D31.1
Formal Specification of a Generic Separation Kernel

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

Deliverable type:	R
Deliverable reference number:	ICT-318353 / D31.1 / 0.0
Activity and Work package contributing to deliverable:	Activity 3 / WP 3.1
Due date:	September 2013 – M12
Actual submission date:	17th March, 2025

Responsible organisation:	Open University of The Netherlands
Editors:	Freek Verbeek, Julien Schmaltz
Dissemination level:	PU
Revision:	0.0 (r-2)

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

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

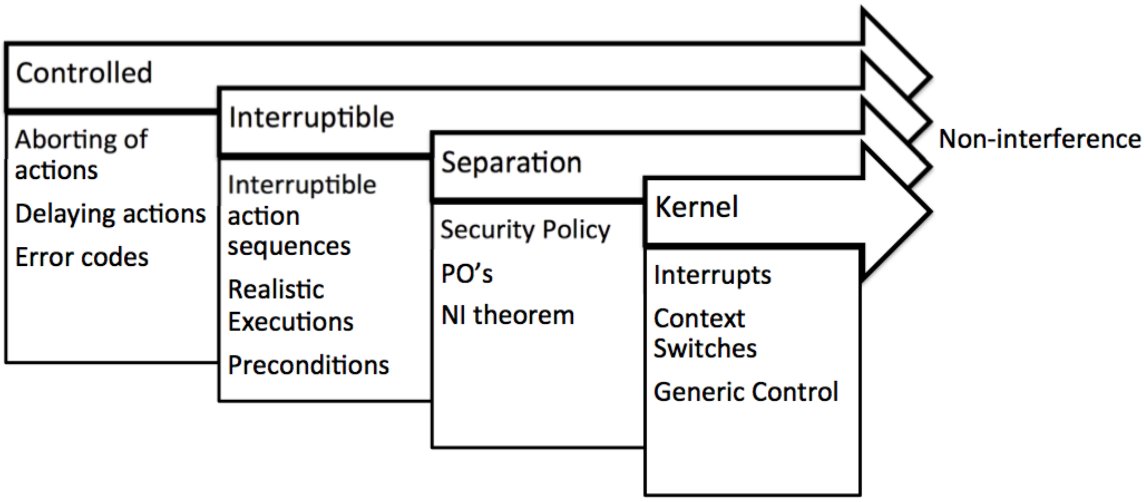

Figure 1: Overview of CISK modular structure

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

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 s1 t1*))
⟨*proof*⟩
lemma *rewrite-B-None*[*simp*]:
shows *None* ⇀ *α* = *True*
⟨*proof*⟩
lemma *rewrite-B-m-True*[*simp*]:
shows *m* ⇀ (λ *a* . *True*) = *True*
⟨*proof*⟩
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*)
⟨*proof*⟩

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* ⟨*proof*⟩
lemma *length-ge-one*:
shows *x* ≠ [] ⟶ *length x* ≥ *1* ⟨*proof*⟩
lemma *take-but-one*[*simp*]:
shows *x* ≠ [] ⟶ *lastn* ((*length x*) − *1*) *x* = *tl x* ⟨*proof*⟩
lemma *Suc-m-minus-n*[*simp*]:
shows *m* ≥ *n* ⟶ *Suc m* − *n* = *Suc* (*m* − *n*) ⟨*proof*⟩
lemma *lastn-one-less*:

shows $n > 0 \wedge n \leq length\ x \wedge lastn\ n\ x = (a\#y) \longrightarrow lastn\ (n - 1)\ x = y$ ⟨*proof*⟩
lemma *list-sub-implies-member*:
shows $\forall\ a\ x\ .\ set\ (a\#x) \subseteq Z \longrightarrow a \in Z$ ⟨*proof*⟩
lemma *subset-smaller-list*:
shows $\forall\ a\ x\ .\ set\ (a\#x) \subseteq Z \longrightarrow set\ x \subseteq Z$ ⟨*proof*⟩
lemma *second-elt-is-hd-tl*:
shows $tl\ x = (a\ \#\ x') \longrightarrow a = x\ !\ 1$
⟨*proof*⟩
lemma *length-ge-2-implies-tl-not-empty*:
shows $length\ x \geq 2 \longrightarrow tl\ x \neq []$
⟨*proof*⟩
lemma *length-lt-2-implies-tl-empty*:
shows $length\ x < 2 \longrightarrow tl\ x = []$
⟨*proof*⟩
lemma *first-second-is-sub-seq*:
shows $length\ x \geq 2 \Longrightarrow is\text{-}sub\text{-}seq\ (hd\ x)\ (x!1)\ x$
⟨*proof*⟩
lemma *hd-drop-is-nth*:
shows $n < length\ x \Longrightarrow hd\ (drop\ n\ x) = x!n$
⟨*proof*⟩

lemma *def-of-hd*:
shows $y = a\ \#\ x \longrightarrow hd\ y = a$ ⟨*proof*⟩
lemma *def-of-tl*:
shows $y = a\ \#\ x \longrightarrow tl\ y = x$ ⟨*proof*⟩
lemma *drop-yields-results-implies-nbound*:
shows $drop\ n\ x \neq [] \longrightarrow n < length\ x$
⟨*proof*⟩
lemma *consecutive-is-sub-seq*:
shows $a\ \#\ (b\ \#\ x) = lastn\ n\ y \Longrightarrow is\text{-}sub\text{-}seq\ a\ b\ y$
⟨*proof*⟩

lemma *sub-seq-in-prefixes*:
assumes $\exists\, y \in prefixes\ X.\ is\text{-}sub\text{-}seq\ a\ a'\ y$
shows $\exists\, y \in X.\ is\text{-}sub\text{-}seq\ a\ a'\ y$
⟨*proof*⟩

lemma *set-tl-is-subset*:
shows $set\ (tl\ x) \subseteq set\ x$ ⟨*proof*⟩
lemma *x-is-hd-snd-tl*:
shows $length\ x \geq 2 \longrightarrow x = (hd\ x)\ \#\ x!1\ \#\ tl(tl\ x)$
⟨*proof*⟩

lemma *tl-x-not-x*:
shows $x \neq [] \longrightarrow tl\ x \neq x$ ⟨*proof*⟩
lemma *tl-hd-x-not-tl-x*:
shows $x \neq [] \wedge hd\ x \neq [] \longrightarrow tl\ (hd\ x)\ \#\ tl\ x \neq x$ ⟨*proof*⟩

end

3 A generic model for separation kernels

theory *K*
imports *List-Theorems Option-Binders*
begin

This section defines a detailed generic model of separation kernels called CISK (Controlled Interruptible Separation Kernel). It contains a generic functional model of the behaviour of a separation kernel as a transition system, definitions of the security property and proofs that the functional model satisfies security properties. It is based on Rushby's approach [25] for noninterference. For an explanation of the model, its structure and an overview of the proofs, we refer to the document entitled "A New Theory of Intransitive Noninterference for Separation Kernels with Control" [31].

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

- *locale "Kernel" defines a highly generic model for a kernel, with execution semantics. It defines a state transition system with some extensions to the one used in [25]. The transition system defined here stores the currently active domain in the state, and has transitions for explicit context switches and interrupts and provides a notion of control. As each operation of the system will be split into atomic actions in our model, only certain sequences of actions will correspond to a run on a real system. Therefore, the function* run*, which applies an execution on a state and computes the resulting new state, is partial and defined for realistic traces only. Later, but not in this locale, we will define a predicate to distinguish realistic traces from other traces. Security properties are also not part of this locale, but will be introduced in the locales to be described next.*
- *locale "Separation_Kernel" extends "Kernel" with constraints concerning non-interference. The theorem is only sensical for realistic traces; for unrealistic trace it will hold vacuously.*
- *locale "Interruptible_Separation_Kernel" refines "Separation_Kernel" with interruptible action sequences. It defines function "realistic_trace" based on these action sequences. Therefore, we can formulate a total run function.*
- *locale "Controlled_Interruptible_Separation_Kernel" refines "Interruptible_Separation_Kernel" with abortable action sequences. It refines function "control" which now uses a generic predicate "aborting" and a generic function "set_error_code" to manage aborting of action sequences.*

3.1 K (Kernel)

The model makes use of the following types:

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

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

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

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

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

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

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

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

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

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

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

3.1.1 Execution semantics

Short hand notations for using function control.

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

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

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

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

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

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

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

function *run* :: *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-execs s execs*)
⟨*proof*⟩
termination ⟨*proof*⟩
end

end

3.2 SK (Separation Kernel)

theory *SK*
imports *K*
begin

Locale Kernel is now refined to a generic model of a separation kernel. The security policy is represented using function *ia*. Function *vpeq* is adopted from Rushby and is an equivalence relation represeting 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 t iff an interrupt occurs in the given state at the given time
and *kprecondition* :: *'state-t* ⇒ *'action-t* ⇒ *bool* — Returns t if an precondition holds that relates the current action to the state
and *realistic-execution* :: *'action-t execution* ⇒ *bool* — In this locale, this function is completely unconstrained.
and *control* :: *'state-t* ⇒ *'dom-t* ⇒ *'action-t execution* ⇒ ((*'action-t option*) × *'action-t execution* × *'state-t*)
and *kinvolved* :: *'action-t* ⇒ *'dom-t set*
+
fixes *ifp* :: *'dom-t* ⇒ *'dom-t* ⇒ *bool*
and *vpeq* :: *'dom-t* ⇒ *'state-t* ⇒ *'state-t* ⇒ *bool*
assumes *vpeq-transitive*: ∀ *a b c u*. (*vpeq u a b* ∧ *vpeq u b c*) ⟶ *vpeq u a c*
and *vpeq-symmetric*: ∀ *a b u*. *vpeq u a b* ⟶ *vpeq u b a*
and *vpeq-reflexive*: ∀ *a u*. *vpeq u a a*
and *ifp-reflexive*: ∀ *u* . *ifp u u*
and *weakly-step-consistent*: ∀ *s t u a*. *vpeq u s t* ∧ *vpeq* (*current s*) *s t* ∧ *kprecondition s a* ∧ *kprecondition t a* ∧ *current s* = *current t* ⟶ *vpeq u* (*kstep s a*) (*kstep t a*)
and *locally-respects*: ∀ *a s 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* . *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-execs-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* (*current s*) (*execs* (*current s*)))) = *fst* (*snd* (*control t* (*current s*) (*execs* (*current s*))))
and *next-state-consistent*: ∀ *s t u execs* . *vpeq* (*current s*) *s t* ∧ *vpeq u s t* ∧ *current s* = *current t* ⟶ *vpeq u* (*next-state s execs*) (*next-state t execs*)
and *current-next-state*: ∀ *s execs* . *current* (*next-state s execs*) = *current s*
and *locally-respects-next-state*: ∀ *s u execs*. ¬*ifp* (*current s*) *u* ⟶ *vpeq u s* (*next-state s execs*)
and *involved-ifp*: ∀ *s a* . ∀ *d* ∈ (*involved a*) . *kprecondition s* (*the a*) ⟶ *ifp d* (*current s*)
and *next-action-from-execs*: ∀ *s execs* . *next-action s execs* ⇀ (λ *a* . *a* ∈ *actions-in-execution* (*execs* (*current s*)))
and *next-execs-subset*: ∀ *s execs u* . *actions-in-execution* (*next-execs s execs u*) ⊆ *actions-in-execution* (*execs u*)
begin

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

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

abbreviation *equivalent-states* :: *'state-t option* ⇒ *'state-t option* ⇒ *'dom-t* ⇒ *bool*
where *equivalent-states s t u* ≡ *s* ∥ *t* ⇀ (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*::*bool* **where** *view-partitioned*
≡ ∀ *execs ms mt n u* . *equivalent-states ms mt u* ⟶
(*run n ms execs* ∥
run n mt (*purge execs u*) ⇀
(λ *rs rt* . *vpeq u rs rt* ∧ *current rs* = *current rt*))

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

definition *purged-relation* :: *'dom-t* ⇒ (*'dom-t* ⇒ *'action-t execution*) ⇒ (*'dom-t* ⇒ *'action-t execution*) ⇒ *bool*
where *purged-relation u execs1 execs2* ≡ ∀ *d* . *ifp^∗∗ d u* ⟶ *execs1 d* = *execs2 d*

The inductive version of view partitioning says that runs on two states that are u-equivalent and on two executions that are purged_related yield u-equivalent states.

definition *view-partitioned-ind*::*bool* **where** *view-partitioned-ind*
≡ ∀ *execs1 execs2 s t n u* . *equivalent-states s t u* ∧ *purged-relation u execs1 execs2* ⟶ *equivalent-states* (*run n s execs1*) (*run n t execs2*) *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*⟩

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

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

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

lemma *vpeq-ns-nt-ifp-u*:
assumes *vpeq-s-t*: ∀ *v* . *ifp*^** *v u* ⟶ *vpeq v s t′*
and *current-s-t*: *current s* = *current t′*
shows *precondition* (*next-state s execs*) *a* ∧ *precondition* (*next-state t′ execs*) *a* ⟹ (*ifp*^** (*current s*) *u* ⟹ (∀ *v* . *ifp*^** *v u* ⟶ *vpeq v* (*step* (*next-state s execs*) *a*) (*step* (*next-state t′ execs*) *a*)))
⟨*proof*⟩

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

lemma *vpeq-ns-nt-not-ifp-u*:
assumes *purged-a-a2*: *purged-relation u execs execs2*
and *prec-s*: *precondition* (*next-state s execs*) (*next-action s execs*)
and *current-s-t*: *current s* = *current t′*
and *vpeq-s-t*: ∀ *v* . *ifp*^** *v u* ⟶ *vpeq v s t′*
shows ¬*ifp*^** (*current s*) *u* ∧ *precondition* (*next-state t′ execs2*) (*next-action t′ execs2*) ⟶ (∀ *v* . *ifp*^** *v u* ⟶ *vpeq v* (*step* (*next-state s execs*) (*next-action s execs*)) (*step* (*next-state t′ execs2*) (*next-action t′ execs2*)))
⟨*proof*⟩

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

lemma *unwinding-implies-view-partitioned-ind*:
shows *view-partitioned-ind*
⟨*proof*⟩

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

lemma *unwinding-implies-view-partitioned*:
shows *view-partitioned*
⟨*proof*⟩

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

theorem *unwinding-implies-NI-unrelated*:
shows *NI-unrelated*
⟨*proof*⟩

3.2.2 Security for indirectly interfering domains

Consider the following security policy over three domains A, B and C: $A \rightsquigarrow B \rightsquigarrow C$, but $A \not\rightsquigarrow C$. 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* ≡ ∃ *v* . *ifp^** v d* ∧ *ifp d u* ∧ ¬*ifp v u* ∧ *d* ≠ *u*
primrec *remove-gateway-communications* :: *'dom-t* ⇒ *'action-t execution* ⇒ *'action-t execution*
where *remove-gateway-communications u* [] = []
| *remove-gateway-communications u* (*aseq#exec*) = (*if* ∃ *a* ∈ *set aseq* . ∃ *v* . *intermediary v u* ∧ *v* ∈ *involved* (*Some a*) *then* [] *else aseq*)#(*remove-gateway-communications u exec*)

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

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

abbreviation *ind-source* :: *'dom-t* ⇒ *'dom-t* ⇒ *bool*
where *ind-source d u* ≡ *ifp^** d u* ∧ ¬*ifp d u*
definition *ipurge-r* ::
(*'dom-t* ⇒ *'action-t execution*) ⇒ *'dom-t* ⇒ (*'dom-t* ⇒ *'action-t execution*) **where**
ipurge-r execs u ≡ λ *d* . *if intermediary d u then*
[]
else if ind-source d u then
SOME alpha . *realistic-execution alpha*
else if d = *u then*
remove-gateway-communications u (*execs u*)
else
execs d

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

definition *NI-indirect-sources* ::*bool*
where *NI-indirect-sources*
≡ ∀ *execs a n*. *run n* (*Some s0*) *execs* ⇀
(λ *s-f* . (*run n* (*Some s0*) (*ipurge-l execs* (*current s-f*)) ‖
run n (*Some s0*) (*ipurge-r execs* (*current s-f*)) ⇀
(λ *s-l s-r* . *output-f s-l a* = *output-f s-r a*)))

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

This allows us to define security over intransitive policies.

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

abbreviation *iequivalent-states* :: *'state-t option* ⇒ *'state-t option* ⇒ *'dom-t* ⇒ *bool*
where *iequivalent-states s t u* ≡ *s* ‖ *t* ⇀ (λ *s t* . (∀ *v* . *ifp v u* ∧ ¬*intermediary v u* ⟶ *vpeq v s t*) ∧ *current s* = *current t*)

definition *does-not-communicate-with-gateway*
where *does-not-communicate-with-gateway u execs* ≡ ∀ *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* = *execs2 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*⟩

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*⟩

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

lemma *iunwinding-implies-view-partitioned1*:
shows *iview-partitioned*
⟨*proof*⟩

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

definition *mcurrents* :: *'state-t option* ⇒ *'state-t option* ⇒ *bool*
where *mcurrents m1 m2* ≡ *m1* ∥ *m2* ⇀ (λ *s t* . *current s* = *current t*)

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

lemma *current-independent-of-domain-actions*:
assumes *current-s-t*: *mcurrents s t*
shows *mcurrents* (*run n s execs*) (*run n t execs2*)
⟨*proof*⟩

theorem *unwinding-implies-NI-indirect-sources*:
shows *NI-indirect-sources*
⟨*proof*⟩

theorem *unwinding-implies-isecure*:

shows *isecure*
⟨*proof*⟩

end
end

3.3 ISK (Interruptible Separation Kernel)

theory *ISK*
imports *SK*
begin

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

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

$$\gamma = [COPY\_INIT, COPY\_CHECK, COPY\_COPY]$$

If action sequence γ 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 γ, 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 iff an interrupt occurs in the given state at the given time
and *kprecondition* :: *'state-t* ⇒ *'action-t* ⇒ *bool* — Returns t if an precondition holds that relates the current action to the state
and *realistic-execution* :: *'action-t execution* ⇒ *bool* — In this locale, this function is completely unconstrained.
and *control* :: *'state-t* ⇒ *'dom-t* ⇒ *'action-t execution* ⇒ ((*'action-t option*) × *'action-t execution* × *'state-t*)
and *kinvolved* :: *'action-t* ⇒ *'dom-t set*
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-prec-first-action*: ∀ *s d aseq* . *invariant s* ∧ *aseq* ∈ *AS-set* ∧ *aseq* ≠ [] ⟶ *AS-precondition s d* (*hd aseq*)
and *AS-prec-after-step*: ∀ *s a a'* . (∃ *aseq* ∈ *AS-set* . *is-sub-seq a a' aseq*) ∧ *invariant s* ∧ *AS-precondition s* (*current s*) *a* ∧ ¬*aborting s* (*current s*) *a* ∧ ¬ *waiting s* (*current s*) *a* ⟶ *AS-precondition* (*kstep s a*) (*current s*) *a'*
and *AS-prec-dom-independent*: ∀ *s d a a'* . *current s* ≠ *d* ∧ *AS-precondition s d a* ⟶ *AS-precondition* (*kstep s a'*) *d a*
and *spec-of-invariant*: ∀ *s a* . *invariant s* ⟶ *invariant* (*kstep s a*)

and *kprecondition-def*: *kprecondition s a* ≡ *invariant s* ∧ *AS-precondition s* (*current s*) *a*
and *realistic-execution-def*: *realistic-execution aseq* ≡ *set aseq* ⊆ *AS-set*
and *control-spec*: ∀ *s d aseqs* . *case control s d aseqs of* (*a*,*aseqs'*,*s'*) ⇒
(*thread-empty aseqs* ∧ (*a*,*aseqs'*) = (*None*,[])) ∨ — Nothing happens
(*aseqs* ≠ [] ∧ *hd aseqs* ≠ [] ∧ ¬*aborting s' d* (*the a*) ∧ ¬ *waiting s' d* (*the a*) ∧ (*a*,*aseqs'*) = (*Some* (*hd* (*hd aseqs*)), (*tl* (*hd aseqs*))#(*tl aseqs*))) ∨ — Execute the first action of the current action sequence
(*aseqs* ≠ [] ∧ *hd aseqs* ≠ [] ∧ *waiting s' d* (*the a*) ∧ (*a*,*aseqs'*,*s'*) = (*Some* (*hd* (*hd aseqs*)),*aseqs*,*s*)) ∨ — Nothing happens, waiting to execute the next action
(*a*,*aseqs'*) = (*None*,*tl aseqs*)
and *next-action-after-cswitch*: ∀ *s n d aseqs* . *fst* (*control* (*cswitch n s*) *d aseqs*) = *fst* (*control s d aseqs*)
and *next-action-after-next-state*: ∀ *s execs d* . *current s* ≠ *d* ⟶ *fst* (*control* (*next-state s execs*) *d* (*execs d*)) = *None* ∨ *fst* (*control* (*next-state s execs*) *d* (*execs d*)) = *fst* (*control s d* (*execs d*))
and *next-action-after-step*: ∀ *s a d aseqs* . *current s* ≠ *d* ⟶ *fst* (*control* (*step s a*) *d aseqs*) = *fst* (*control s d aseqs*)
and *next-state-precondition*: ∀ *s d a execs*. *AS-precondition s d a* ⟶ *AS-precondition* (*next-state s execs*) *d a*
and *next-state-invariant*: ∀ *s execs* . *invariant s* ⟶ *invariant* (*next-state s execs*)
and *spec-of-waiting*: ∀ *s a* . *waiting s* (*current s*) *a* ⟶ *kstep s a* = *s*
begin

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

function *run-total* :: *time-t* ⇒ *'state-t* ⇒ (*'dom-t* ⇒ *'action-t execution*) ⇒ *'state-t*
where *run-total 0 s execs* = *s*
| *interrupt* (*Suc n*) ⟹ *run-total* (*Suc n*) *s execs* = *run-total n* (*cswitch* (*Suc n*) *s*) *execs*
| ¬*interrupt* (*Suc n*) ⟹ *thread-empty*(*execs* (*current s*)) ⟹ *run-total* (*Suc n*) *s execs* = *run-total n s execs*

| ¬*interrupt* (*Suc n*) ⟹ ¬*thread-empty*(*execs* (*current s*)) ⟹
run-total (*Suc n*) *s execs* = *run-total n* (*step* (*next-state s execs*) (*next-action s execs*)) (*next-execs s execs*)
⟨*proof*⟩
termination ⟨*proof*⟩

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

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

Lemma run_total_equals_run is proven by doing induction. It is however not inductive and can therefore not be proven directly: a realistic execution is not necessarily realistic after performing one action. We generalize to do induction. Predicate realistic_executions_ind is the inductive version of realistic_executions. All action sequences in the tail of the executions must be complete action sequences (i.e., they must be from AS_set). The first action sequence, however, is being executed and is therefore not necessarily an action sequence from AS_set, but it is *the last part* of some action sequence from AS_set.

definition *realistic-AS-partial* :: ′*action-t list* ⇒ *bool*
where *realistic-AS-partial aseq* ≡ ∃ *n aseq*′ . *n* ≤ *length aseq*′ ∧ *aseq*′ ∈ *AS-set* ∧ *aseq* = *lastn n aseq*′
definition *realistic-executions-ind* :: (′*dom-t* ⇒ ′*action-t execution*) ⇒ *bool*
where *realistic-executions-ind execs* ≡ ∀ *d* . (*case execs d of* [] ⇒ *True* | (*aseq*#*aseqs*) ⇒ *realistic-AS-partial aseq* ∧ *set aseqs* ⊆ *AS-set*)

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

definition *precondition-ind* :: ′*state-t* ⇒ (′*dom-t* ⇒ ′*action-t execution*) ⇒ *bool*
where *precondition-ind s execs* ≡ *invariant s* ∧ (∀ *d* . *fst*(*control s d* (*execs d*)) ⇀ *AS-precondition s d*)

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

lemma *next-execution-is-realistic-partial*:
assumes *na-def*: *next-execs s execs d* = *aseq* # *aseqs*
and *d-is-curr*: *d* = *current s*
and *realistic*: *realistic-executions-ind execs*
and *thread-not-empty*: ¬*thread-empty*(*execs* (*current s*))
shows *realistic-AS-partial aseq* ∧ *set aseqs* ⊆ *AS-set*
⟨*proof*⟩

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*⟩

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

lemma *realistic-purge*:
shows ∀ *execs d* . *realistic-executions execs* ⟶ *realistic-executions* (*purge execs d*)
⟨*proof*⟩

lemma *remove-gateway-comm-subset*:
shows *set* (*remove-gateway-communications d exec*) ⊆ *set exec* ∪ {[]}
⟨*proof*⟩

lemma *realistic-ipurge-l*:
shows ∀ *execs d* . *realistic-executions execs* ⟶ *realistic-executions* (*ipurge-l execs d*)
⟨*proof*⟩

lemma *realistic-ipurge-r*:
shows ∀ *execs d* . *realistic-executions execs* ⟶ *realistic-executions* (*ipurge-r execs d*)
⟨*proof*⟩

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

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

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

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

theorem *unwinding-implies-isecure-total*:
shows *isecure-total*
⟨*proof*⟩

end
end

3.4 CISK (Controlled Interruptible Separation Kernel)

theory *CISK*
imports *ISK*
begin

This section presents a generic model of a Controlled Interruptible Separation Kernel (CISK). It formulates security, i.e., intransitive noninterference. For a presentation of this model, see Section 2 of [31].

First, a locale is defined that defines all generic functions and all proof obligations (see Section 2.3 of [31]).

locale *Controllable-Interruptible-Separation-Kernel* = — CISK
fixes *kstep* :: *'state-t* ⇒ *'action-t* ⇒ *'state-t* — Executes one atomic kernel action
and *output-f* :: *'state-t* ⇒ *'action-t* ⇒ *'output-t* — Returns the observable behavior
and *s0* :: *'state-t* — The initial state
and *current* :: *'state-t* => *'dom-t* — Returns the currently active domain

and *cswitch* :: *time-t* ⇒ *'state-t* ⇒ *'state-t* — Performs a context switch
and *interrupt* :: *time-t* ⇒ *bool* — Returns t 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* ⇒ *'dom-t* ⇒ *bool* — The security policy.
and *vpeq* :: *'dom-t* ⇒ *'state-t* ⇒ *'state-t* ⇒ *bool* — View partitioning equivalence
and *AS-set* :: (*'action-t list*) *set* — Returns a set of valid action sequences, i.e., the attack surface
and *invariant* :: *'state-t* ⇒ *bool* — Returns an inductive state-invariant
and *AS-precondition* :: *'state-t* ⇒ *'dom-t* ⇒ *'action-t* ⇒ *bool* — Returns the preconditions under which the given action can be executed.
and *aborting* :: *'state-t* ⇒ *'dom-t* ⇒ *'action-t* ⇒ *bool* — Returns true iff the action is aborted.
and *waiting* :: *'state-t* ⇒ *'dom-t* ⇒ *'action-t* ⇒ *bool* — Returns true iff execution of the given action is delayed.
and *set-error-code* :: *'state-t* ⇒ *'action-t* ⇒ *'state-t* — Sets an error code when actions are aborted.
assumes *vpeq-transitive*: ∀ *a b c u*. (*vpeq u a b* ∧ *vpeq u b c*) ⟶ *vpeq u a c*
and *vpeq-symmetric*: ∀ *a b u*. *vpeq u a b* ⟶ *vpeq u b a*
and *vpeq-reflexive*: ∀ *a u*. *vpeq u a a*
and *ifp-reflexive*: ∀ *u* . *ifp u u*
and *weakly-step-consistent*: ∀ *s t u a*. *vpeq u s t* ∧ *vpeq* (*current s*) *s t* ∧ *invariant s* ∧ *AS-precondition s* (*current s*) *a* ∧ *invariant t* ∧ *AS-precondition t* (*current t*) *a* ∧ *current s* = *current t* ⟶ *vpeq u* (*kstep s a*) (*kstep t a*)
and *locally-respects*: ∀ *a s u*. ¬*ifp* (*current s*) *u* ∧ *invariant s* ∧ *AS-precondition s* (*current s*) *a* ⟶ *vpeq u s* (*kstep s a*)
and *output-consistent*: ∀ *a s t*. *vpeq* (*current s*) *s t* ∧ *current s* = *current t* ⟶ (*output-f s a*) = (*output-f t a*)
and *step-atomicity*: ∀ *s a* . *current* (*kstep s a*) = *current s*
and *cswitch-independent-of-state*: ∀ *n s t* . *current s* = *current t* ⟶ *current* (*cswitch n s*) = *current* (*cswitch n t*)
and *cswitch-consistency*: ∀ *u s t n* . *vpeq u s t* ⟶ *vpeq u* (*cswitch n s*) (*cswitch n t*)
and *empty-in-AS-set*: [] ∈ *AS-set*
and *invariant-s0*: *invariant s0*
and *invariant-after-cswitch*: ∀ *s n* . *invariant s* ⟶ *invariant* (*cswitch n s*)
and *precondition-after-cswitch*: ∀ *s d n a*. *AS-precondition s d a* ⟶ *AS-precondition* (*cswitch n s*) *d a*
and *AS-prec-first-action*: ∀ *s d aseq* . *invariant s* ∧ *aseq* ∈ *AS-set* ∧ *aseq* ≠ [] ⟶ *AS-precondition s d* (*hd aseq*)
and *AS-prec-after-step*: ∀ *s a a'* . (∃ *aseq* ∈ *AS-set* . *is-sub-seq a a' aseq*) ∧ *invariant s* ∧ *AS-precondition s* (*current s*) *a* ∧ ¬*aborting s* (*current s*) *a* ∧ ¬ *waiting s* (*current s*) *a* ⟶ *AS-precondition* (*kstep s a*) (*current s*) *a'*
and *AS-prec-dom-independent*: ∀ *s d a a'* . *current s* ≠ *d* ∧ *AS-precondition s d a* ⟶ *AS-precondition* (*kstep s a'*) *d a*
and *spec-of-invariant*: ∀ *s a* . *invariant s* ⟶ *invariant* (*kstep s a*)
and *aborting-switch-independent*: ∀ *n s* . *aborting* (*cswitch n s*) = *aborting s*
and *aborting-error-update*: ∀ *s d a' a* . *current s* ≠ *d* ∧ *aborting s d a* ⟶ *aborting* (*set-error-code s a'*) *d a*
and *aborting-after-step*: ∀ *s a d* . *current s* ≠ *d* ⟶ *aborting* (*kstep s a*) *d* = *aborting s d*
and *aborting-consistent*: ∀ *s t u* . *vpeq u s t* ⟶ *aborting s u* = *aborting t u*
and *waiting-switch-independent*: ∀ *n s* . *waiting* (*cswitch n s*) = *waiting s*
and *waiting-error-update*: ∀ *s d a' a* . *current s* ≠ *d* ∧ *waiting s d a* ⟶ *waiting* (*set-error-code s a'*) *d a*
and *waiting-consistent*: ∀ *s t u a* . *vpeq* (*current s*) *s t* ∧ (∀ *d* ∈ *kinvolved a* . *vpeq d s t*) ∧ *vpeq u s t* ⟶ *waiting s u a* = *waiting t u a*
and *spec-of-waiting*: ∀ *s a* . *waiting s* (*current s*) *a* ⟶ *kstep s a* = *s*
and *set-error-consistent*: ∀ *s t u a* . *vpeq u s t* ⟶ *vpeq u* (*set-error-code s a*) (*set-error-code t a*)
and *set-error-locally-respects*: ∀ *s u a* . ¬*ifp* (*current s*) *u* ⟶ *vpeq u s* (*set-error-code s a*)
and *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*: ∀ *s a* . *invariant s* ⟶ *invariant* (*set-error-code s a*)
and *involved-ifp*: ∀ *s a* . ∀ *d* ∈ (*kinvolved a*) . *AS-precondition s* (*current s*) *a* ⟶ *ifp d* (*current s*)
begin

3.4.1 Execution semantics

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

function *CISK-control* :: *'state-t* ⇒ *'dom-t* ⇒ *'action-t execution* ⇒ (*'action-t option* × *'action-t execution* × *'state-t*)
where *CISK-control s d* [] = (*None*,[],*s*) — The thread is empty
| *CISK-control s d* ([]#[]) = (*None*,[],*s*) — The current action sequence has been finished and the thread has no next action sequences to execute
| *CISK-control s d* ([]#(*as'*#*execs'*)) = (*None*,*as'*#*execs'*,*s*) — The current action sequence has been finished. Skip to the next sequence
| *CISK-control s d* ((*a*#*as*)#*execs'*) = (*if aborting s d a then*
(*None*, *execs'*,*set-error-code s a*)
else if waiting s d a then
(*Some a*, (*a*#*as*)#*execs'*,*s*)
else
(*Some a*, *as*#*execs'*,*s*)) — Executing an action sequence
⟨*proof*⟩
termination ⟨*proof*⟩

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

abbreviation *next-action*::*'state-t* ⇒ (*'dom-t* ⇒ *'action-t execution*) ⇒ *'action-t option*
where *next-action* ≡ *Kernel.next-action current CISK-control*
abbreviation *next-execs*::*'state-t* ⇒ (*'dom-t* ⇒ *'action-t execution*) ⇒ (*'dom-t* ⇒ *'action-t execution*)
where *next-execs* ≡ *Kernel.next-execs current CISK-control*
abbreviation *next-state*::*'state-t* ⇒ (*'dom-t* ⇒ *'action-t execution*) ⇒ *'state-t*
where *next-state* ≡ *Kernel.next-state current CISK-control*

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

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

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

function *run* :: *time-t* ⇒ *'state-t* ⇒ (*'dom-t* ⇒ *'action-t execution*) ⇒ *'state-t*
where *run 0 s execs* = *s*
| *interrupt* (*Suc n*) ⟹ *run* (*Suc n*) *s execs* = *run n* (*cswitch* (*Suc n*) *s*) *execs*
| ¬*interrupt* (*Suc n*) ⟹ *thread-empty*(*execs* (*current s*)) ⟹ *run* (*Suc n*) *s execs* = *run n s execs*
| ¬*interrupt* (*Suc n*) ⟹ ¬*thread-empty*(*execs* (*current s*)) ⟹
run (*Suc n*) *s execs* = (*let control-a* = *next-action s execs*;
control-s = *next-state s execs*;
control-x = *next-execs s execs in*
case control-a of None ⇒ *run n control-s control-x*
| (*Some a*) ⇒ *run n* (*kstep control-s a*) *control-x*)
⟨*proof*⟩
termination ⟨*proof*⟩

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 kinvolved*
abbreviation *step* **where** *step ≡ Kernel.step kstep*
abbreviation *purge* **where** *purge ≡ Separation-Kernel.purge realistic-execution ifp*
abbreviation *ipurge-l* **where** *ipurge-l ≡ Separation-Kernel.ipurge-l kinvolved ifp*
abbreviation *ipurge-r* **where** *ipurge-r ≡ Separation-Kernel.ipurge-r realistic-execution kinvolved ifp*

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

3.4.3 Proofs

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

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

lemma *next-action-consistent*:
shows ∀ *s t execs . vpeq (current s) s t ∧ (∀ d ∈ involved (next-action s execs) . vpeq d s t) ∧ current s = current t* ⟶ *next-action s execs = next-action t execs*
⟨*proof*⟩

lemma *next-execs-consistent*:
shows ∀ *s t execs . vpeq (current s) s t ∧ (∀ d ∈ involved (next-action s execs) . vpeq d s t) ∧ current s = current t* ⟶ *fst (snd (CISK-control s (current s) (execs (current s)))) = fst (snd (CISK-control t (current s) (execs (current s))))*
⟨*proof*⟩

lemma *next-state-consistent*:
shows ∀ *s t u execs . vpeq (current s) s t ∧ vpeq u s t ∧ current s = current t* ⟶ *vpeq u (next-state s execs) (next-state t execs)*
⟨*proof*⟩

lemma *current-next-state*:
shows ∀ *s execs . current (next-state s execs) = current s*
⟨*proof*⟩

lemma *locally-respects-next-state*:

shows ∀ *s u execs*. ¬*ifp* (*current s*) *u* ⟶ *vpeq u s* (*next-state s execs*)
⟨*proof*⟩

lemma *CISK-control-spec*:
shows ∀ *s d aseqs*.
case CISK-control s d aseqs of
(*a*, *aseqs′*, *s′*) ⇒
thread-empty aseqs ∧ (*a*, *aseqs′*) = (*None*, []) ∨
aseqs ≠ [] ∧ *hd aseqs* ≠ [] ∧ ¬ *aborting s′ d* (*the a*) ∧ ¬ *waiting s′ d* (*the a*) ∧ (*a*, *aseqs′*) = (*Some* (*hd* (*hd aseqs*)), *tl* (*hd aseqs*) # *tl aseqs*) ∨
aseqs ≠ [] ∧ *hd aseqs* ≠ [] ∧ *waiting s′ d* (*the a*) ∧ (*a*, *aseqs′*, *s′*) = (*Some* (*hd* (*hd aseqs*)), *aseqs*, *s*) ∨ (*a*, *aseqs′*) = (*None*, *tl aseqs*)
⟨*proof*⟩

lemma *next-action-after-cswitch*:
shows ∀ *s n d aseqs* . *fst* (*CISK-control* (*cswitch n s*) *d aseqs*) = *fst* (*CISK-control s d aseqs*)
⟨*proof*⟩

lemma *next-action-after-next-state*:
shows ∀ *s execs d* . *current s* ≠ *d* ⟶ *fst* (*CISK-control* (*next-state s execs*) *d* (*execs d*)) = *None* ∨ *fst* (*CISK-control* (*next-state s execs*) *d* (*execs d*)) = *fst* (*CISK-control s d* (*execs d*))
⟨*proof*⟩

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*⟩

lemma *next-state-precondition*:
shows ∀ *s d a execs*. *AS-precondition s d a* ⟶ *AS-precondition* (*next-state s execs*) *d a*
⟨*proof*⟩

lemma *next-state-invariant*:
shows ∀ *s execs*. *invariant s* ⟶ *invariant* (*next-state s execs*)
⟨*proof*⟩

lemma *next-action-from-execs*:
shows ∀ *s execs* . *next-action s execs* ⇀ (λ *a* . *a* ∈ *actions-in-execution* (*execs* (*current s*)))
⟨*proof*⟩

lemma *next-execs-subset*:
shows ∀ *s execs u* . *actions-in-execution* (*next-execs s execs u*) ⊆ *actions-in-execution* (*execs u*)
⟨*proof*⟩

theorem *unwinding-implies-isecure-CISK*:
shows *isecure*
⟨*proof*⟩

end
end

4 Instantiation by a separation kernel with concrete actions

theory *Step-configuration*
imports *Main*
begin

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

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

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

4.1 Model of a separation kernel configuration

4.1.1 Type definitions

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

typedecl *partition-id-t*
typedecl *thread-id-t*

typedecl *page-t* — physical address of a memory page
typedecl *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*

end

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

theory *Step-policies*
imports *Step-configuration*
begin

4.2.1 Specification

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

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

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

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

and *ifp-reflexive*:
∀ *p* . *ifp p p*

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

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

4.2.2 Derivation

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

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

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

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

definition *ifp a b* ≡
sp-spec-subj-subj a b
∨ *sp-spec-subj-subj b a*
∨ (∃ *c y* . *sp-spec-subj-subj a c*
∧ *sp-spec-subj-obj c* (*PAGE y*) *WRITE*
∧ *sp-spec-subj-obj b* (*PAGE y*) *READ*)
∨ (*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 sp-spec-subj-obj sp-spec-subj-subj ifp*
⟨*proof*⟩
end

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

interpretation *Policy*: *abstract-policy-derivation configured-subj-obj*⟨*proof*⟩
interpretation *Policy-properties*: *policy-axioms Policy.sp-spec-subj-obj Policy.sp-spec-subj-subj Policy.ifp*
⟨*proof*⟩

lemma *example-how-to-use-properties-in-proofs*:
shows ∀ *p* . *Policy.ifp p p*
⟨*proof*⟩

end

4.3 Separation kernel state and atomic step function

theory *Step*
imports *Step-policies*
begin

4.3.1 Interrupt points

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

datatype *ipc-direction-t* = *SEND* | *RECV*
datatype *ipc-stage-t* = *PREP* | *WAIT* | *BUF page-t*

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

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

4.3.2 System state

typedecl *obj-t* — value of an object

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

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

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

record *thread-t* =

ev-counter :: *nat* — event counter

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

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

4.3.3 Atomic step

Helper functions Set new value for an object.

definition *set-object-value* :: *obj-id-t* ⇒ *obj-t* ⇒ *state-t* ⇒ *state-t* **where**
set-object-value obj-id val s =
s (| *obj* := *fun-upd* (*obj s*) *obj-id val* |)

Return a representation of the opposite direction of IPC communication.

definition *opposite-ipc-direction* :: *ipc-direction-t* ⇒ *ipc-direction-t* **where**
opposite-ipc-direction dir ≡ *case dir of SEND* ⇒ *RECV* | *RECV* ⇒ *SEND*

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

definition *add-access-right* :: *partition-id-t* => *obj-id-t* => *mode-t* => *state-t* => *state-t* **where**
add-access-right part-id obj-id m s =
s (| *sp-impl-subj-obj* := λ *q q′ q″*. (*part-id* = *q* ∧ *obj-id* = *q′* ∧ *m* = *q″*)
∨ *sp-impl-subj-obj s q q′ q″*|)

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

definition *add-comm-right* :: *partition-id-t* ⇒ *partition-id-t* ⇒ *state-t* ⇒ *state-t* **where**
add-comm-right p p′ s ≡
s (| *sp-impl-subj-subj* := λ *q q′* . (*p* = *q* ∧ *p′* = *q′*) ∨ *sp-impl-subj-subj s q q′* |)

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

1. The model contains the system calls for sending an IPC (SEND) and receiving an IPC (RECV), often implementations have a richer API (e.g. combining SEND and RECV in one invocation).
2. We model only a copying ("BUF") mode, not a memory-mapping mode.
3. The model always copies one page per syscall.

definition *ipc-precondition* :: *thread-id-t* ⇒ *ipc-direction-t* ⇒ *thread-id-t* ⇒ *page-t* ⇒ *state-t* ⇒ *bool* **where**
ipc-precondition tid dir partner page s ≡
let sender = (*case dir of SEND* ⇒ *tid* | *RECV* ⇒ *partner*) *in*
let receiver = (*case dir of SEND* ⇒ *partner* | *RECV* ⇒ *tid*) *in*
let local-access-mode = (*case dir of SEND* ⇒ *READ* | *RECV* ⇒ *WRITE*) *in*
(*sp-impl-subj-subj s* (*partition sender*) (*partition receiver*)
∧ *sp-impl-subj-obj s* (*partition tid*) (*PAGE page*) *local-access-mode*)

definition *atomic-step-ipc* :: *thread-id-t* ⇒ *ipc-direction-t* ⇒ *ipc-stage-t* ⇒ *thread-id-t* ⇒ *page-t* ⇒ *state-t* ⇒ *state-t* **where**
atomic-step-ipc tid dir stage partner page s ≡
case stage of
PREP ⇒
s
| *WAIT* ⇒
s
| *BUF page′* ⇒
(*case dir of*
SEND ⇒
(*set-object-value* (*PAGE page′*) (*obj s* (*PAGE page*)) *s*)
| *RECV* ⇒ *s*)

Model of event syscalls **definition** *ev-signal-precondition* :: *thread-id-t* ⇒ *thread-id-t* ⇒ *state-t* ⇒ *bool* **where**
ev-signal-precondition tid partner s ≡
(*sp-impl-subj-subj s* (*partition tid*) (*partition partner*))

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

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

definition *atomic-step-ev-wait-all* :: *thread-id-t* ⇒ *state-t* ⇒ *state-t* **where**
atomic-step-ev-wait-all tid s =
s (| *thread* := *fun-upd* (*thread s*) *tid* (*thread s tid* (| *ev-counter* := *0* |)) |)

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

definition *aborting* :: *state-t* ⇒ *thread-id-t* ⇒ *int-point-t* ⇒ *bool*
where *aborting s tid a* ≡ *case a of SK-IPC dir PREP partner page* ⇒
¬*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* ≡
(∀ *p1 p2* . *sp-impl-subj-subj s p1 p2* ⟶ *Policy.sp-spec-subj-subj p1 p2*)
∧ (∀ *p1 p2 m*. *sp-impl-subj-obj s p1 p2 m* ⟶ *Policy.sp-spec-subj-obj p1 p2 m*)

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

definition *atomic-step-precondition* :: *state-t* ⇒ *thread-id-t* ⇒ *int-point-t* ⇒ *bool* **where**
atomic-step-precondition s tid ipt ≡
case ipt of
SK-IPC dir WAIT partner page ⇒
— the thread managed it past PREP stage
ipc-precondition tid dir partner page s
| *SK-IPC dir* (*BUF page′*) *partner page* ⇒
— both the calling thread and its communication partner managed it past PREP and WAIT stages
ipc-precondition tid dir partner page s

  ∧ *ipc-precondition partner* (*opposite-ipc-direction dir*) *tid page′ s*
| *SK-EV-SIGNAL EV-SIGNAL-FINISH partner* ⇒
  ev-signal-precondition tid partner s
| - ⇒
  — No precondition for other interrupt points.
  True

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

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

4.4.1 Atomic steps of SK_IPC preserve invariants

lemma *set-object-value-invariant*:
 shows *atomic-step-invariant s* = *atomic-step-invariant* (*set-object-value ob va s*)
⟨*proof*⟩

lemma *set-thread-value-invariant*:
 shows *atomic-step-invariant s* = *atomic-step-invariant* (*s* (| *thread* := *thrst* |))
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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

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* [*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*
⟨*proof*⟩

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*⟩

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

theorem *cswitch-consistency-and-respect*:
fixes *u* :: *partition-id-t*
and *s* :: *state-t*
and *new-current* :: *thread-id-t*
assumes *atomic-step-invariant s*
shows *vpeq u s* (*s* ⦇ *current* := *new-current* ⦈)
⟨*proof*⟩

end

4.6 Atomic step locally respects the information flow policy

theory *Step-vpeq-locally-respects*
imports *Step Step-invariants Step-vpeq*
begin

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

4.6.1 Locally respects of atomic step functions

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

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*⟩

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*⟩

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*⟩

4.6.2 Summary theorems on view-partitioning locally respects

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

theorem *atomic-step-respects-policy*:
assumes *no*: ¬ *Policy.ifp (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*⟩

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 *atomic-step-invariant* holds (see [31]).

4.7.1 Weak step consistency of auxiliary functions

lemma *ipc-precondition-weakly-step-consistent*:
assumes *eq-tid*: *vpeq (partition tid) s1 s2*
and *inv1*: *atomic-step-invariant s1*
and *inv2*: *atomic-step-invariant s2*
shows *ipc-precondition tid dir partner page s1 = ipc-precondition tid dir partner page s2*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*)
 ⟨*proof*⟩

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*)
 ⟨*proof*⟩

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*)
 ⟨*proof*⟩

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

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

definition *extend-subj-subj* :: (*partition-id-t* ⇒ *partition-id-t* ⇒ *bool*) ⇒ *state-t* ⇒ *state-t* **where**

extend-subj-subj f s ≡ *s* ⦇ *sp-impl-subj-subj* := *extend-f f* (*sp-impl-subj-subj s*) ⦈

lemma *extend-subj-subj-consistent*:
fixes *f* :: *partition-id-t* ⇒ *partition-id-t* ⇒ *bool*
assumes *vpeq u s1 s2*
shows *vpeq u* (*extend-subj-subj f s1*) (*extend-subj-subj f s2*)
⟨*proof*⟩

4.7.3 Summary theorems on view-partitioning weak step consistency

The atomic step is weakly step consistent with view partitioning. Here, the "weakness" is that we assume that the two states are vp-equivalent not only w.r.t. the observer domain *u*, but also w.r.t. the caller domain *Step.partition tid*).

theorem *atomic-step-weakly-step-consistent*:
assumes *eq-obs*: *vpeq u s1 s2*
and *eq-act*: *vpeq* (*partition* (*current s1*)) *s1 s2*
and *inv1*: *atomic-step-invariant s1*
and *inv2*: *atomic-step-invariant s2*
and *prec1*: *atomic-step-precondition s1* (*current s1*) *ipt*
and *prec2*: *atomic-step-precondition s2* (*current s2*) *ipt*
and *eq-curr*: *current s1* = *current s2*
shows *vpeq u* (*atomic-step s1 ipt*) (*atomic-step s2 ipt*)
⟨*proof*⟩
end

4.8 Separation kernel model

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

First (Section 4.8.1) we instantiate the CISK generic model. Functions that instantiate a generic function of the CISK model are prefixed with an 'r', 'r' standing for "Rushby';, as CISK is derived originally from a model by Rushby [31]. For example, 'rifp' is the instantiation of the generic 'ifp'.

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

4.8.1 Initial state of separation kernel model

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

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

definition *s0* :: *state-t* **where**
s0 ≡ ⦇ *sp-impl-subj-subj* = *Policy.sp-spec-subj-subj*,
sp-impl-subj-obj = *Policy.sp-spec-subj-obj*,

current = *initial-current*,
obj = *initial-obj*,
thread = λ - . (| *ev-counter* = *0* |)
|)

lemma *initial-invariant*:
shows *atomic-step-invariant s0*
⟨*proof*⟩

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.

typedef (**overloaded**) *rstate-t* = { *s* . *atomic-step-invariant s* }
⟨*proof*⟩

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

lemma *rstate-invariant*:
shows *atomic-step-invariant* (↓*s*)
⟨*proof*⟩

lemma *rstate-down-up*[*simp*]:
shows (↑↓*s*) = *s*
⟨*proof*⟩

lemma *rstate-up-down*[*simp*]:
assumes *atomic-step-invariant s*
shows (↓↑*s*) = *s*
⟨*proof*⟩

A CISK action is identified with an interrupt point.

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

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

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

Each CISK domain is identified with a thread id.

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

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

datatype *visible-obj-t* = *VALUE obj-t* | *EXCEPTION*
type-synonym *routput-t* = *page-t* ⇒ *visible-obj-t*

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

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

definition *rprecondition* :: *rstate-t* ⇒ *rdom-t* ⇒ *raction-t* ⇒ *bool* **where**
rprecondition s d a ≡ *atomic-step-precondition* (↓*s*) *d a*
abbreviation *rinvariant*
where *rinvariant s* ≡ *True* — The invariant is already in the state type.

Translate view-partitioning and interaction-allowed relations.

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

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

Context Switches

definition *rcswitch* :: *nat* ⇒ *rstate-t* ⇒ *rstate-t* **where**
rcswitch n s ≡ ↑((↓*s*) (| *current* := (*SOME t* . *True*) |))

4.8.3 Possible action sequences

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

definition *is-SK-IPC* :: *raction-t list* ⇒ *bool*
where *is-SK-IPC aseq* ≡ ∃ *dir partner page* .
aseq = [*SK-IPC dir PREP partner page*,*SK-IPC dir WAIT partner page*,*SK-IPC dir* (*BUF* (*SOME page′* . *True*)) *partner page*]

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

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

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

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

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

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

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* :: *int-point-t option* ⇒ *rdom-t set*
  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*
    ⟨*proof*⟩

lemma *inst-ifp-refl*:
  shows ∀ *u* . *rifp u u*
⟨*proof*⟩

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

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

lemma *inst-cswitch-independent-of-state*:
  assumes *rcurrent s* = *rcurrent t*
  shows *rcurrent* (*rcswitch n s*) = *rcurrent* (*rcswitch n t*)

⟨*proof*⟩

lemma *inst-cswitch-consistency*:
assumes *rvpeq u s t*
shows *rvpeq u* (*rcswitch n s*) (*rcswitch n t*)
⟨*proof*⟩

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

lemma *prec-first-IPC-action*:
assumes *is-SK-IPC aseq*
shows *rprecondition s d* (*hd aseq*)
⟨*proof*⟩

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

lemma *prec-first-EV-WAIT-action*:
assumes *is-SK-EV-WAIT aseq*
shows *rprecondition s d* (*hd aseq*)
⟨*proof*⟩

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

lemma *prec-first-EV-SIGNAL-action*:
assumes *is-SK-EV-SIGNAL aseq*
shows *rprecondition s d* (*hd aseq*)
⟨*proof*⟩

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

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

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

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

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*⟩

lemma *on-set-object-value*:
shows *sp-impl-subj-subj* (*set-object-value ob val s*) = *sp-impl-subj-subj s*
and *sp-impl-subj-obj* (*set-object-value ob val s*) = *sp-impl-subj-obj s*
⟨*proof*⟩

lemma *prec-IPC-dom-independent*:
assumes *current s* ≠ *d*
and *atomic-step-invariant s*
and *atomic-step-precondition s d a*
shows *atomic-step-precondition* (*atomic-step-ipc* (*current s*) *dir stage partner page s*) *d a*
⟨*proof*⟩

lemma *prec-ev-signal-dom-independent*:
assumes *current s* ≠ *d*
and *atomic-step-invariant s*
and *atomic-step-precondition s d a*
shows *atomic-step-precondition* (*atomic-step-ev-signal* (*current s*) *partner s*) *d a*
⟨*proof*⟩

lemma *prec-ev-wait-one-dom-independent*:
assumes *current s* ≠ *d*
and *atomic-step-invariant s*
and *atomic-step-precondition s d a*
shows *atomic-step-precondition* (*atomic-step-ev-wait-one* (*current s*) *s*) *d a*
⟨*proof*⟩

lemma *prec-ev-wait-all-dom-independent*:
assumes *current s* ≠ *d*
and *atomic-step-invariant s*
and *atomic-step-precondition s d a*
shows *atomic-step-precondition* (*atomic-step-ev-wait-all* (*current s*) *s*) *d a*
⟨*proof*⟩

lemma *prec-dom-independent*:
shows $\forall\ s\ d\ a\ a'.\ rcurrent\ s \neq d \wedge rprecondition\ s\ d\ a \longrightarrow rprecondition\ (rstep\ s\ a')\ d\ a$
⟨*proof*⟩

lemma *ipc-precondition-after-cswitch*[*simp*]:
shows *ipc-precondition d dir partner page* ((↓ *s*)⦇*current* := *new-current*⦈)
= *ipc-precondition d dir partner page* (↓ *s*)
⟨*proof*⟩

lemma *precondition-after-cswitch*:
shows $\forall\ s\ d\ n\ a.\ rprecondition\ s\ d\ a \longrightarrow rprecondition\ (rcswitch\ n\ s)\ d\ a$
⟨*proof*⟩

lemma *aborting-switch-independent*:
shows $\forall\ n\ s.\ raborting\ (rcswitch\ n\ s) = raborting\ s$
⟨*proof*⟩
lemma *waiting-switch-independent*:
shows $\forall\ n\ s.\ rwaiting\ (rcswitch\ n\ s) = rwaiting\ s$
⟨*proof*⟩

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

lemma *waiting-after-IPC-step*:
assumes $d1 \neq d2$
shows *waiting* (*atomic-step-ipc d1 dir stage partner page s*) *d2 a* = *waiting s d2 a*
⟨*proof*⟩

lemma *raborting-consistent*:
shows $\forall$ *s t u*. *rvpeq u s t* $\longrightarrow$ *raborting s u* = *raborting t u*
⟨*proof*⟩

lemma *aborting-dom-independent*:
assumes *rcurrent s* $\neq$ *d*
shows *raborting* (*rstep s a*) *d a′* = *raborting s d a′*
⟨*proof*⟩

lemma *ipc-precondition-of-partner-consistent*:
assumes *vpeq*: $\forall$ *d* $\in$ *rkinvolved* (*SK-IPC dir WAIT partner page*) . *rvpeq d s t*
shows *ipc-precondition partner dir′ u page′* (↓ *s*) = *ipc-precondition partner dir′ u page′* ↓ *t*
⟨*proof*⟩

lemma *ev-signal-precondition-of-partner-consistent*:
assumes *vpeq*: $\forall$ *d* $\in$ *rkinvolved* (*SK-EV-SIGNAL EV-SIGNAL-FINISH partner*) . *rvpeq d s t*
shows *ev-signal-precondition partner u* (↓ *s*) = *ev-signal-precondition partner u* (↓ *t*)
⟨*proof*⟩

lemma *waiting-consistent*:
shows $\forall$ *s t u a* . *rvpeq* (*rcurrent s*) *s t* $\wedge$ ($\forall$ *d* $\in$ *rkinvolved a* . *rvpeq d s t*)
$\wedge$ *rvpeq u s t*
$\longrightarrow$ *rwaiting s u a* = *rwaiting t u a*
⟨*proof*⟩

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*⟩

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*⟩

lemma *involved-ifp*:
shows $\forall$ *s a* . $\forall$ *d* $\in$ *rkinvolved a* . *rprecondition s* (*rcurrent s*) *a* $\longrightarrow$ *rifp d* (*rcurrent s*)
⟨*proof*⟩

lemma *spec-of-waiting-ev*:
shows $\forall$ *s a*. *rwaiting s* (*rcurrent s*) (*SK-EV-WAIT EV-FINISH EV-CONSUME-ALL*)
$\longrightarrow$ *rstep s a* = *s*
⟨*proof*⟩

lemma *spec-of-waiting-ev-w*:
shows $\forall$ *s a*. *rwaiting s* (*rcurrent s*) (*SK-EV-WAIT EV-WAIT EV-CONSUME-ALL*)
$\longrightarrow$ *rstep s* (*SK-EV-WAIT EV-WAIT EV-CONSUME-ALL*) = *s*

⟨*proof*⟩

lemma *spec-of-waiting*:
shows ∀ *s a*. *rwaiting s* (*rcurrent s*) *a* ⟶ *rstep s a* = *s*
⟨*proof*⟩
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
(↑*s0*)
rcurrent
rcswitch
rkinvolved
rifp
rvpeq
rAS-set
rinvariant
rprecondition
raborting
rwaiting
rset-error-code
⟨*proof*⟩

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

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

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

theorem *risecure*:
Inst.isecure

⟨*proof*⟩

end

5 Related Work

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

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

Roscoe and Goldsmith provide CSP-based definitions of NI for the transitive and the intransitive case, here dubbed as lazy and mixed independence. The latter one is more restrictive than Rushby's IP-security. Their critique on IP-secure, however, is not universally accepted [**?**]. 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 $l := \texttt{declassify}(h)$ (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-security for a model in which the security policy is Dynamic. Eggert et al. defined i-secure, an extension of IP-secure. Their model extends Rushby's model (Mealy machines) with Local security policies. Murray et al. extends Von Oheimb definition of noninfluence to apply to a model that does not assume a static mapping of actions to domains. This makes it applicable to OS's, as in such a setting such a mapping does not exist [20]. NI-OS has been applied to the seL4 separation kernel [20], [14].

Most definitions have an associated methodology. Various methodologies are based on unwinding [8]. This breaks down the proof of NI into smaller proof obligations (PO's). These PO's can be checked by some manual proof [24], [10], model checking [32] or dedicated algorithms [5]. The methodology of

Murray et al. is a combination of unwinding, automated deduction and manual proofs. Some definitions are undecidable and have no suitable unwinding.

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

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

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

6 Conclusion

We have introduced a generic theory of intransitive non-interference for separation kernels with control as a series of locales and extensible record definitions in order to a 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

[1] E. Alkassar, M. A. Hillebrand, D. Leinenbach, N. Schirmer, A. Starostin, and A. Tsyban. Balancing the load. *J. Autom. Reasoning*, 42(2-4):389–454, 2009.

[2] J. Brygier, R. Fuchsen, and H. Blasum. Pikeos: Safe and secure virtualization in a separation microkernel. Technical report, 2009.

[3] M. Daum, J. Dörrenbächer, and B. Wolff. Proving fairness and implementation correctness of a microkernel scheduler. *J. Autom. Reasoning*, 42(2-4):349–388, 2009.

[4] S. Eggert, H. Schnoor, and T. Wilke. Noninterference with local policies. In K. Chatterjee and J. Sgall, editors, *Mathematical Foundations of Computer Science 2013*, volume 8087 of *Lecture Notes in Computer Science*, pages 337–348. Springer Berlin Heidelberg, 2013.

[5] S. Eggert, R. van der Meyden, H. Schnoor, and T. Wilke. The complexity of intransitive noninterference. In *IEEE Symposium on Security and Privacy*, pages 196–211, 2011.

[6] K. Elphinstone, G. Klein, P. Derrin, T. Roscoe, and G. Heiser. Towards a practical, verified kernel. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*, HOTOS'07, pages 20:1–20:6, Berkeley, CA, USA, 2007. USENIX Association.

[7] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1984.

[8] J. A. Goguen and J. Meseguer. Unwinding and inference control. In *IEEE Symposium on Security and Privacy*, pages 75–87, 1984.

[9] D. Greve, M. Wilding, and W. M. Vanfleet. A separation kernel formal security policy. In *Fourth International Workshop on the ACL2 Prover and Its Applications (ACL2-2003)*, 2003.

[10] J. Haigh and W. Young. Extending the non-interference version of mls for sat. *IEEE Transactions on Software Engineering*, 13:141–150, 1987 1987.

[11] C. L. Heitmeyer, M. Archer, E. I. Leonard, and J. McLean. Formal specification and verification of data separation in a separation kernel for an embedded system. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 346–355, New York, NY, USA, 2006. ACM.

[12] R. Kaiser and S. Wagner. Evolution of the PikeOS microkernel. In *In: FirstInternational Workshop on Microkernels for Embedded Systems*, 2007.

[13] G. Klein. Operating system verification—an overview. *Sadhana*, 34(1):27–69, 2009.

[14] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[15] R. Leslie. Dynamic intransitive noninterference. In *IEEE International Symposium on Secure Software Engineering*, pages 75–87, 2006.

[16] J. Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 237–250. ACM Press, 1995.

[17] H. Mantel. Information flow control and applications — bridging a gap —. In J. Oliveira and P. Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity*, volume 2021 of *Lecture Notes in Computer Science*, pages 153–172. Springer Berlin Heidelberg, 2001.

[18] W. Martin, P. White, F. S. Taylor, and A. Goldberg. Formal construction of the mathematically analyzed separation kernel. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, ASE '00, pages 133–, Washington, DC, USA, 2000. IEEE Computer Society.

[19] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. sel4: from general purpose to a proof of information flow enforcement. In *IEEE Symposium on Security and Privacy*, pages 415–429, San Francisco, CA, May 2013.

[20] T. Murray, D. Matichuk, M. Brassil, P. Gammie, and G. Klein. Noninterference for operating system kernels. In Chris Hawblitzel and Dale Miller, editor, *The Second International Conference on Certified Programs and Proofs*, pages 126–142, Kyoto, dec 2012. Springer.

[21] T. Nipkow, L. C. Paulson, and M. Wenzel. Isabelle/hol: a proof assistant for higher- order logic. 2012.

[22] A. W. Roscoe. What is intransitive noninterference. In *In Proc. of the 12th IEEE Computer Security Foundations Workshop*, pages 228–238, 1999.

[23] J. Rushby. Design and verification of secure systems. *ACM SIGOPS Operating Systems Review*, 15:12–21, 1981.

[24] J. Rushby. Noninterference, transitivity, and channel-control security policies. Technical report, dec 1992.

[25] J. Rushby. Noninterference, transitivity, and channel-control security policies. Technical report, dec 1992.

[26] A. Sabelfeld and A. C. Myers. Language-based information-flow security,. *Selected Areas in Communications, IEEE Journal on,*, 21(1):5–19, 2003.

[27] G. Schellhorn, W. Reif, A. Schairer, P. Karger, V. Austel, and D. Toll. Verification of a formal security model for multiapplicative smart cards. In F. Cuppens, Y. Deswarte, D. Gollmann, and M. Waidner, editors, *Computer Security - ESORICS 2000*, volume 1895 of *Lecture Notes in Computer Science*, pages 17–36. Springer Berlin Heidelberg, 2000.

[28] J. S. Shapiro and S. Weber. Verifying the eros confinement mechanism. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, SP '00, pages 166–, Washington, DC, USA, 2000. IEEE Computer Society.

[29] R. Van Der Meyden. What, indeed, is intransitive noninterference? In *Proceedings of the 12th European Conference on Research in Computer Security*, ESORICS'07, pages 235–250, Berlin, Heidelberg, 2007. Springer-Verlag.

[30] R. van der Meyden and C. Zhang. A comparison of semantic models for noninterference. *Theoretical Computer Science*, 411(47):4123 – 4147, 2010.

[31] F. Verbeek, J. Schmaltz, S. Tverdyshev, H. Blasum, and O. Havle. A new theory of intransitive noninterference for separation kernels with control (manuscript), 2013.

[32] M. Whalen, D. Greve, and L. Wagner. Model checking information flow. In D. S. Hardin, editor, *Design and Verification of Microprocessor Systems for High-Assurance Applications*, pages 381–428. Springer US, 2010.