spec-of-waiting:} \forall s a . \text{waiting s (current s) a \rightarrow kstep s a = s}
\end{itemize}

\begin{verbatim}
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.

\textbf{function run-total :: time-t} \Rightarrow ' \text{state-t} \Rightarrow \text{('dom-t \Rightarrow ' action-t execution') \Rightarrow 'state-t}\\
\textbf{where run-total 0 s execs = s}\\
| \text{interrupt (Suc n) \Rightarrow run-total (Suc n) s execs = run-total n (cswitch (Suc n) s) execs}\\
| \text{\neg interrupt (Suc n) \Rightarrow thread-empty (execs (current s)) \Rightarrow run-total (Suc n) s execs = run-total n s execs}\\
| \text{\neg interrupt (Suc n) \Rightarrow thread-empty (execs (current s)) \Rightarrow run-total (Suc n) s execs = run-total n s execs (step (next-state s execs) (next-action s execs)) (next-actions s execs)}\\
\textbf{using not0-implies-Suc by (meis prod-cases3,auto)}\\
\textbf{termination by lexicographic-order}
\end{verbatim}

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

\textbf{definition realistic-executions :: ('dom-t \Rightarrow ' action-t execution) \Rightarrow bool}\\
\textbf{where realistic-executions \equiv \forall d . \text{realistic-execution (execs d)}}

Lemma \text{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 \text{AS_set}). The first action sequence, however, is being executed and is therefore not necessarily an action sequence from \text{AS_set}, but it is the last part of some action sequence from \text{AS_set}.

\textbf{definition realistic-AS-partial :: 'action-t list \Rightarrow bool}\\
\textbf{where realistic-AS-partial \equiv \exists \ n aseq'. \ n \leq \text{length aseq'} \land aseq' \in \text{AS-set} \land \text{alseq} = \text{lastn n aseq}'}

\textbf{definition realistic-executions-ind :: ('dom-t \Rightarrow ' action-t execution) \Rightarrow bool}\\
\textbf{where realistic-executions-ind execs \equiv \forall d . (case execs d of [] \Rightarrow True | (aseq\#aseq) \Rightarrow realistic-AS-partial aseq \land \text{set aseq} \subseteq \text{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.

\textbf{definition precondition-ind :: 'state-t \Rightarrow ('dom-t \Rightarrow ' action-t execution) \Rightarrow bool}\\
\textbf{where precondition-ind s execs \equiv invariant s \land (\forall d . \text{fst (control s d (execs d)) \rightarrow 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.

\textbf{lemma next-execution-is-realistic-partial:}\\
\textbf{assumes na-def: next-execs s execs d = aseq \# aseqs}
and d-is-curr: \( d = \text{current } s \)
and realistic: realistic-executions-ind \( \text{execs} \)
and thread-not-empty: \( \neg \text{thread-empty}(\text{execs (current } s)) \)

shows realistic-AS-partial \( \text{aseq} \land \text{set aseqs} \subseteq \text{AS-set} \)

proof–
let \( ?c = \text{control } s (\text{current } s) \) (execs (current } s))

\{  
assume c-empty: let \( a, \text{aseqs}'(s') = ?c \) in  
\( (a, \text{aseqs}') = (\text{None}, []) \)  
from na-def d-is-curr c-empty  
have ?thesis  
unfolding realistic-executions-ind-def next-exec-def by (auto) 
\}

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

moreover
\{
let \( ?\text{ct}= \text{execs (current } s) \)
let \( ?\text{execs}' = ?\text{ct} \)
let \( ?\text{a}' = \text{Some } (\text{hd } ?\text{ct}) \)
assume c-waiting: let \( (a, \text{aseqs}', s') = ?c \) in  
\( (a, \text{aseqs}') = (?\text{a}', ?\text{execs}') \)  
from na-def c-waiting d-is-curr  
have as-defs: \( \text{aseq} = \text{hd } ?\text{execs}' \land \text{aseq} = \text{tl } ?\text{execs}' \)  
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=\text{execs}']  
have subset: \( \text{set } (\text{tl } ?\text{execs}') \subseteq \text{AS-set} \)  
unfolding Let-def realistic-AS-partial-def  
by (cases \text{execs } d.auto)  
from na-def c-waiting d-is-curr
have \( \text{?execs}' \neq [] \) unfolding next-execs-def by auto
from realistic[unfolded realistic-executions-ind-def,THEN spec,where \( x=d \)] d-is-curr thread-not-empty
obtain \( n \) aseq' where witness: \( n \leq \text{length aseq}' \land \text{aseq}' \in \text{AS-set} \land \text{hd(execs d)} = \text{lastn n aseq}'
unfolding realistic-AS-partial-def by (cases execs d,auto)
from d-is-curr this subset as-defs have \( \text{?thesis} \)
unfolding realistic-AS-partial-def by (cases execs d,auto)

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

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

lemma run-total-equals-run.
assumes realistic-exec: realistic-executions execs
and invariant: invariant s
shows strict-equal (run n (Some s) execs) (run-total n s execs)
proof--
{
  fix \( n \) ms s execs
  have strict-equal ms s \land realistic-executions-ind execs \land precondition-ind s execs \implies \text{strict-equal (run n ms execs) (run-total n s execs)}
    proof (induct n ms execs arbitrary: s rule: run.induct)
    case (1 s execs sa)
    show \( \text{?case by auto} \)
    next
    case (2 n execs s)
show ?case unfolding strict-equal-def by auto
next
case (3 n s execs sa)
  assume interrupt: interrupt (Suc n)
  assume IH: (∀sa. strict-equal (Some (cswitch (Suc n) s)) sa ∧ realistic-executions-ind execs ∧ precondition-ind sa execs
→
  strict-equal (run n (Some (cswitch (Suc n) s)) execs) (run-total n sa execs))
  { assume equal-s-sa: strict-equal (Some s) sa
    assume realistic: realistic-executions-ind execs
    assume inv-sa: precondition-ind sa execs
    have inv-nsa: precondition-ind (cswitch (Suc n) sa) execs
    proof-
    { fix d
      have fst (control (cswitch (Suc n) sa) d (execs d)) → AS-precondition (cswitch (Suc n) sa) d
        using next-action-after-cswitch inv-sa unfolded precondition-ind-def ,THEN conjunct2,THEN spec,where
        x=d]
        precondition-after-cswitch
        unfolding Let-def B-def precondition-ind-def
        by(cases fst (control (cswitch (Suc n) sa) d (execs d)),auto)
    } thus ?thesis using inv-sa invariant-after-cswitch unfolding precondition-ind-def by auto
  qed
  from equal-s-sa realistic inv-nsa 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 ∧ precondition-ind sa execs
→
  strict-equal (run n (Some (Suc n) execs) (run-total n sa execs))
  have current-s-sa: strict-equal (Some s) sa
→ current s = current sa unfolding strict-equal-def by auto
  { assume equal-s-sa: strict-equal (Some s) sa
    assume realistic: realistic-executions-ind execs
    assume inv-sa: precondition-ind sa execs
    from equal-s-sa realistic inv-sa IH[where sa=sa]
    have equal-ns-nt: strict-equal (run n (Some s) execs) (run-total n sa execs)
    unfolding strict-equal-def by(auto)
  }
  from this current-s-sa thread-empty not-interrupt show ?case by auto
next
case (5 n execs s sa)
  assume not-interrupt: ¬interrupt (Suc n)
  assume thread-not-empty: ¬thread-empty(execs (current s))
  assume not-prec: ¬precondition (next-state s execs) (next-action s execs)
  — In locale ISK, the precondition can be proven to hold at all times. This case cannot happen, and we can prove False.
  { assume equal-s-sa: strict-equal (Some s) sa
    assume realistic: realistic-executions-ind execs
    assume inv-sa: precondition-ind sa execs

from equal-s-sa have s-sa: s = sa unfolding strict-equal-def by auto
from inv-sa have
  next-action sa execs → AS-precondition sa (current sa)
  unfolding precondition-ind-def B-def next-action-def
  by (cases next-action sa execs auto)

from this next-state-precondition
  have next-action sa execs → AS-precondition (next-state sa execs) (current sa)
  unfolding precondition-ind-def B-def
  by (cases next-action sa execs auto)

from inv-sa this s-sa next-state-invariant current-next-state
  have next-action sa execs ↭ AS-precondition (next-state sa execs) (current sa)
  unfolding precondition-ind-def B-def
  by (cases next-action sa execs auto)

from this not-prec have False by auto

thus ?case by auto

next case (6 n execs s sa)
  assume not-interrupt: ¬ interrupt (Suc n)
  assume thread-not-empty: ¬ thread-empty (execs (current s))
  assume prec: precondition (next-state s execs) (next-action s execs)
  assume IH: (\sa. strict-equal (Some (step (next-state s execs) (next-action s execs))) sa ∧
  realistic-executions-ind (next-execs s execs) (next-execs s execs)) (run-total n sa (next-execs s execs))
  have current-s-sa: strict-equal (Some s) sa ↭ current s = current sa unfolding strict-equal-def by auto

  assume equal-s-sa: strict-equal (Some s) sa
  assume realistic: realistic-executions-ind execs
  assume inv-sa: precondition-ind sa execs

from equal-s-sa have s-sa: s = sa unfolding strict-equal-def by auto

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

have equal-ns-nsa: strict-equal (Some ?ns) ?ns unfolding strict-equal-def by auto

from inv-sa equal-s-sa have inv-s: invariant s unfolding strict-equal-def precondition-ind-def by auto

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

have realistic-na: realistic-executions-ind ?na
proof-
  { fix d
    have case ?na d of [] ⇒ True | aseq # aseqs ⇒ realistic-AS-partial aseq ∧ set aseqs ⊆ AS-set
    proof(cases ?na d simp, rename-lac aseq aseqs simp, cases d = current s)
    case False
    fix aseq aseqs
    assume next-execs s execs d = aseq # aseqs
    from False this realistic unfolded realistic-executions-ind-def, THEN spec, where x=d
    show realistic-AS-partial aseq ∧ set aseqs ⊆ AS-set
    unfolding next-execs-def by simp
  next
case True
  fix aseq aseqs
  assume na-def: next-execs s execs d = aseq # aseqs
  from next-execution-is-realistic-partial na-def True realistic thread-not-empty
  show realistic-AS-partial aseq ∧ set aseqs ⊆ AS-set by blast
  qed

thus ?thesis unfolding realistic-executions-ind-def by auto
qed

have invariant-na: precondition-ind ?ns ?na
proof-
  from spec-of-invariant inv-sa next-state-invariant s-sa 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,THEN spec,where x2=sa and x1=da and x=execs d]
        have a-def: a = hd (execs (current s)) ∧ ?na d = (tl (hd (execs (current s)))) # (tl (execs (current s)))
        unfolding next-action-def next-execs-def Let-def
        by (auto)
        from a-def True snd-action-not-none control-spec[THEN spec,THEN spec,THEN spec,where x2=?ns and x1=da and x=?na d]
        second-elt-is-hd-tl[where x=hd (execs (current s)) and a=hd(tl(hd (execs (current s)))) and x'=tl (tl(hd (execs (current s))))]
        have na-def: the ?a' = (hd (execs (current s)))!1
        unfolding next-execs-def
        by (auto)
        from Some realistic[unfolded realistic-executions-ind-def,THEN spec,where x=d] True thread-not-empty
        obtain n aseq' where witness: n ≤ length aseq' ∧ aseq' ∈ AS-set ∧ hd(execs d) = lastn n aseq'
        unfolding realistic-AS-partial-def by (cases execs d auto)
        from True executing length-lt-2-implies-tp-empty[where x=hd (execs (current s))]
        Some control-spec[THEN spec,THEN spec,THEN spec,where x2=sa and x1=da and x=execs d]
        snd-action-not-none control-spec[THEN spec,THEN spec,THEN spec,where x2=?ns and x1=da and x=?na d]
        have in-action-sequence: length (hd (execs (current s))) ≥ 2
        unfolding next-action-def next-execs-def
    }
by auto
from this witness consecutive-is-sub-seq (where \( a = a \) and \( b = \text{the } ?a' \) and \( n = n \) and \( y = \text{aseq}' \) and \( x = tl (tl \text{ (execs (current s)))})
\( a \)-def \( na \)-def True in-action-sequence
\( x \)-is-hd-snd-tl (where \( x = \text{hd (execs (current s))} \))
have 1: \( \exists \text{ aseq}' \in \text{AS-set} \cdot \text{is-sub-seq a (the } ?a' \text{) 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: \( \text{AS-precondition s (current s) a} \)
unfolding strict-equal-def next-action-def B-def by auto
from executing True Some control-spec,[THEN spec],[THEN spec],[THEN spec,where \( x = \text{execs d} \) x=execs d]
have not-aborting: \( \neg \text{aborting (next-state s execs) (current s) (the } ?a \text{)} \)
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 \( x = \text{execs d} \) x=execs d]
have not-waiting: \( \neg \text{waiting (next-state s execs) (current s) (the } ?a \text{)} \)
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],[THEN spec,where \( x = \text{execs d} \) x=execs d]
\text{AS-prec-after-step,[THEN spec],[THEN spec],[THEN spec,where \( x = \text{next-state s execs} \text{ and } x = a \text{ and } x = \text{the } ?a' \) x=the ?a']
next-state-precondition not-aborting not-waiting
show \( \text{thesis} \)
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 \( x = \text{execs d} \) x=execs d]
have a-def: \( ?na = \text{execs (current s)} \wedge \text{next-state s execs} = s \wedge \text{waiting s d} \) (the \( ?a \) d)
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 \( x = \text{execs d} \) x=execs d]
have na-def: \( \text{the } ?a' = \text{hd (execs (current s))} \)
unfolding next-action-def next-execs-def
by (auto)
from spec-of-waiting a-def True
have no-step: step s ?a = s unfolding step-def by (cases next-action s execs auto)
from no-step Some True a-def
\( \text{inv-sa [unfolded precondition-ind-def,[THEN conjunct2],[THEN spec,where } x = \text{current s} \) s-sa
have 2: \( \text{AS-precondition s (current s) (the } ?a' \text{)} \)
unfolding next-action-def B-def
by (auto)
from a-def na-def this True Some no-step
show \( \text{thesis} \)
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 \( \text{snd-action-not-none} \)), 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 \( \text{AS-set} \).

```plaintext
D31.1 – Formal Specification of a Generic Separation Kernel

from None True \( \text{snd-action-not-none} \) 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 \( x=\text{execs} \ d \)]
\[ \text{have na-def: the } ?a' = \text{hd} (\text{tl} (\text{execs} (\text{current} \ s))) \land ?na = \text{tl} (\text{execs} (\text{current} \ s)) \]
\begin{align*}
\text{unfolding next-action-def next-execs-def by(auto)}
\text{from True None } \text{snd-action-not-none control-spec[THEN spec,THEN spec,THEN spec,where \( x2=?ns \) and \( x1=d \) and \( x=?na d \)]}
\text{this}
\begin{align*}
\text{have I: } \text{tl} (\text{execs} (\text{current} \ s)) \neq [] \land \text{hd} (\text{tl} (\text{execs} (\text{current} \ s))) \neq []
\end{align*}
\text{by(auto)}
\text{from this realistic unfolded realistic-executions-ind-def,THEN spec,where \( x=d \) } \text{True thread-not-empty}
\begin{align*}
\text{have } \text{hd} (\text{tl} (\text{execs} (\text{current} \ s))) & \in \text{AS-set} \\
\text{by (cases execs d,auto)}
\text{from True } \text{snd-action-not-none this}
\begin{align*}
\text{inv-ns this na-def 1} \\
\text{AS-prec-first-action[THEN spec,THEN spec,THEN spec,where \( x2=?ns \) and \( x1=\text{current} \ s \) and \( x=?na \) (current s)]}
\text{and \( x1=d \)}
\end{align*}
\begin{align*}
\text{show \( ?\text{thesis} \) by(auto)}
\text{qed}
\end{align*}
\text{thus \( ?\text{thesis} \)
\begin{align*}
\text{using control-spec[THEN spec,THEN spec,THEN spec,where \( x2=?ns \) and \( x1=\text{current} \ s \) and \( x=?na \) (current s)]}
\text{thread-not-empty True } \text{snd-action-not-none}
\end{align*}
\text{by (auto simp add: Let-def)}
\text{next}
\text{case False}
\begin{align*}
\text{from False have equal-na-a } \text{?na d = execs d}
\text{unfolding next-execs-def by(auto)}
\text{from this False current-next-state next-action-after-step}
\begin{align*}
\text{have } ?a' = \text{fst} (\text{control} (\text{next-state} s \text{ execs} \ d) (\text{next-execs} s \text{ execs} \ d))
\text{unfolding next-action-def by(auto)}
\text{from inv-sa[unfolded precondition-ind-def,THEN conjunct2,THEN spec,where \( x=d \)] s-sa equal-na-a this}
\text{next-action-after-next-state[THEN spec,THEN spec,THEN spec,where \( x=d \) and \( x2=s \) and \( x1=\text{execs} \)]}
\text{snd-action-not-none False}
\text{have AS-precondition \( s \) \( \text{d (the } ?a' \)\)}
\text{unfolding precondition-ind-def next-action-def B-def by (cases fst (control sa d (execs d)),auto)}
\text{from equal-na-a False this next-state-precondition current-next-state}
\text{AS-prec-dom-independent[THEN spec,THEN spec,THEN spec,THEN spec,where \( x3=\text{next-state} s \text{ execs} \) and \( x2=d \) and \( x=\text{the } ?a \) and \( x1=\text{the } ?a' \)]}
\text{show \( ?\text{thesis} \) and \( x3=\text{next-state} s \text{ execs} \)}
\begin{align*}
\text{show \( ?\text{thesis} \) and \( x3=\text{next-state} s \text{ execs} \)}
\text{by (cases next-action s execs,auto)}
\text{qed}
\end{align*}
\text{hence } \text{fst (control } ?\text{ns d (} ?\text{na d)}) \rightarrow \text{AS-precondition } ?\text{ns d unfolding B-def}
\text{by (cases fst (control } ?\text{ns d (} ?\text{na d) ,auto)}
\end{align*}
\text{thus \( ?\text{thesis} \) by(auto)}
\text{qed}
\text{from this inv-ns show \( ?\text{thesis} \) unfolding precondition-ind-def B-def Let-def}
Theorem unwinding_implies_isecure gives security for all realistic executions. For unrealistic exe-


cutions, 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** \( \forall \text{execs } d . \text{realistic-executions execs } \rightarrow \text{realistic-executions (purge execs } d) \)

**proof**

\[
\begin{align*}
\{ \\
\text{fix execs } d \\
\text{assume realistic-executions execs} \\
\text{hence realistic-executions (purge execs } d) \\
\quad \text{using some}\{\text{where } P = \text{realistic-execution and } x = \text{execs } d\} \\
\text{unfolding realistic-executions-def purge-def by (simp)}
\}
\]

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

**qed**

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

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

**by (induct exec, auto)**

**lemma** realistic-ipurge-l:

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

**proof**

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

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

**qed**

**lemma** realistic-ipurge-r:

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

**proof**

\[
\begin{align*}
\{ \\
\text{fix execs } d \\
\text{assume 1: realistic-executions execs} \\
\quad \text{from empty-in-AS-set remove-gateway-comm-subset}\{\text{where } d = d \text{ and exec = execs } d\} \text{ I have realistic-executions (ipurge-r execs } d) \\
\quad \text{using some}\{\text{where } P = \lambda x . \text{realistic-execution } x \text{ and } x = \text{execs } d\} \\
\quad \text{unfolding realistic-execution-def realistic-executions-def ipurge-r-def by (auto)}
\}
\]

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

**qed**

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

**definition** NI-unrelated-total: bool

**where** NI-unrelated-total

\( \equiv \forall \text{execs } a \text{ n } . \text{realistic-executions execs } \rightarrow \)

\[(\text{let } s-f = \text{run-total } n \text{ s0 execs in} \\
\text{output-f s-f a } = \text{output-f (run-total } n \text{ s0 (purge execs (current s-f))) a} \\
\land \text{current s-f } = \text{current (run-total } n \text{ s0 (purge execs (current s-f)))}) \]

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

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

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

have NI-unrelated-total: NI-unrelated-total
proof−
  { fix execs a n
    assume realistic: realistic-executions execs
    from invariant-s0 realistic run-total-equals-run[where n=n and s=s0 and execs=execs]
    have 1: strict-equal (run n (Some s0) execs) (run-total n s0 execs) by auto
    have let s-f = run-total n s0 execs in output-f s-f a = output-f (run-total n s0 (purge execs (current s-f))) a ∧
      current s-f = current (run-total n s0 (purge execs (current s-f)))
    proof (cases run n (Some s0) execs)
      case None
        thus ?thesis 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 unfolding NI-unrelated-total-def strict-equal-def by auto
        from 2 None show ?thesis unfolding NI-unrelated-total-def strict-equal-def by auto
      next
      case (Some s-f2)
        from \( \text{run n (Some s0) execs = Some s-f} \) 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 NI-indirect-sources-total: 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 \(s-f = \text{run-total} \ n \ s0 \ \text{execs} \ \text{in output-f} \ (\text{run-total} \ n \ s0 \ \text{execs} (\text{current} \ s-f)) \ a = \text{output-f} \ (\text{run-total} \ n \ s0 \ (\text{ipurge-r} \ \text{execs} (\text{current} \ s-f)))\) a

proof (cases run \ n \ (\text{Some} \ s0) \ \text{execs})

\begin{enumerate}
\item case None
  \begin{enumerate}
  \item thus \(\text{thesis using 1 unfolding NI-unrelated-total-def strict-equal-def by auto}\)
  \end{enumerate}
\item next
  \begin{enumerate}
  \item case (Some \ s-f)
    \begin{enumerate}
    \item from realistic-ipurge-l invariant-s0 realistic run-total-equals-run [where \(n=n\) and \(s=s0\) and \text{execs} = \text{ipurge-l execs (current s-f)}]
    \begin{enumerate}
    \item have 2: strict-equal (run \ n \ (\text{Some} \ s0) \ (\text{ipurge-l execs (current s-f)})) (run-total \ n \ s0 \ (\text{ipurge-l execs (current s-f)})) by auto
    \end{enumerate}
    \item from realistic-ipurge-r invariant-s0 realistic run-total-equals-run [where \(n=n\) and \(s=s0\) and \text{execs} = \text{ipurge-r execs (current s-f)}]
    \begin{enumerate}
    \item have 3: strict-equal (run \ n \ (\text{Some} \ s0) \ (\text{ipurge-r execs (current s-f)})) (run-total \ n \ s0 \ (\text{ipurge-r execs (current s-f)})) by auto
    \end{enumerate}
    \end{enumerate}
  \end{enumerate}
\end{enumerate}

show ?thesis proof(cases run \ n \ (\text{Some} \ s0) \ (\text{ipurge-l execs (current s-f)}))

\begin{enumerate}
\item case None
  \begin{enumerate}
  \item from 2 None show ?thesis using 2 unfolding NI-unrelated-total-def strict-equal-def by auto
  \end{enumerate}
\item next
  \begin{enumerate}
  \item case (Some \ s-ipurge-l)
    \begin{enumerate}
    \item show ?thesis
    \begin{enumerate}
    \item proof(cases run \ n \ (\text{Some} \ s0) \ (\text{ipurge-r execs (current s-f)}))
    \end{enumerate}
    \end{enumerate}
  \end{enumerate}
\end{enumerate}

\begin{enumerate}
\item case (Some \ s-ipurge-r)
  \begin{enumerate}
  \item from \(\text{run} \ n \ (\text{Some} \ s0) \ \text{execs} = \text{Some} \ s-f \cdot \text{run} \ n \ (\text{Some} \ s0) \ (\text{ipurge-l execs (current s-f)}) = \text{Some} \ s-ipurge-l\)
    \begin{enumerate}
    \item \(\text{x=n and x2=execs}\)
      \begin{enumerate}
      \item show ?thesis
      \begin{enumerate}
      \item unfolding strict-equal-def NI-unrelated-def
      \begin{enumerate}
      \item by(simp add: Let-def B-def B2-def)
      \end{enumerate}
      \end{enumerate}
      \end{enumerate}
    \end{enumerate}
  \end{enumerate}
\end{enumerate}

\begin{enumerate}
\item thus ?thesis unfolding NI-indirect-sources-total-def by auto
  \begin{enumerate}
  \item qed
  \end{enumerate}
\end{enumerate}

\begin{enumerate}
\item end
  \begin{enumerate}
  \item end
  \end{enumerate}
\end{enumerate}

\begin{enumerate}
\item end
  \begin{enumerate}
  \item end
  \end{enumerate}
\end{enumerate}

\begin{enumerate}
\item end
  \begin{enumerate}
  \item end
  \end{enumerate}
\end{enumerate}

\begin{enumerate}
\item end
  \begin{enumerate}
  \item end
  \end{enumerate}
\end{enumerate}

3.4 CISK (Controlled Interruptible Separation Kernel)

theory CISK
import ISK
begin

This section presents a generic model of a Controlled Interruptible Separation Kernel (CISK). It formulates security, i.e., intransitive noninterference. For a presentation of this model, see Section 2 of [31].
First, a locale is defined that defines all generic functions and all proof obligations (see Section 2.3 of [31]).

**locale** Controllable-Interruptible-Separation-Kernel = ⋯ CISK

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**assumes** vpeq-transitive: ∀ a b c u. (vpeq u a b ∧ vpeq u b c) ⇒ vpeq u a c

**and** vpeq-symmetric: ∀ a b u. vpeq u a b ⇒ vpeq u b a

**and** vpeq-reflexive: ∀ a u. vpeq a a

**and** ifp-reflexive: ∀ a 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: ∀ s a u. ¬ifp (current s) u ∧ invariant s ∧ AS-precondition s (current s) a ⇒ vpeq u s (kstep s a)

**and** output-consistent: ∀ a s t. vpeq (current s) s t ∧ current s = current t → (output-f s a) = (output-f t a)

**and** step-atomicity: ∀ a s. current (kstep s a) = current s

**and** cswitch-independent-of-state: ∀ n s t. current s = current t → (cswitch n s) = (cswitch n t)

**and** cswitch-consistency: ∀ u s t n. vpeq u s t ⇒ vpeq (cswitch n s) (cswitch n t)

**and** empty-in-AS-set: [] ∈ AS-set

**and** invariant-s0 invariant s0

**and** invariant-after-cswitch: ∀ s n a. invariant s ∧ AS-precondition s d a → AS-precondition (cswitch n s) d a

**and** precondition-after-cswitch: ∀ s n a. AS-precondition s d a → AS-precondition (cswitch n s) d a

**and** AS-prec-first-action: ∀ s d aseq. invariant s ∧ aseq ∈ AS-set ∧ aseq ≠ [] → AS-precondition s d (hd aseq)

**and** AS-prec-after-step: ∀ s a a'. (∃ aseq ∈ AS-set. is-sub-seq a a' aseq) ∧ invariant s ∧ AS-precondition s (current s) a ∧ ¬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 d a. current s ≠ d → aborting (kstep s a) d = aborting d s d

**and** aborting-consistent: ∀ s n u. vpeq u s t ⇒ aborting s u = aborting t u

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

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

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

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

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

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

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

**and** precondition-after-set-error-code: ∀ s d a a'. AS-precondition s d a ∧ aborting s (current s) a' → AS-precondition (set-error-code s a') d a
and invariant-after-set-error-code: \( \forall \ s \ a . \ \text{invariant} \ s \rightarrow \text{invariant} (\text{set-error-code} \ s \ a) \)
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

3.4.1 Execution semantics

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

function CISK-control :: \('\text{state-t} \Rightarrow \text{dom-t} \Rightarrow \text{action-t}\) execution \Rightarrow (\text{action-t option} \times \text{action-t execution} \times \text{state-t})\)
where CISK-control s d [] = (None, [], s) — The thread is empty
| CISK-control s d ([]#[]) = (None, [], s) — The current action sequence has been finished and the thread has no next action sequences to execute
| CISK-control s d ([]#(as'#execs')) = (None, as'#execs', s) — The current action sequence has been finished. Skip to the next sequence

by pat-completeness auto
termination by lexicographic-order

Function run defines the execution semantics. This function is presented in [31] by pseudo code (see Algorithm 1). Before defining the run function, we define accessor functions for the control mechanism. Functions next_action, next_execs and next_state correspond to “control.a”, “control.x” and “control.s” in [31].

abbreviation next-action::\('\text{state-t} \Rightarrow (\text{dom-t} \Rightarrow \text{action-t execution}) \Rightarrow \text{action-t option}\)\)
where next-action \equiv Kernel.next-action current CISK-control
abbreviation next-execs::\('\text{state-t} \Rightarrow (\text{dom-t} \Rightarrow \text{action-t execution}) \Rightarrow (\text{dom-t} \Rightarrow \text{action-t execution})\)\)
where next-execs \equiv Kernel.next-execs current CISK-control
abbreviation next-state::\('\text{state-t} \Rightarrow (\text{dom-t} \Rightarrow \text{action-t execution}) \Rightarrow \text{state-t}\)\)
where next-state \equiv 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::\('\text{action-t execution} \Rightarrow \text{bool}\)\)
where thread-empty exec \equiv exec = [] \lor exec = [[]]

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

function run : \text{time-t} \Rightarrow (\text{state-t} \Rightarrow (\text{dom-t} \Rightarrow \text{action-t execution}) \Rightarrow \text{state-t})\)
where run 0 s execs = s
| interrupt (Suc n) \rightarrow run (Suc n) s execs = run n (\text{cswitch} (Suc n) \text{ s execs})
| \neg interrupt (Suc n) \rightarrow thread-empty(execs (\text{current} \ s)) \rightarrow run (Suc n) s execs = run n s execs
| \neg interrupt (Suc n) \rightarrow \neg thread-empty(execs (\text{current} \ s)) \rightarrow run (Suc n) s execs = (\text{let} control-a = \text{next-action} s execs;
| \text{control-s} = \text{next-state s execs};
| \text{control-x} = \text{next-exec s execs in case control-a of \text{None} \Rightarrow run n \text{control-s control-x}};
| (\text{Some a}) \Rightarrow run n (\text{kstep control-s a control-x})\)
using not0-implies-Suc by (metis prod-cases3.auto)
termination by lexicographic-order
### 3.4.2 Formulations of security

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

**abbreviation** kprecondition

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

**definition** realistic-execution

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

**definition** realistic-executions :: (dom t ⇒ action t execution) ⇒ bool

where realistic-executions execs ≡ ∀ d. realistic-execution (execs d)

**abbreviation** involved where involved ≡ Kernel.involved

**abbreviation** step where step ≡ Kernel.step

**abbreviation** purge where purge ≡ Separation-Kernel.purge

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

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

**definition** NI-unrelated :: bool

where NI-unrelated ≡ ∀ execs a n. realistic-executions execs →

(let s-f = run n s0 execs in
  output-f s-f a = output-f (run n s0 (purge execs (current s-f))) a)

**definition** NI-indirect-sources :: bool

where NI-indirect-sources ≡ ∀ execs a n. realistic-executions execs →

(let s-f = run n s0 execs in
  output-f (run n s0 (ipurge-l execs (current s-f))) a =
  output-f (run n s0 (ipurge-r execs (current s-f))) a)

**definition** isecure :: bool

where isecure ≡ NI-unrelated ∧ NI-indirect-sources

### 3.4.3 Proofs

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

To prove this theorem, the refinement framework is used. CISK is a refinement of ISK, as the only 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-exec-consistent:
shows \( \forall s t \text{ execs} \cdot \text{vpeq}(\text{current } s) s t \land (\forall d \in \text{involved}(\text{next-action } s \text{ execs}) \cdot \text{vpeq } d \; s \; t) \land \text{current } s = \text{current } t \implies \text{fst}(\text{snd}(\text{CISK-control } s (\text{current } s)(\text{execs}(\text{current } s)))) = \text{fst}(\text{snd}(\text{CISK-control } t (\text{current } s)(\text{execs}(\text{current } s)))) \)
proof -
{  
  fix \; s \; t \; \text{execs}  
  assume \; \text{vpeq}(\text{current } s) s t  
  assume \; \text{vpeq-involved}: \forall d \in \text{involved}(\text{next-action } s \text{ execs}) \cdot \text{vpeq } d \; s \; t  
  assume \; \text{current-s-t}: \text{current } s = \text{current } t  
  from aborting-consistent \; \text{current-s-t} \; \text{vpeq}  
  have \; I: \text{aborting } t (\text{current } s) = \text{aborting } s (\text{current } s) \; \text{by auto}  
  from \; I \; \text{vpeq} \; \text{current-s-t} \; \text{vpeq-involved} \; \text{waiting-consistent}[\text{THEN spec},\text{THEN spec},\text{THEN spec},\text{THEN spec},\text{where } x_3=s \; \text{and } x_2=t \; \text{and } x_1=\text{current } s \; \text{and } x=\text{the } (\text{next-action } s \text{ execs})]  
  have \; \text{fst}(\text{snd}(\text{CISK-control } s (\text{current } s)(\text{execs}(\text{current } s)))) = \text{fst}(\text{snd}(\text{CISK-control } t (\text{current } s)(\text{execs}(\text{current } s))))  
  unfolding \; \text{Kernel.next-action-def} \; \text{Kernel.involved-def}  
  by(\text{cases } (s,\text{current } s,\text{execs}(\text{current } s)) \; \text{rule: CISK-control.cases,auto} \; \text{split: if-split-asm})  
}  
thus \; ?\text{thesis} \; \text{by auto}  
qed

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

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

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

fix \( s u \) execs  
assume \( \neg \text{ifp} \ (\text{current} \ s) \ u \)  
hence vpeq \( u \ s \) \((\text{next-state} \ s \ \text{execs})\)  
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 \( \forall \ s d \ \text{aseqs}. \)  
case CISK-control \( s d \ \text{aseqs} \) of  
\((a, \text{aseqs}' , s') \Rightarrow \)  
thread-empty \text{aseqs} \wedge (a, \text{aseqs}') = (\text{None}, [\]) \or  
\text{aseqs} \notin [\] \wedge \text{hd} \text{aseqs} \notin [\] \wedge \neg \text{aborting} \ s' d (\text{the a}) \wedge \neg \text{waiting} \ s' d (\text{the a}) \wedge (a, \text{aseqs}') = (\text{Some} \ (\text{hd} \ (\text{hd} \text{aseqs}))), \text{tl} \ (\text{hd} \text{aseqs}) \# \text{tl} \text{aseqs} \or  
\text{aseqs} \notin [\] \wedge \text{hd} \text{aseqs} \notin [\] \wedge \neg \text{aborting} \ s' d (\text{the a}) \wedge \neg \text{waiting} \ s' d (\text{the a}) \wedge (a, \text{aseqs}', s') = (\text{Some} \ (\text{hd} \ (\text{hd} \text{aseqs}))), \text{aseqs}, s \or (a, \text{aseqs}') = (\text{None}, \text{tl} \text{aseqs})  
proof  
\{  
fix \( s d \ \text{aseqs}\)  
have case CISK-control \( s d \ \text{aseqs} \) of  
\((a, \text{aseqs}', s') \Rightarrow \)  
thread-empty \text{aseqs} \wedge (a, \text{aseqs}') = (\text{None}, [\]) \or  
\text{aseqs} \notin [\] \wedge \text{hd} \text{aseqs} \notin [\] \wedge \neg \text{aborting} \ s' d (\text{the a}) \wedge \neg \text{waiting} \ s' d (\text{the a}) \wedge (a, \text{aseqs}') = (\text{Some} \ (\text{hd} \ (\text{hd} \text{aseqs}))), \text{tl} \ (\text{hd} \text{aseqs}) \# \text{tl} \text{aseqs} \or  
\text{aseqs} \notin [\] \wedge \text{hd} \text{aseqs} \notin [\] \wedge \neg \text{aborting} \ s' d (\text{the a}) \wedge \neg \text{waiting} \ s' d (\text{the a}) \wedge (a, \text{aseqs}', s') = (\text{Some} \ (\text{hd} \ (\text{hd} \text{aseqs}))), \text{aseqs}, s \or (a, \text{aseqs}') = (\text{None}, \text{tl} \text{aseqs})  
by (\(\text{cases} \ (s,d,\text{aseqs}) \) rule: CISK-control.cases,auto)  
\}  
thus \(?\)thesis by auto  
qed

lemma next-action-after-cswitch:  
shows \( \forall \ s n d \ \text{aseqs} . \) fst (CISK-control (cswitch n s) d aseqs) = fst (CISK-control s d aseqs)  
proof  
\{  
fix \( s n d \ \text{aseqs}\)  
have \(\text{fst} \ (\text{CISK-control} \ (\text{cswitch} \ n s) \ d \ \text{aseqs}) = \text{fst} \ (\text{CISK-control} \ s \ d \ \text{aseqs})\)  
using aborting-switch-independent waiting-switch-independent  
by (\(\text{cases} \ (s,d,\text{aseqs}) \) rule: CISK-control.cases,auto)  
\}  
thus \(?\)thesis by auto  
qed

lemma next-action-after-next-state:  
shows \( \forall \ s \ \text{execs} d . \) current \( s \# d \) \(\rightarrow \) \(\text{fst} \ (\text{CISK-control} \ \text{next-state} \ s \ \text{execs} \ d \ (\text{execs} \ d)) = \text{None} \or \text{fst} \ (\text{CISK-control} \ \text{next-state} \ s \ \text{execs} \ d \ (\text{execs} \ d)) = \text{fst} \ (\text{CISK-control} \ s \ d \ (\text{execs} \ d))\)  
proof  
\{  
fix \( s \ \text{execs} d \ \text{aseqs}\)  
assume \(1: \text{current} \ s \# d\)  
have \(\text{fst} \ (\text{CISK-control} \ \text{next-state} \ s \ \text{execs} \ d \ \text{aseqs}) = \text{None} \or \text{fst} \ (\text{CISK-control} \ \text{next-state} \ s \ \text{execs} \ d \ \text{aseqs}) = \text{fst} \ (\text{CISK-control} \ s \ d \ \text{aseqs})\)  
proof (\(\text{cases} \ (s,d,\text{aseqs}) \) 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-asn)
q.e.d.
}{
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
\begin{verbatim}
{ 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-asm) }

hence next-action s execs ↭ (λ a. a ∈ actions-in-execution (execs (current s))) unfolding Kernel.next-execs-def actions-in-execution-def by (cases next-action s execs, auto)

thus ⊨ thesis unfolding B-def by (auto)
qed

lemma next-execs-subset:
shows ∀ s execs u. actions-in-execution (next-execs s execs u) ⊆ actions-in-execution (execs u)
proof−
{ fix s execs u
  have actions-in-execution (next-execs s execs u) ⊆ 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-asm)
  }

thus ⊨ thesis by auto
qed

theorem unwinding-implies-isecure-CISK:
shows isecure
proof−
interpret int: Interruptible-Separation-Kernel kstep output-f s0 current cswitch interrupt kprecondition realistic-execution CISK-control kmixed involved ifp vpeq AS-set invariant AS-precondition aborting waiting

proof (unfold-locales)
  show ∀ a b c u. vpeq u a b ∧ vpeq u b c → vpeq u a c using vpeq-transitive by blast
  show ∀ a b u. vpeq u a b → vpeq u b a using vpeq-symmetric by blast
  show ∀ a u. vpeq u a a using vpeq-reflexive by blast
  show ∀ u. ifp u u using ifp-reflexive by blast
  show ∀ a 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)
    using weakly-step-consistent by blast
  show ∀ a s t u a. ¬ifp (current s) u ∧ kprecondition s a → vpeq u s (kstep s a)
    using locally-respects by blast
  show ∀ a s t. vpeq (current s) s t ∧ current s = current t → (output-f s a) = (output-f t a)
    using output-consistent by blast
  show ∀ s a . current (kstep s a) = current s
    using step-atomicity by blast
  show ∀ n s t . current s = current t → current (cswitch n s) = current (cswitch n t)
    using cswitch-independent-of-state by blast
  show ∀ u s t n . vpeq u s t → vpeq u (cswitch n s) (cswitch n t)
    using cswitch-consistency by blast
  show ∀ s t execs. vpeq (current s) s t ∧ (∀ d ∈ involved (next-action s execs) . vpeq d s t) ∧ current s = current t → next-action s execs = next-action t execs
    using next-action-consistent by blast

\end{verbatim}
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. actions-in-execution (next-execs s execs u) ⊆ actions-in-execution (execs u)
  using next-execs-subset by blast

show ∀ s d aseqs.
  case CISK-control s d aseqs of
  (a, aseqs', s') ⇒
    thread-empty aseqs ∧ (a, aseqs') = (None, []) ∨
    aseqs # [] ∧ hd aseqs # [] ∧ ¬ aborting s' d (the a) ∧ ¬ waiting s' d (the a) ∧ (a, aseqs') = (Some (hd (hd aseqs))), tl (hd aseqs) # tl aseqs) ∨
    aseqs # [] ∧ hd aseqs # [] ∧ waiting s' d (the a) ∧ (a, aseqs', s') = (Some (hd (hd aseqs)), aseqs, s) ∨ (a, aseqs') = (None, tl aseqs)
  using CISK-control-spec by blast

show ∀ s n d aseqs. fst (CISK-control (cswitch n s) d aseqs) = fst (CISK-control s d aseqs)
  using next-action-after-cswitch by auto

show ∀ s execs d.
  current s # d ---
  fst (CISK-control (next-state s execs) d (execs d)) = None v fst (CISK-control (next-state s execs) d (execs d))
  using next-action-after-next-state by auto
show \( \forall s \in D \) \( \text{current}(s) \neq d \rightarrow \text{fst}(\text{CISK-control}(\text{step}(s, a), d, a \text{seqs})) = \text{fst}(\text{CISK-control}(s, d, a \text{seqs})) \)

using next-action-after-step by auto

show \( \forall s \in D \) \( \text{execs}(s) \rightarrow \text{AS-precondition}(\text{next-state}(s, \text{execs}), d, a) \)

using next-state-precondition by auto

show \( \forall s \in D \) \( \text{invariant}(s) \rightarrow \text{invariant}(\text{next-state}(s, \text{execs})) \)

using next-state-invariant by auto

show \( \forall s \in D \) \( \text{waiting}(s) \rightarrow \text{kstep}(s, a) = s \)

using spec-of-waiting by blast

qed

note interpreted = intInterruptible-Separation-Kernel-axioms

note run-total-induct = Interruptible-Separation-Kernel.run-total.induct[of kstep output-f s0 current cs switch kprecondition realistic-execution]

aborting waiting - interrupt]

have run-equals-run-total:

\[ n \in D \rightarrow \text{run}(n, s, \text{execs}) \equiv \text{Interruptible-Separation-Kernel.run-total}(\text{kstep}(n, s, \text{execs})) \text{ interrupt CISK-control(n, s, \text{execs})} \]

proof −

fix \( n \in D \)

show \( \text{run}(n, s, \text{execs}) \equiv \text{Interruptible-Separation-Kernel.run-total}(\text{kstep}(n, s, \text{execs})) \text{ interrupt CISK-control(n, s, \text{execs})} \)

using interpreted int.step-def by (induct n, s, \text{execs} rule: run-total-induct, auto split: option.splits)

qed

from interpreted

have \( \text{NI-unrelated}(\text{kstep output-f s0 current cs switch interrupt realistic-execution CISK-control kinvolved ifp}) \)

by (metis int.unwinding-implies-secure-total)

from \( \text{NI-unrelated}(\text{kstep output-f s0 current cs switch interrupt realistic-execution CISK-control kinvolved ifp}) \)

have \( \text{NI-unrelated-def}(\text{int.realistic-executions-def int.realistic-executions-def int.NI-unrelated-total-def}) \)

by (metis realistic-executions-def int.realistic-executions-def int.realistic-executions-def int.NI-unrelated-total-def)

from \( \text{NI-unrelated-def}(\text{int.realistic-executions-def int.realistic-executions-def int.realistic-executions-def int.NI-unrelated-total-def}) \)

have \( \text{NI-indirect-sources}(\text{int.realistic-executions-def int.NI-indirect-sources-total-def int.realistic-executions-def int.NI-indirect-sources-def}) \)

by (metis realistic-executions-def int.NI-indirect-sources-total-def int.realistic-executions-def int.NI-indirect-sources-def)

from \( \text{NI-indirect-sources}(\text{int.realistic-executions-def int.NI-indirect-sources-total-def int.realistic-executions-def int.NI-indirect-sources-def}) \)

have \( \text{NI-indirect-sources-def}(\text{int.realistic-executions-def int.NI-indirect-sources-total-def int.realistic-executions-def int.NI-indirect-sources-def}) \)

by (metis realistic-executions-def int.NI-indirect-sources-total-def int.realistic-executions-def int.NI-indirect-sources-def)

from \( \text{NI-indirect-sources-def}(\text{int.realistic-executions-def int.NI-indirect-sources-total-def int.realistic-executions-def int.NI-indirect-sources-def}) \)

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

qed

end

4 Instantiation by a separation kernel with concrete actions

theory Step-configuration

imports Main

begin

In the previous section, no concrete actions for the step function were given. The foremost point we want to make by this instantiation is to show that we can instantiate the CISK model of the previous section with an implementation that, for the step function, as actions, provides events and interprocess communication (IPC) System calls invocation that can be interrupted at certain interrupt points are split into several atomic steps. A communication interface of events and IPC is less “trivial” than it may seem at first glance, for example the L4 microkernel API only provided IPC as communication primitive [16]. In particular, the concrete actions illustrate how an application of the CISK framework
can be used to separate policy enforcement from other computations unrelated to policy enforcement.

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

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

4.1 Model of a separation kernel configuration

4.1.1 Type definitions

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

typedec partition-id-t
typedec thread-id-t

typedec page-t — physical address of a memory page
typedec filep-t — name of file provider

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

datatype mode-t =
  READ — The subject has right to read from the memory page, from the files served by a file provider.
  WRITE — The subject has right to write to the memory page, from the files served by a file provider.
  PROVIDE — The subject has right serve as the file provider. This mode is not used for memory pages or ports.

4.1.2 Configuration

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

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-subject sp-spec-subj-obj and subject-object sp-spec-subj-subj access control policy for the implementation of the model. The following locale summarizes all properties we need.

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

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

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

and ifp-reflexive:
  ∀ p . ifp p p

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

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

begin end

4.2.2 Derivation

The configuration data only consists of a subject-object policy. We derive the subject-subject policy and the information flow policy from the configuration data and prove that properties we specified in Section 4.2.1 are satisfied.

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

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

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

definition ifp a b ≡
  sp-spec-subj-subj a b
\[ \forall \text{sp-spec-subj-subj } b \ a \\
\forall (\exists \ c \ y . \text{sp-spec-subj-subj } a \ c \\
\land \text{sp-spec-subj-obj } c \ (\text{PAGE } y) \ \text{WRITE} \\
\land \text{sp-spec-subj-obj } b \ (\text{PAGE } y) \ \text{READ}) \\
\forall (a = b) \]

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

**Lemma correct:**

**Shows** policy-axioms \text{sp-spec-subj-obj} \text{sp-spec-subj-subj} ifp

**Proof** (unfold-locales)

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

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

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

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

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

**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:

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

**Unfolding** ifp-def by auto

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

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

**Unfolding** ifp-def by auto

qed

end

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

**Type-synonym** \text{sp-subj-obj-t} = \text{partition-id-t} \Rightarrow \text{obj-id-t} \Rightarrow \text{mode-t} \Rightarrow \text{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 \textit{obj-t} — value of an object

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

\textbf{consts}

\begin{itemize}
  \item partition :: thread-id-t \to partition-id-t
\end{itemize}

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

\textbf{record} \textit{thread-t} =

\begin{itemize}
  \item \textit{ev-counter} :: nat — event counter
\end{itemize}

\textbf{record} \textit{state-t} =

\begin{itemize}
  \item sp-impl-subj-subj :: sp-subj-subj-t — current subject-subject policy
  \item sp-impl-subj-obj :: sp-subj-obj-t — current subject-object policy
  \item current :: thread-id-t — current thread
  \item obj :: obj-id-t \to obj-t — values of all objects
  \item thread :: thread-id-t \to thread-t — internal state of threads
\end{itemize}

Later (Section 4.4), the system invariant \textit{sp-subset} will be used to ensure that the dynamic policies (\textit{sp_impl_...}) are a subset of the corresponding static policies (\textit{sp_spec_...}).

**4.3.3 Atomic step**

\textbf{Helper functions} Set new value for an object.

\textbf{definition} \textit{set-object-value} :: obj-id-t \to obj-t \to state-t \to state-t where

\begin{itemize}
  \item \textit{set-object-value} obj-id val s = s \langle obj = \textit{fun-upd} (obj s) obj-id val \rangle
\end{itemize}

Return a representation of the opposite direction of IPC communication.

\textbf{definition} \textit{opposite-ipc-direction} :: ipc-direction-t \to ipc-direction-t where

\begin{itemize}
  \item \textit{opposite-ipc-direction} dir \equiv \text{case \textit{dir} of SEND \to RECV | RECV \to SEND}
\end{itemize}

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.

\textbf{definition} \textit{add-access-right} :: partition-id-t \to obj-id-t \to mode-t \to state-t \to state-t where

\begin{itemize}
  \item \textit{add-access-right} part-id obj-id mode s = s \langle\text{sp-impl-subj-obj} := \lambda q q' \to q'' \rangle. \ (\text{part-id} = q \land \text{obj-id} = q' \land m = q'') \lor \text{sp-impl-subj-obj} q q' q''
\end{itemize}

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

**Model of event syscalls**  

**Definition**  

\[
\text{add-comm-right} :: \text{partition-id-t} \Rightarrow \text{partition-id-t} \Rightarrow \text{state-t} \Rightarrow \text{state-t} \text{ where } \]

\[
\text{add-comm-right } p \ p' s \equiv \ s \ ( \sp-impl-subj-subj = \lambda q q'. (p \ q \land p' = q') \lor \sp-impl-subj-subj \ s \ q \ q' )
\]

**Atomic step for IPC**  

**Definition**  

\[
\text{ipc-precondition} :: \text{thread-id-t} \Rightarrow \text{ipc-direction-t} \Rightarrow \text{thread-id-t} \Rightarrow \text{page-t} \Rightarrow \text{state-t} \Rightarrow \text{bool} \text{ where } \]

\[
\text{ipc-precondition } tid \ dir \ partner \ page \ s \equiv \]

\[
\begin{align*}
& \text{let } sender = (\text{case } dir \Rightarrow \text{SEND } \Rightarrow tid \ | \ \text{RECV } \Rightarrow \text{partner}) \text{ in} \\
& \text{let } receiver = (\text{case } dir \Rightarrow \text{SEND } \Rightarrow \text{partner} \ | \ \text{RECV } \Rightarrow \text{tid}) \text{ in} \\
& \text{let } local-access-mode = (\text{case } dir \Rightarrow \text{READ } \Rightarrow \text{READ} \ | \ \text{RECV } \Rightarrow \text{WRITE}) \text{ in} \\
& \quad \sp-impl-subj-subj \ s \ (\text{partition sender}) \ (\text{partition receiver}) \\
& \quad \land \ sp-impl-subj-obj \ s \ (\text{partition tid}) \ (\text{PAGE page}) \ local-access-mode
\end{align*}
\]

**Atomic step for event calls**  

**Definition**  

\[
\text{atomic-step-ipc} :: \text{thread-id-t} \Rightarrow \text{ipc-direction-t} \Rightarrow \text{ipc-stage-t} \Rightarrow \text{thread-id-t} \Rightarrow \text{page-t} \Rightarrow \text{state-t} \Rightarrow \text{state-t} \text{ where } \]

\[
\text{atomic-step-ipc } tid \ dir \ stage \ partner \ page \ s \equiv \]

\[
\begin{align*}
& \text{case } stage \Rightarrow \\
& \quad \text{PREP } \Rightarrow s \\
& \quad \text{WAIT } \Rightarrow s \\
& \quad \text{BUF } page' \Rightarrow \\
& \quad \quad \text{(case } dir \Rightarrow \text{SEND } \Rightarrow \\
& \quad \quad \quad \quad \text{set-object-value } (\text{PAGE page'}) \ (\text{obj } s \ (\text{PAGE page})) \ s) \\
& \quad \quad \text{| RECV } \Rightarrow s)
\end{align*}
\]

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

**definition aborting :: state-t ⇒ thread-id-t ⇒ int-point-t ⇒ bool**

**where** aborting s tid a ≡ case a of SK-IPC dir PREP partner page ⇒ ~ipc-precondition tid dir partner page s
| SK-EV-SIGNAL EV-SIGNAL-PREP partner ⇒ ~ev-signal-precondition tid partner s
| - ⇒ False

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

**definition waiting :: state-t ⇒ thread-id-t ⇒ int-point-t ⇒ bool**

**where** waiting s tid a ≡ case a of SK-IPC dir WAIT partner page ⇒ ~ipc-precondition partner (opposite-ipc-direction dir) tid (SOME page'). True) s
| SK-EV-WAIT EV-PREP - ⇒ False
| SK-EV-WAIT EV-WAIT - ⇒ ev-counter (thread s tid) = 0
| SK-EV-WAIT EV-FINISH - ⇒ False
| - ⇒ False

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

**definition atomic-step :: state-t ⇒ int-point-t ⇒ state-t where**

**atomic-step s ipt ≡**

**case ipt of**

| SK-IPC dir stage partner page ⇒
| atomic-step-ipc (current s) dir stage partner page s
| SK-EV-WAIT EV-PREP consume ⇒ s
| SK-EV-WAIT EV-WAIT consume ⇒ s
| SK-EV-WAIT EV-FINISH consume ⇒
| case consume of
| EV-CONSUME-ONE ⇒ atomic-step-ev-wait-one (current s) s
| EV-CONSUME-ALL ⇒ atomic-step-ev-wait-all (current s) s
| SK-EV-SIGNAL EV-SIGNAL-PREP partner ⇒ s
| SK-EV-SIGNAL EV-SIGNAL-FINISH partner ⇒
| atomic-step-ev-signal (current s) partner s
| NONE ⇒ s
| end

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

**theory Step-invariants**

**imports Step**

**begin**

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

**definition sp-subset s ≡**
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 asms 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 ?thesis unfolding atomic-step-def
  using assms atomic-ipc-preserves-invariants
  by simp
next case (SK-EV-WAIT ev-wait-stage consume)
  then show ?thesis
  proof (cases consume)
  case EV-CONSUME-ALL
  then show ?thesis unfolding atomic-step-def
  using SK-EV-WAIT assms atomic-ev-wait-all-preserves-invariants
  by (simp split: ev-wait-stage-t.splits)
next case EV-CONSUME-ONE
  then show ?thesis unfolding atomic-step-def
  using SK-EV-WAIT assms atomic-ev-wait-one-preserves-invariants
  by (simp split: ev-wait-stage-t.splits)
qed
next case SK-EV-SIGNAL
  then show ?thesis unfolding atomic-step-def
  using assms atomic-ev-signal-preserves-invariants
  by (simp add: ev-signal-stage-t.splits)
next case NONE
  then show ?thesis unfolding atomic-step-def
  using assms
  by auto
qed

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

theorem cswitch-preserves-invariants:
  fixes s :: state-t
  and new-current :: thread-id-t
  assumes atomic-step-invariant s
  shows atomic-step-invariant (s (current := new-current))
proof -
  let ?s1 = s (current := new-current)
  have sp-subset s = sp-subset ?s1
    unfolding sp-subset-def by auto
  from assms this show ?thesis
    unfolding atomic-step-invariant-def by metis
qed

theorem atomic-step-does-not-change-current-thread:
  shows current (atomic-step s ipt) = current s
proof -
  show ?thesis
    unfolding atomic-step-def
    and atomic-step-ipc-def
    and set-object-value-def Let-def
    and atomic-step-ev-wait-one-def atomic-step-ev-wait-all-def
    and atomic-step-ev-signal-def
    by (simp split: int-point-t.splits ipc-stage-t.splits ipc-direction-t.splits ev-consume-t.splits ev-wait-stage-t.splits ev-signal-stage-t.splits)
qed

end
4.5 The view-partitioning equivalence relation

theory Step-vpeq
imports Step Step-invariants
begin

The view consists of

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

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

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

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

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

definition vpeq u s t ≡
vpeq-obj u s t ∧ vpeq-subj-subj u s t ∧ vpeq-subj-obj u s t ∧ vpeq-local u s t

4.5.1 Elementary properties

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

Auxiliary equivalence relation.

lemma set-object-value-ign:
assumes eq-obs: ~ Policy.sp-spec-subj-obj u x READ
shows vpeq u s (set-object-value x y s)
proof –
from assms show ?thesis
unfolding vpeq-def vpeq-obj-def vpeq-subj-subj-def vpeq-subj-obj-def set-object-value-def
vpeq-local-def
by auto
Context-switch and fetch operations are also consistent with vpeq and locally respect everything.

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

```plaintext
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
```

4.6 Atomic step locally respects the information flow policy

**Theory Step-vpeq-locally-respects**

```plaintext
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:**

```plaintext
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

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

lemma ev-wait-one-respects-policy:
assumes no: ~ Policy.ifp (partition tid) u
and inv: atomic-step-invariant s
and prec: atomic-step-precondition s tid ipt
and ipt-case: ipt = SK-EV-WAIT ev-wait-stage EV-CONSUME-ONE
shows vpeq u s (atomic-step-ev-wait-one tid s)
proof –
from assms have 1: (partition tid) # u
unfolding Policy.ifp-def
by simp
then have 2: vpeq-local u s (atomic-step-ev-wait-one tid s)
unfolding vpeq-local-def atomic-step-ev-wait-one-def
by simp
have 3: vpeq-obj u s (atomic-step-ev-wait-one tid s)
unfolding vpeq-obj-def atomic-step-ev-wait-one-def
by simp
have 4: vpeq-subj-subj u s (atomic-step-ev-wait-one tid s)
unfolding vpeq-subj-subj-def atomic-step-ev-wait-one-def
by simp
have 5: vpeq-subj-obj u s (atomic-step-ev-wait-one tid s)
unfolding vpeq-subj-obj-def atomic-step-ev-wait-one-def
by simp
with 2 3 4 5 show ?thesis
unfolding vpeq-def
by simp
qed

4.6.2 Summary theorems on view-partitioning locally respects

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

theorem atomic-step-respects-policy:
assumes na: ~ Policy.ifp (partition (current s)) u
and inv: atomic-step-invariant s
and prec: atomic-step-precondition s (current s) ipt
shows vpeq u s (atomic-step s ipt)
proof –
show ?thesis
using assms ipc-respects-policy vpeq-refl
ev-signal-respects-policy ev-wait-one-respects-policy
ev-wait-all-respects-policy
unfolding atomic-step-def
by \((\text{auto split: int-point-t.splits ev-consume-t.splits ev-wait-stage-t.splits ev-signal-stage-t.splits})\)
qed

end

4.7 Weak step consistency

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

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

4.7.1 Weak step consistency of auxiliary functions

\begin{lemma}
\textit{ipc-precondition-weakly-step-consistent:}
\end{lemma}

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

\begin{shows}
\textit{ipc-precondition tid dir partner page s1 = ipc-precondition tid dir partner page s2}
\end{shows}

\begin{proof}
\begin{align*}
\text{let } ?sender & = \text{case dir of SEND \Rightarrow tid | RECV \Rightarrow partner} \\
\text{let } ?receiver & = \text{case dir of SEND \Rightarrow partner | RECV \Rightarrow tid} \\
\text{let } ?local-access-mode & = \text{case dir of SEND \Rightarrow READ | RECV \Rightarrow WRITE} \\
\text{let } ?A & = \text{sp-impl-subj-subj s1 (partition ?sender) (partition ?receiver)} \\
& \quad = \text{sp-impl-subj-subj s2 (partition ?sender) (partition ?receiver)} \\
\text{let } ?B & = \text{sp-impl-subj-obj s1 (partition tid) (PAGE page) ?local-access-mode} \\
& \quad = \text{sp-impl-subj-obj s2 (partition tid) (PAGE page) ?local-access-mode}
\end{align*}

\text{have } A: ?A \\
\text{proof (cases Policy.sp-spec-subj-subj (partition ?sender) (partition ?receiver))} \\
\hspace{1em} \text{case True} \\
\hspace{2em} \text{thus } ?A \\
\hspace{3em} \text{using eq-tid unfolding vpeq-def vpeq-subj-subj-def} \\
\hspace{4em} \text{by (simp split: ipc-direction-t.splits)} \\
\text{next case False} \\
\hspace{1em} \text{have sp-subset s1 and sp-subset s2} \\
\hspace{2em} \text{using inv1 inv2 unfolding atomic-step-invariant-def sp-subset-def by auto} \\
\hspace{3em} \text{hence } \neg \text{sp-impl-subj-subj s1 (partition ?sender) (partition ?receiver)} \\
\hspace{4em} \text{and } \neg \text{sp-impl-subj-subj s2 (partition ?sender) (partition ?receiver)} \\
\hspace{5em} \text{using False unfolding sp-subset-def by auto} \\
\hspace{6em} \text{thus } ?A \text{ by auto} \\
\text{qed}
\end{proof}

\text{have } B: ?B \\
\text{proof (cases Policy.sp-spec-subj-obj (partition tid) (PAGE page) ?local-access-mode)} \\
\hspace{1em} \text{case True} \\
\hspace{2em} \text{thus } ?B \\
\hspace{3em} \text{using eq-tid unfolding vpeq-def vpeq-subj-obj-def} \\
\hspace{4em} \text{by (simp split: ipc-direction-t.splits)} \\
\hspace{5em} \text{next case False} \\
\hspace{2em} \text{have sp-subset s1 and sp-subset s2} \\
\hspace{3em} \text{using inv1 inv2 unfolding atomic-step-invariant-def sp-subset-def by auto} \\
\hspace{4em} \text{hence } \neg \text{sp-impl-subj-obj s1 (partition tid) (PAGE page) ?local-access-mode} \\
\hspace{5em} \text{and } \neg \text{sp-impl-subj-obj s2 (partition tid) (PAGE page) ?local-access-mode} \\
\hspace{6em} \text{using False unfolding sp-subset-def by auto} \\
\hspace{7em} \text{thus } ?B \text{ by auto} \\
\end{proof}

\end{lemma}
qed
show ?thesis using A B unfolding ipc-precondition-def by auto
qed

lemma ev-signal-precondition-weakly-step-consistent:
assumes eq-tid: vpeq (partition tid) s1 s2
and inv1: atomic-step-invariant s1
and inv2: atomic-step-invariant s2
shows ev-signal-precondition tid partner s1 = ev-signal-precondition tid partner s2
proof –
let ?A = sp-impl-subj-subj s1 (partition tid) (partition partner) = sp-impl-subj-subj s2 (partition tid) (partition partner)
have A: ?A
proof (cases Policy.sp-spec-subj-subj (partition tid) (partition partner))
case True
  thus ?A
  using eq-tid unfolding vpeq-def vpeq-subj-subj-def
  by (simp split: 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
4.7.2 Weak step consistency of atomic step functions

lemma ipc-weakly-step-consistent:
assumes eq-obs: vpeq u s1 s2
  and eq-act: vpeq (partition tid) s1 s2
  and inv1: atomic-step-invariant s1
  and inv2: atomic-step-invariant s2
  and prec1: atomic-step-precondition s1 tid ipt
  and prec2: atomic-step-precondition s1 tid ipt
  and ipt-case: ipt = SK-IPC dir stage partner page
shows vpeq u
  (atomic-step-ipc tid dir stage partner page s1)
  (atomic-step-ipc tid dir stage partner page s2)
proof –
  have \( \forall \) mypage . \[ \text{dir = SEND; stage = BUF mypage} \] \( \implies \) \( ?\)thesis
proof –
  fix mypage
  assume dir-send: \( \text{dir = SEND} \)
  assume stage-buf: stage = BUF mypage
  have Policy.sp-spec-subj-obj (partition tid) (PAGE page) READ
    using inv1 prec1 dir-send stage-buf ipt-case
    unfolding atomic-step-invariant-def sp-subset-def
    unfolding atomic-step-precondition-def ipc-precondition-def opposite-ipc-direction-def
    by auto
  hence \( \text{obj s1 (PAGE page) = obj s2 (PAGE page)} \)
    using eq-act unfolding vpeq-def vpeq-obj-def vpeq-local-def
    by auto
  thus vpeq u
    (atomic-step-ipc tid dir stage partner page s1)
    (atomic-step-ipc tid dir stage partner page s2)
    using dir-send stage-buf eq-obs set-object-value-consistent
    unfolding atomic-step-ipc-def
    by auto
  qed
thus \( ?\)thesis
  using eq-obs unfolding atomic-step-ipc-def
  by (cases stage, auto, cases dir, auto)
qed
lemma ev-wait-one-weakly-step-consistent:
assumes eq-obs: vpeq u s1 s2
and eq-act: vpeq (partition tid) s1 s2
and inv1: atomic-step-invariant s1
and inv2: atomic-step-invariant s2
and prec1: atomic-step-precondition s1 (current s1) ipt
and prec2: atomic-step-precondition s1 (current s1) ipt
shows vpeq u
  (atomic-step-ev-wait-one tid s1)
  (atomic-step-ev-wait-one tid s2)
using assms
unfolding vpeq-def vpeq-subj-subj-def vpeq-obj-def vpeq-subj-obj-def vpeq-local-def
  atomic-step-ev-wait-one-def
by simp

lemma ev-wait-all-weakly-step-consistent:
assumes eq-obs: vpeq u s1 s2
and eq-act: vpeq (partition tid) s1 s2
and inv1: atomic-step-invariant s1
and inv2: atomic-step-invariant s2
and prec1: atomic-step-precondition s1 (current s1) ipt
and prec2: atomic-step-precondition s1 (current s1) ipt
shows vpeq u
  (atomic-step-ev-wait-all tid s1)
  (atomic-step-ev-wait-all tid s2)
using assms
unfolding vpeq-def vpeq-subj-subj-def vpeq-obj-def vpeq-subj-obj-def vpeq-local-def
  atomic-step-ev-wait-all-def
by simp

lemma ev-signal-weakly-step-consistent:
assumes eq-obs: vpeq u s1 s2
and eq-act: vpeq (partition tid) s1 s2
and inv1: atomic-step-invariant s1
and inv2: atomic-step-invariant s2
and prec1: atomic-step-precondition s1 (current s1) ipt
and prec2: atomic-step-precondition s1 (current s1) ipt
shows vpeq u
  (atomic-step-ev-signal tid partner s1)
  (atomic-step-ev-signal tid partner s2)
using assms
unfolding vpeq-def vpeq-subj-subj-def vpeq-obj-def vpeq-subj-obj-def vpeq-local-def
  atomic-step-ev-signal-def
by simp

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

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

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

lemma extend-subj-subj-consistent:
fixes f :: partition-id-t ⇒ partition-id-t ⇒ bool
assumes vpeq u s1 s2
shows \( \text{vpeq} \ u \ (\text{extend-subj-subj} \ f \ s1) \ (\text{extend-subj-subj} \ f \ s2) \)

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

using assms unfolding vpeq-subj-subj-def extend-subj-subj-def by auto

have \(2\)∶ \(\text{vpeq-obj} \ u \ (\text{extend-subj-subj} \ f \ s1) \ (\text{extend-subj-subj} \ f \ s2)\)

by auto

have \(3\)∶ \(\text{vpeq-local} \ u \ (\text{extend-subj-subj} \ f \ s1) \ (\text{extend-subj-subj} \ f \ s2)\)

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: \(\text{vpeq} \ (\text{partition} \ (\text{current} \ s1)) \ s1 \ s2\)

and inv1: \(\text{atomic-step-invariant} \ s1\)

and inv2: \(\text{atomic-step-invariant} \ s2\)

and prec1: \(\text{atomic-step-precondition} \ s1 \ (\text{current} \ s1) \ ipt\)

and prec2: \(\text{atomic-step-precondition} \ s2 \ (\text{current} \ s2) \ ipt\)

and eq-curr: \(\text{current} \ s1 = \text{current} \ s2\)

shows \(\text{vpeq} \ u \ (\text{atomic-step} \ s1 \ ipt) \ (\text{atomic-step} \ s2 \ ipt)\)

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.

```plaintext
cons
  initial-current :: thread-id-t
  initial-obj :: obj-id-t => obj-t

definition
  s0 :: state-t where
  s0 = (| sp-impl-subj-subj = Policy.sp-spec-subj-subj,
          sp-impl-subj-obj = Policy.sp-spec-subj-obj,
          current = initial-current,
          obj = initial-obj,
          thread = \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 at witness that `rstate-t` is non-empty.

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

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

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

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

lemma rstate-down-up[simp]:
  shows (↑↓s) = s
  unfolding rep-def abs-def using Rep-rstate-t-inverse by auto
```
lemma \textit{rstate-up-down}:\ \textbf{simp}:
\begin{itemize}
  \item \textbf{assumes} \textit{atomic-step-invariant} \( s \)
  \item \textbf{shows} \( (↓\downarrow s) = s \)
  \item \textbf{using} \textit{assms} \textit{Abs-rstate-t-inverse} \textbf{unfolding} \textit{rep-def} \textit{abs-def} \textbf{by auto}
\end{itemize}

A CISK action is identified with an interrupt point.

type-synonym \textit{raction-t} = \textit{int-point-t}

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

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

Each CISK domain is identified with a thread id.

type-synonym \textit{rdom-t} = \textit{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 \textit{visible-obj-t} = \textit{VALUE} \textit{obj-t} | \textit{EXCEPTION}

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

definition \textit{routput-f} :: \textit{rstate-t} \Rightarrow \textit{raction-t} \Rightarrow \textit{routput-t} where
\textit{routput-f} \( s \) \( a \) \( p \) ≡ 
\begin{itemize}
  \item \textbf{if} \textit{sp-impl-subj-obj} (\( ↓\downarrow s \)) \textbf{and} \textit{partition} (\textit{rcurrent} \( s \)) \textbf{and} \( \textit{PAGE} \( p \) \) \textbf{then}
    \textit{VALUE} (\textit{obj} (\( ↓\downarrow s \)) (\textit{PAGE} \( p \)) )
  \item \textbf{else}
    \textit{EXCEPTION}
\end{itemize}

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

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

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

Translate view-partitioning and interaction-allowed relations.

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

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

Context Switches

definition \textit{rcswitch} :: \textit{nat} \Rightarrow \textit{rstate-t} \Rightarrow \textit{rstate-t} where
\textit{rcswitch} \( n \) \( s \) ≡ \textbf{true} ((\( ↓\downarrow s \)) (\textit{current} := (\textit{SOME} \textit{t} . \textit{True}) ) )

4.8.3 Possible action sequences

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

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

**definition** is-SK-EV-WAIT :: raction-t list ⇒ bool

**where** is-SK-EV-WAIT aseq ≡ ∃ consume .

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

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

**definition** is-SK-EV-SIGNAL :: raction-t list ⇒ bool

**where** is-SK-EV-SIGNAL aseq ≡ ∃ partner .

aseq = [SK-EV-SIGNAL EV-SIGNAL-PREP partner , SK-EV-SIGNAL EV-SIGNAL-FINISH partner]

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

**definition** rAS-set :: raction-t list set

**where** rAS-set ≡ { aseq . is-SK-IPC aseq v is-SK-EV-WAIT aseq v is-SK-EV-SIGNAL aseq } ∪ {}

### 4.8.4 Control

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

**abbreviation** raborting

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

**abbreviation** rwaiting

**where** rwaiting s ≡ waiting (↓ s)

**definition** rset-error-code :: rstate-t ⇒ raction-t ⇒ rstate-t

**where** rset-error-code s a ≡ s

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

**definition** rkinvolved :: int-point-t ⇒ rdom-t set

**where** rkinvolved a ≡ case a of SK-IPC dir WAIT partner page ⇒ {partner}

| SK-EV-SIGNAL EV-SIGNAL-FINISH partner ⇒ {partner} |

| - ⇒ {} |

**abbreviation** rinvolved

**where** rinvolved ≡ Kernel.involved rkinvolved

### 4.8.5 Discharging the proof obligations

**lemma** inst-vpeq-rel:

**shows** rvpeq-refl: rvpeq u s s

and rvpeq-sym: rvpeq u s1 s2 ⇒ rvpeq u s2 s1

and rvpeq-trans: [[ rvpeq u s1 s2; rvpeq u s2 s3 ]] ⇒ rvpeq u s1 s3

unfolding rvpeq-def using vpeq-rel by metis+

**lemma** inst-ifp-refl:

**shows** ∀ u . rifp u u

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

**lemma** inst-step-atomicity [simp]:

**shows** ∀ s a . rcurrent (rstep s a) = rcurrent s

unfolding rstep-def rcurrent-def
using atomic-step-does-not-change-current-thread rstate-up-down rstate-invariant atomic-step-preserves-invariants 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 :: action-t
    fix s t :: rstate-t
    fix p :: page-t
    assume 1: rvpeq (rcurrent s) s t
    and 2: rcurrent s = rcurrent t
    let ?part = partition (rcurrent s)
    have routput-f s a p = routput-f t a p
    proof (cases Policy.sp-spec-subj-obj ?part (PAGE p) READ rule: case-split [case-names Allowed Denied])
    case Allowed
    have 5: obj (↓s) (PAGE p) = obj (↓t) (PAGE p)
    using 1 Allowed unfolding rvpeq-def vpeq-def vpeq-obj-def by auto
    have 6: sp-impl-subj-obj (↓s) ?part (PAGE p) READ = sp-impl-subj-obj (↓t) ?part (PAGE p) READ
    using 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 (↓s) ?part (PAGE p) READ = False
    and sp-impl-subj-obj (↓t) ?part (PAGE p) READ = False
    using rstate-invariant unfolding atomic-step-invariant-def sp-subset-def
by auto
thus \( \text{routput-f}\ s\ a\ p = \text{routput-f}\ t\ a\ p \)
using 2 unfolding \( \text{routput-f-def} \) by simp
qed 

thus \( \forall\ a\ s\ t.\ \text{rvpeq}\ (\text{rcurrent}\ s)\ s\ t \wedge\ \text{rcurrent}\ s = \text{rcurrent}\ t \rightarrow\ \text{routput-f}\ s\ a = \text{routput-f}\ t\ a \)
by auto
qed

thus \( ?\text{thesis} \) using \( \text{assms} \) by auto
qed

\[
\text{lemma\ inst-cswitch-independent-of-state:}\n\begin{align*}
\text{assumes}\ & \text{rcurrent}\ s = \text{rcurrent}\ t \\
\text{shows}\ & \text{rcurrent}\ (\text{rcswitch}\ n\ s) = \text{rcurrent}\ (\text{rcswitch}\ n\ t) \\
\text{using}\ & \text{rstate-invariant cswitch-preserves-invariants}\ unfolding\ \text{rcurrent-def}\ \text{rcswitch-def}\ by\ simp
\end{align*}
\]

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

qed

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

\[
\text{lemma\ prec-first-IPC-action:}\n\begin{align*}
\text{assumes}\ & \text{is-SK-IPC}\ aseq \\
\text{shows}\ & \text{rprecondition}\ s\ d\ (\text{hd}\ aseq) \\
\text{using}\ & \text{assms} \\
\text{unfolding}\ & \text{is-SK-IPC-def}\ \text{rprecondition-def}\ \text{atomic-step-precondition-def} \\
\text{by}\ & auto \\
\end{align*}
\]

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

\[
\text{lemma\ prec-first-EV-WAIT-action:}\n\begin{align*}
\text{assumes}\ & \text{is-SK-EV-WAIT}\ aseq \\
\text{shows}\ & \text{rprecondition}\ s\ d\ (\text{hd}\ aseq) \\
\text{using}\ & \text{assms} \\
\text{unfolding}\ & \text{is-SK-EV-WAIT-def}\ \text{rprecondition-def}\ \text{atomic-step-precondition-def} \\
\text{by}\ & auto \\
\end{align*}
\]

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

\[
\text{lemma\ prec-first-EV-SIGNAL-action:}\n\begin{align*}
\text{assumes}\ & \text{is-SK-EV-SIGNAL}\ aseq \\
\text{shows}\ & \text{rprecondition}\ s\ d\ (\text{hd}\ aseq) \\
\text{using}\ & \text{assms} \\
\text{unfolding}\ & \text{is-SK-EV-SIGNAL-def}\ \text{rprecondition-def}\ \text{atomic-step-precondition-def} \\
\text{by}\ & auto \\
\end{align*}
\]

EURO-MILS D31.1 Page 80 of 94
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**
- \( \text{prec} : \text{rprecondition } s \ (rcurrent \ s) \ (aseq \ ! \ n) \)
- \( n \text{-bound}: \text{Suc } n < \text{length } aseq \)
- \( \text{IPC}: \text{is-SK-IPC } aseq \)
- \( \text{not-aborting}: \neg \text{raborting } s \ (rcurrent \ s) \ (aseq \ ! \ n) \)
- \( \text{not-waiting}: \neg \text{rwaiting } s \ (rcurrent \ s) \ (aseq \ ! \ n) \)
**shows** \( \text{rprecondition } (rstep \ s \ (aseq \ ! \ n)) \ (rcurrent \ s) \ (aseq \ ! \ Suc \ n) \)
**proof**

- \{ fix \( \text{dir partner page} \)
  let \( ?\text{page} = (\text{SOME page}', \text{True}) \)
  assume \( \text{IPC: } aseq = \left[ \text{SK-IPC dir PREP partner page,SK-IPC dir WAIT partner page,SK-IPC dir (BUF ?page')} \right. \)
  partner page] \}
  \{ assume \( 0: n=0 \)
  from \( 0 \text{ IPC prec not-aborting} \)
  have \( \text{thesis} \)
  unfolding \( \text{rprecondition-def atomic-step-precondition-def rstep-def rcurrent-def atomic-step-def atomic-step-ipc-def aborting-def} \)
  by(\text{auto}) \}
  moreover
  \{ assume \( 1: n=1 \)
  from \( 1 \text{ IPC prec not-waiting} \)
  have \( \text{thesis} \)
  unfolding \( \text{rprecondition-def atomic-step-precondition-def rstep-def rcurrent-def atomic-step-def atomic-step-ipc-def waiting-def} \)
  by(\text{auto}) \}
  moreover from \( \text{IPC} \)
  have \( \text{length } aseq = 3 \)
  by \text{auto} 
  ultimately have \( \text{thesis} \)
  using \( n\text{-bound} \)
  by \text{arith} \}
thus \( \text{thesis} \)
using \( \text{IPC} \)
unfolding \( \text{is-SK-IPC-def} \)
by(\text{auto})
qed

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

**lemma** prec-after-EV-WAIT-step:
**assumes**
- \( \text{prec} : \text{rprecondition } s \ (rcurrent \ s) \ (aseq \ ! \ n) \)
- \( n\text{-bound}: \text{Suc } n < \text{length } aseq \)
- \( \text{IPC}: \text{is-SK-EV-WAIT } aseq \)
- \( \text{not-aborting}: \neg \text{raborting } s \ (rcurrent \ s) \ (aseq \ ! \ n) \)
- \( \text{not-waiting}: \neg \text{rwaiting } s \ (rcurrent \ s) \ (aseq \ ! \ n) \)
**shows** \( \text{rprecondition } (rstep \ s \ (aseq \ ! \ n)) \ (rcurrent \ s) \ (aseq \ ! \ Suc \ n) \)
**proof**


fix consume

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

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

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

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

proof-
{   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
aborting-def rstep-def atomic-step-def
by auto
}
moreover
from SIGNAL1
have length aseq = 2
by auto
ultimately
have \( ? \text{thesis} \)
using n-bound
by arith
}
thus \( ? \text{thesis} \)
using assms
unfolding is-SK-EV-SIGNAL-def
by auto
qed

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

lemma prec-IPC-dom-independent:
assumes current \( s \neq d \)
and atomic-step-invariant s
and atomic-step-precondition s d a
shows atomic-step-precondition (atomic-step-ipc (current s) dir stage partner page s) d a
using assms on-set-object-value
unfolding atomic-step-precondition-def atomic-step-ipc-def ipc-precondition-def
by (auto split: int-point-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 \( s \neq d \)
and atomic-step-invariant s
and atomic-step-precondition s d a
shows atomic-step-precondition (atomic-step-ev-signal (current s) partner s) d a
using assms on-set-object-value
unfolding atomic-step-precondition-def atomic-step-ev-signal-def ipc-precondition-def
by (auto split: int-point-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 \( s \neq d \)
and atomic-step-invariant s
and atomic-step-precondition s d a
shows atomic-step-precondition (atomic-step-ev-wait-one (current s) s) d a
using assms on-set-object-value
unfolding atomic-step-precondition-def atomic-step-ev-wait-one-def ipc-precondition-def
by (auto split: int-point-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 \( s \neq d \)
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
lemma prec-dom-independent:
shows ∀ s d a a'. rcurrent s \neq d ∧ rprecondition s d a \rightarrow rprecondition (rstep s a') d a
using atomic-step-preserves-invariants
rstate-invariant prec-IPC-dom-independent prec-ev-signal-dom-independent
prec-ev-wait-all-dom-independent prec-ev-wait-one-dom-independent
unfolding rcurrent-def rprecondition-def rstep-def atomic-step-def
by (auto split: int-point-t.splits ipc-stage-t.splits ev-consume-t.splits ev-wait-stage-t.splits ev-signal-stage-t.splits)

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

lemma pre-condition-after-cswitch:
shows ∀ s d n a. rprecondition s d a \rightarrow rprecondition (rcswitch n s) d a
using cswitch-preserves-invariants rstate-invariant
unfolding rprecondition-def rcswitch-def atomic-step-precondition-def
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
switch-consistency-and-respect
unfolding aborting-def rcsswitch-def
apply (auto split: int-point-t.splits ipc-stage-t.splits ev-signal-stage-t.splits)
ev-wait-stage-t.splits ev-signal-stage-t.splits)
apply (metis (full-types))
by blast
}
hence raborting (rcswitch n s) = raborting s by auto
}
thus ?thesis by auto
qed

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

\textbf{lemma \text{aborting-after-IPC-step:}}
\textbf{assumes } d_1 \neq d_2
\textbf{shows } \text{aborting} (\text{atomic-step-ipc } d_1 \text{ dir stage partner page } s) \ d_2 \ a = \text{aborting } s \ d_2 \ a
\textbf{unfolding } \text{atomic-step-ipc-def aborting-def set-object-value-def ipc-precondition-def ev-signal-precondition-def}
\textbf{by} (auto split: int-point-t.splits ipc-stage-t.splits ipc-direction-t.splits
\text{ev-signal-stage-t.splits})

\textbf{lemma \text{waiting-after-IPC-step:}}
\textbf{assumes } d_1 \neq d_2
\textbf{shows } \text{waiting} (\text{atomic-step-ipc } d_1 \text{ dir stage partner page } s) \ d_2 \ a = \text{waiting } s \ d_2 \ a
\textbf{unfolding } \text{atomic-step-ipc-def waiting-def set-object-value-def ipc-precondition-def ev-signal-precondition-def}
\textbf{by} (auto split: int-point-t.splits ipc-stage-t.splits ipc-direction-t.splits
\text{ev-wait-stage-t.splits})

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

\textbf{lemma \text{aborting-dom-independent:}}
\textbf{assumes } \text{rcurrent } s \neq d
\textbf{shows } \text{raborting} (\text{rstep } s \ a) \ d \ a' = \text{raborting } s \ d \ a'
\textbf{proof -}
\textbf{have } \land \text{tid dir partner page } s . \text{ipc-precondition tid dir partner page } s = \text{ipc-precondition tid dir partner page} (\text{atomic-step } s \ a)
\land \text{ev-signal-precondition tid partner } s = \text{ev-signal-precondition tid partner} (\text{atomic-step } s \ a)
\textbf{proof -}
\textbf{fix } \text{tid dir partner page } s
\textbf{let } ?s = \text{atomic-step } s \ a
have \( (\forall p q . \text{sp-impl-subj-subj } s p q = \text{sp-impl-subj-subj } ?s p q) \) \\
\land (\forall p x m . \text{sp-impl-subj-obj } s p x m = \text{sp-impl-subj-obj } ?s p x m) \\

unfolding \text{atomic-step-def} \text{atomic-step-ipc-def} \\
\text{atomic-step-ev-wait-all-def} \text{atomic-step-ev-wait-one-def} \\
\text{atomic-step-ev-signal-def} \text{set-object-value-def} \\
by (\text{auto split: int-point-t.splits ipc-stage-t.splits ipc-direction-t.splits} \\
\text{ev-wait-stage-t.splits ev-consume-t.splits} \text{ev-signal-stage-t.splits}) \\
thus \text{ipc-precondition tid dir partner page } s = \text{ipc-precondition tid dir partner page } (\text{atomic-step } s a) \\
\land \text{ev-signal-precondition tid partner } s = \text{ev-signal-precondition tid partner } (\text{atomic-step } s a) \\

unfolding \text{ipc-precondition-def} \text{ev-signal-precondition-def} by \text{simp} \\
qed \\
moreover have \( (\forall b . \text{atomic-step } (\downarrow \uparrow (\text{atomic-step } (\downarrow \uparrow b)))) = \text{atomic-step } (\downarrow \uparrow b) \)
using \text{rstate-invariant} \text{atomic-step-preserves-invariants} \text{rstate-up-down by} \text{auto} \\
ultimately show \( ?\text{thesis} \)
unfolding \text{aborting-def} \text{rstep-def} \text{ev-signal-precondition-def} \\
by (\text{simp split: int-point-t.splits ipc-stage-t.splits ev-wait-stage-t.splits} \\
\text{ev-signal-stage-t.splits}) \\
qed \\

lemma \text{ipc-precondition-of-partner-consistent:} \\
assumes \( \forall d \in \text{rkinvolved } (\text{SK-IPC } \text{dir } \text{WAIT } \text{partner page}) \) . \text{rypeq } d s t \\
shows \text{ipc-precondition partner } \text{dir'} \text{ u page'} (\downarrow s) = \text{ipc-precondition partner } \text{dir'} \text{ u page'} (\downarrow t) \\
proof− \\
\{ \text{fix } s t u a \} \\
\text{assume} \text{vpeq:} \text{rypeq } (\text{rcurrent } s) s t \land (\forall d \in \text{rkinvolved } a . \text{rypeq } d s t) \\
\text{assume} \text{vpeq-involved:} \forall d \in \text{rkinvolved } a . \text{rypeq } d s t \\
\text{assume} \text{vpeq-u:} \text{rypeq } u s t \\
\text{have} \text{rwaiting } s u a = \text{rwaiting } t u a \text{ proof (cases a)} \\
\text{case } \text{SK-IPC} \\
\text{ thus} \text{rwaiting } s u a = \text{rwaiting } t u a \\
\text{using} \text{ipc-precondition-of-partner-consistent} \text{vpeq-involved} \\
\text{unfolding} \text{waiting-def} \text{by} (\text{auto split: ipc-stage-t.splits}) \\
\text{next case } \text{SK-EV-WAIT} \\
\text{ thus} \text{rwaiting } s u a = \text{rwaiting } t u a \\
qed
using ev-signal-precondition-of-partner-consistent
vpeq-involved vpeq vpeq-u
unfolding waiting-def rkinvolved-def ev-signal-precondition-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 = \((t1, t2) \to Policy.sp-spec-subj-subj \((\text{partition } t1) \to \text{partition } t2)\)
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 = \((t1, t2) \to Policy.sp-spec-subj-subj \((\text{partition } t1) \to \text{partition } t2)\)
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 \(\forall \ a \ . \ \forall \ d \in rkinvolved \ a \ . \ \text{rprecondition} \ (\text{rcurrent } s) \ a \to \text{rifp} \ d \ (\text{rcurrent } s)\)
proof–
{\text{fix } s \ a \ d
assume d-involved: d \in rkinvolved \ a
assume prec: rprecondition \ (\text{rcurrent } s) \ a
from d-involved prec have rifp d \ (\text{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 \(\forall \ s \ a \ . \ \text{rwaiting} \ s \ (\text{rcurrent } s) \ (SK-EV-WAIT EV-FINISH EV-CONSUME-ALL) \to \text{rstep} \ s \ a = s\)
unfolding waiting-def
by auto
lemma spec-of-waiting-ev-w:
shows ∀ s a. rwaiting s (rcurrent s) (SK-EV-WAIT EV-WAIT EV-CONSUME-ALL) → rstep s (SK-EV-WAIT EV-WAIT EV-CONSUME-ALL) = s
unfolding rstep-def atomic-step-def
by (auto split: int-point-t.splits ipc-stage-t.splits ev-wait-stage-t.splits)

lemma spec-of-waiting:
shows ∀ s a. rwaiting s (rcurrent s) a → rstep s a = s
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
(r↑)
rcurrent
rcswitch
rkinvolved
rifp
rvpeq
rAS-set
rinvariant
rprecondition
raborting
rwating
rset-error-code

proof (unfold-locales)
— show that rvpeq is equivalence relation
show ∀ a b c u. (rvpeq u a b ∧ rvpeq u b c) → rvpeq u a c
and ∀ a b u. rvpeq u a b → rvpeq u b a
and ∀ a u. rvpeq u a a
using inst-vpeq-rel by metis+
— show output consistency
show ∀ a s t. rvpeq (rcurrent s) s t ∧ rcurrent s = rcurrent t → routput-f s a = routput-f t a
using inst-output-consistency by metis
— show reflexivity of ifp
show ∀ u. rifp u u
using inst-ifp-refl by metis
— show step consistency
show ∀ s t a. rvpeq u s t ∧ rvpeq (rcurrent s) s t ∧ True ∧ rprecondition s (rcurrent s) a ∧ True ∧ rprecondition t (rcurrent t) a ∧ rcurrent s = rcurrent t → rvpeq u (rstep s a) (rstep t a)
using inst-weakly-step-consistent by blast
— show step atomicity
D31.1 – Formal Specification of a Generic Separation Kernel

show ∀ s a . rcurrent (rstep s a) = rcurrent s
  using inst-step-atomicity by metis

show ∀ a s u. ¬ rifp (rcurrent s) u ∧ True ∧ precondition s (rcurrent s) a → rvpeq u s (rstep s a)
  using inst-local-respect by blast
  — show cswitch is independent of state

show ∀ s a u. (rcurrent s) a → rcurrent (rstep s a) = rcurrent s
  using inst-cswitch-independent-of-state by metis

— show cswitch consistency

show ∀ a s u. ¬ rifp (rcurrent s) u ∧ True ∧ rprecondition s (rcurrent s) a /
  → rvpeq u s (rstep s a)
  using inst-cswitch-consistency by metis

— Show the empt action sequence is in AS-set

show [] ∈ rAS-set
  unfolding rAS-set-def by auto

— The invariant for the initial state, already encoded in rstate-t

show True
  by auto

— Step function of the invariant, already encoded in rstate-t

show ∀ s n. True → True
  by auto

— The precondition does not change with a context switch

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

show ∀ s d aseq. True ∧ aseq ∈ rAS-set ∧ aseq ≡ [] → rprecondition s d (hd aseq)
  using prec-first-IPC-action prec-first-EV-WAIT-action prec-first-EV-SIGNAL-action
  unfolding rAS-set-def is-sub-seq-def by auto

— The precondition holds for the next action in an action sequence, assuming the sequence is not aborted or delayed

show ∀ s n a a'. (∃ aseq rAS-set. is-sub-seq a a' aseq) ∧ True ∧ rprecondition s (rcurrent s) a ∧ ¬ raborting s
  (rcurrent s) a ∧ ¬ rwaiting s (rcurrent s) a → 
  rprecondition (rstep s a) (rcurrent s) a'
  unfolding rAS-set-def is-sub-seq-def by auto

— Steps of other domains do not influence the precondition

show ∀ s d a a'. current s d a → rprecondition s a → rprecondition (rstep s a) d a
  using prec-dom-independent by blast

— The invariant

show ∀ s a. True → True
  by auto

— Aborting does not depend on a context switch

show ∀ n s. raborting (rcswitch n s) = raborting s
  using aborting-switch-independent by auto

— Aborting does not depend on actions of other domains

show ∀ s a d. current s d a → raborting (rstep s a) d = raborting s d
  using aborting-dom-independent by auto

— Aborting is consistent

show ∀ s t u. rvpeq u s t → raborting s u = raborting t u
  using raborting-consistent by auto

— Waiting does not depend on a context switch

show ∀ n s. rwaiting (rcswitch n s) = rwaiting s
  using waiting-switch-independent by auto

— Waiting is consistent

show ∀ s t u a. rvpeq (rcurrent s) s t ∧ (∀ d ∈ rkinvolved a . rvpeq d s t) ∧ rvpeq u s t

EURO-MILS D31.1 Page 89 of 94
Now we can instantiate CISK with some initial state, interrupt function, etc.

\[ \text{interpretation \ } \text{Inst} \]

\begin{align*}
\text{Controllable-Interruptible-Separation-Kernel} \\
\text{rstep \ } & \quad \text{step function, without program stack} \\
\text{routput-f \ } & \quad \text{output function} \\
\uparrow s \theta \ & \quad \text{initial state} \\
\text{rcurrent \ } & \quad \text{returns the currently active domain} \\
\text{rcswitch \ } & \quad \text{switches the currently active domain} \\
\text{(\(=\) \(42\) \ } & \quad \text{interupt function (yet unspecified)} \\
\text{rkinvolved \ } & \quad \text{returns a set of threads involved in the give action} \\
\text{rifp \ } & \quad \text{information flow policy} \\
\text{rvpeq \ } & \quad \text{view partitioning} \\
\text{rAS-set \ } & \quad \text{the set of valid action sequences} \\
\text{rinvariant \ } & \quad \text{the state invariant} \\
\text{rprecondition \ } & \quad \text{the precondition for doing an action} \\
\text{raborting \ } & \quad \text{condition under which an action is aborted} \\
\text{rwaiting \ } & \quad \text{condition under which an action is delayed} \\
\text{rset-error-code \ } & \quad \text{updates the state. Has no meaning in the current model.} \\
\end{align*}

\textbf{using CISK-proof-obligations-satisfied by blast}

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

\textbf{theorem \ \text{risecure:}}

\begin{align*}
\text{Inst.isecure} \\
\text{using \ Inst.unwinding-implies-secure-CISK} \\
\text{by blast} \\
\end{align*}

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 \sim 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-secure has been applied to, e.g., smartcards [27] and OS kernel extensions [7]. To the best of our knowledge, Rushby’s definition has not been applied in a certification context. Rushby’s definition has been subject to heavy scrutiny [22], [29] and a vast array of modifications have been proposed.

Roscoe and Goldsmith provide CSP-based definitions of NI for the transitive and the intransitive case, here dubbed as lazy and mixed independence. The latter one is more restrictive than Rushby’s IP-secure. Their critique on IP-secure, however, is not universally accepted [7]. Greve at al. provided the GWV framework developed in ACL2 [9]. Their definition is a non-inductive version of noninterference similar to Rushby’s step consistency. GWV has been used on various industrial systems. The exact relation between GWV and (I)P-secure, i.e., whether they are of equal strength, is still open. The second property, Declassification, means whether the definition allows assignments in the form of \( l := \text{declassify}(h) \) (where we use 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. Eggett et al. defined i-secure, an extension of IP-secure. Their model extends Rushby’s model (Mealy machines) with Local security policies. Murray et al. extends Von Oheimb definition of noninfluence to apply to a model that does not assume a static mapping of actions to domains. This makes it applicable to OS’s, as in such a setting such a mapping does not exist [20]. NI-OS has been applied to the seL4 separation kernel [20], [14].

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

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

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

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


