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

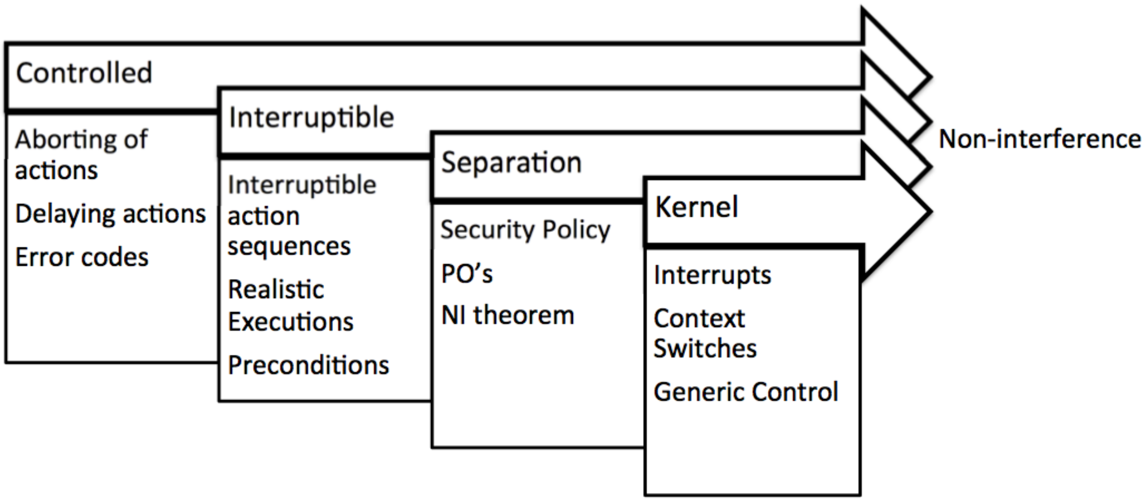

Figure 1: Overview of CISK modular structure

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))*
unfolding *B2-def B-def* **by**(*cases s,cases t,simp+*)
lemma *rewrite-B-None[simp]*:
shows *None ⇀ α = True*
unfolding *B-def* **by**(*auto*)
lemma *rewrite-B-m-True[simp]*:
shows *m ⇀ (λ a . True) = True*
unfolding *B-def* **by**(*cases m,simp+*)
lemma *rewrite-B2-cases*:
shows *(case a of None ⇒ True | (Some s) ⇒ (case b of None ⇒ True | (Some t) ⇒ f s t))*
= (∀ s t . a = (Some s) ∧ b = (Some t) ⟶ f s t)
by(*cases a,simp,cases b,simp+*)

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

end

2.2 Theorems on lists

theory *List-Theorems*
imports *Main*
begin

definition *lastn* :: *nat ⇒ 'a list ⇒ 'a list*
where *lastn n x = drop ((length x) − n) x*

definition *is-sub-seq* :: *'a ⇒ 'a ⇒ 'a list ⇒ bool*
where *is-sub-seq a b x ≡ ∃ n . Suc n < length x ∧ x!n = a ∧ x!(Suc n) = b*

definition *prefixes* :: *'a list set ⇒ 'a list set*
where *prefixes s ≡ {x . ∃ n y . n > 0 ∧ y ∈ s ∧ take n y = x}*

lemma *drop-one[simp]*:
shows *drop (Suc 0) x = tl x* **by**(*induct x,auto*)
lemma *length-ge-one*:
shows *x ≠ [] ⟶ length x ≥ 1* **by**(*induct x,auto*)
lemma *take-but-one[simp]*:
shows *x ≠ [] ⟶ lastn ((length x) − 1) x = tl x* **unfolding** *lastn-def*
using *length-ge-one*[**where** *x=x*] **by** *auto*
lemma *Suc-m-minus-n[simp]*:
shows *m ≥ n ⟶ Suc m − n = Suc (m − n)* **by** *auto*

lemma *lastn-one-less*:
 shows $n > 0 \wedge n \leq length\ x \wedge lastn\ n\ x = (a\#y) \longrightarrow lastn\ (n - 1)\ x = y$ **unfolding** *lastn-def*
 using *drop-Suc*[**where** $n{=}length\ x - n$ **and** $xs{=}x$] *drop-tl*[**where** $n{=}length\ x - n$ **and** $xs{=}x$]
 by(*auto*)
lemma *list-sub-implies-member*:
 shows $\forall\ a\ x\ .\ set\ (a\#x) \subseteq Z \longrightarrow a \in Z$ **by** *auto*
lemma *subset-smaller-list*:
 shows $\forall\ a\ x\ .\ set\ (a\#x) \subseteq Z \longrightarrow set\ x \subseteq Z$ **by** *auto*
lemma *second-elt-is-hd-tl*:
 shows $tl\ x = (a\ \#\ x') \longrightarrow a = x\ !\ 1$
 by (*cases x,auto*)
lemma *length-ge-2-implies-tl-not-empty*:
 shows $length\ x \geq 2 \longrightarrow tl\ x \neq []$
 by (*cases x,auto*)
lemma *length-lt-2-implies-tl-empty*:
 shows $length\ x < 2 \longrightarrow tl\ x = []$
 by (*cases x,auto*)
lemma *first-second-is-sub-seq*:
 shows $length\ x \geq 2 \Longrightarrow is\text{-}sub\text{-}seq\ (hd\ x)\ (x!1)\ x$
proof–
 assume $length\ x \geq 2$
 hence *1*: $(Suc\ 0) < length\ x$ **by** *auto*
 hence $x!0 = hd\ x$ **by**(*cases x,auto*)
 from *this 1* **show** $is\text{-}sub\text{-}seq\ (hd\ x)\ (x!1)\ x$ **unfolding** *is-sub-seq-def* **by** *auto*
qed
lemma *hd-drop-is-nth*:
 shows $n < length\ x \Longrightarrow hd\ (drop\ n\ x) = x!n$
proof(*induct x arbitrary*: *n*)
case *Nil*
 thus *?case* **by** *simp*
next
case (*Cons a x*)
{
 have $hd\ (drop\ n\ (a\ \#\ x)) = (a\ \#\ x)\ !\ n$
 proof(*cases n*)
 case *0*
  thus *?thesis* **by** *simp*
 next
 case (*Suc m*)
  from *Suc Cons* **show** *?thesis* **by** *auto*
 qed
}
thus *?case* **by** *auto*
qed

lemma *def-of-hd*:
 shows $y = a\ \#\ x \longrightarrow hd\ y = a$ **by** *simp*
lemma *def-of-tl*:
 shows $y = a\ \#\ x \longrightarrow tl\ y = x$ **by** *simp*
lemma *drop-yields-results-implies-nbound*:
 shows $drop\ n\ x \neq [] \longrightarrow n < length\ x$
by(*induct x,auto*)
lemma *consecutive-is-sub-seq*:
 shows $a\ \#\ (b\ \#\ x) = lastn\ n\ y \Longrightarrow is\text{-}sub\text{-}seq\ a\ b\ y$
proof–
 assume *1*: $a\ \#\ (b\ \#\ x) = lastn\ n\ y$
 from *1 drop-Suc*[**where** $n{=}(length\ y) - n$ **and** $xs{=}y$]

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

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

lemma *set-tl-is-subset*:
shows *set* (*tl x*) ⊆ *set x* **by**(*induct x*,*auto*)
lemma *x-is-hd-snd-tl*:
shows *length x* ≥ 2 ⟶ *x* = (*hd x*) # *x*!*1* # *tl*(*tl x*)
proof(*induct x*)
case *Nil*
show *?case* **by** *auto*
case (*Cons a xs*)
show *?case* **by**(*induct xs*,*auto*)
qed

lemma *tl-x-not-x*:
shows *x* ≠ [] ⟶ *tl x* ≠ *x* **by**(*induct x*,*auto*)
lemma *tl-hd-x-not-tl-x*:
shows *x* ≠ [] ∧ *hd x* ≠ [] ⟶ *tl* (*hd x*) # *tl x* ≠ *x* **using** *tl-x-not-x* **by**(*induct x*,*simp*,*auto*)

end

3 A generic model for separation kernels

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

This section defines a detailed generic model of separation kernels called CISK (Controlled Inter-

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

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

- *locale "Kernel" defines a highly generic model for a kernel, with execution semantics. It defines a state transition system with some extensions to the one used in [25]. The transition system defined here stores the currently active domain in the state, and has transitions for explicit context switches and interrupts and provides a notion of control. As each operation of the system will be split into atomic actions in our model, only certain sequences of actions will correspond to a run on a real system. Therefore, the function* 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*)
using *not0-implies-Suc* **by** (*metis option.exhaust prod-cases3*,*auto*)
termination by *lexicographic-order*
end

end

3.2 SK (Separation Kernel)

theory *SK*
imports *K*
begin

Locale Kernel is now refined to a generic model of a separation kernel. The security policy is represented using function *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–
 {
  fix *v*
  assume *ifp-v-u*: *ifp^** v u*

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

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

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

  from *ifp-v-u* **and** *not-ifp-curr-u* **have** *unrelated*: *¬ifp^** (current s) v* **using** *rtranclp-trans* **by** *metis*
  from *this current-next-state[THEN spec,THEN spec,***where** *x1=s] vpeq-reflexive*
   *unrelated locally-respects[THEN spec,THEN spec,THEN spec,***where** *x1=next-state s execs* **and** *x=v* **and** *x2=the (next-action s execs)] prec-s*
   have *vpeq v (next-state s execs) (step (next-state s execs) (next-action s execs))*
   unfolding *step-def precondition-def B-def*
   by *(cases next-action s execs,auto)*
 from *unrelated this locally-respects-next-state vpeq-transitive* **have** *vpeq v s (step (next-state s execs) (next-action s execs))* **by** *blast*
  from *this* **and** *ifp-v-u* **and** *vpeq-s-t* **and** *vpeq-symmetric* **and** *vpeq-transitive* **have** *vpeq v (step (next-state s execs) (next-action s execs)) t* **by** *metis*
 }
thus *?thesis* **by** *auto*
qed

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

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

assume *ifp-curr*: *ifp*^∗∗ (*current s*) *u*
from *vpeq-s-t* **have** *vpeq-curr-s-t*: *ifp*^∗∗ (*current s*) *u* ⟶ *vpeq* (*current s*) *s t′* **by** *auto*
from *ifp-curr precs*
next-state-consistent[*THEN spec*,*THEN spec*,**where** *x1*=*s* **and** *x*=*t′*] *vpeq-curr-s-t vpeq-s-t*
current-next-state current-s-t weakly-step-consistent[*THEN spec*,*THEN spec*,*THEN spec*,*THEN spec*,**where** *x3*=*next-state s execs* **and** *x2*=*next-state t′ execs* **and** *x*=*the a*]
show ∀ *v* . *ifp*^∗∗ *v u* ⟶ *vpeq v* (*step* (*next-state s execs*) *a*) (*step* (*next-state t′ execs*) *a*)
unfolding *step-def precondition-def B-def*
by (*cases a*,*auto*)
qed

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

lemma *vpeq-ns-nt-not-ifp-u*:
assumes *purged-a-a2*: *purged-relation u execs execs2*
and *prec-s*: *precondition* (*next-state s execs*) (*next-action s execs*)
and *current-s-t*: *current s* = *current t′*
and *vpeq-s-t*: ∀ *v* . *ifp*^∗∗ *v u* ⟶ *vpeq v s 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−
{
assume *not-ifp*: ¬*ifp*^∗∗ (*current s*) *u*
assume *prec-t*: *precondition* (*next-state t′ execs2*) (*next-action t′ execs2*)
fix *a a′ v*
assume *ifp-v-u*: *ifp*^∗∗ *v u*
from *not-ifp* **and** *purged-a-a2* **have** ¬*ifp*^∗∗ (*current s*) *u* **unfolding** *purged-relation-def* **by** *auto*
from *this* **and** *ifp-v-u* **have** *not-ifp-curr-v*: ¬*ifp*^∗∗ (*current s*) *v* **using** *rtranclp-trans* **by** *metis*
from *this current-next-state*[*THEN spec*,*THEN spec*,**where** *x1*=*s* **and** *x*=*execs*] *prec-s vpeq-reflexive*
locally-respects[*THEN spec*,*THEN spec*,*THEN spec*,**where** *x1*=*next-state s execs* **and** *x2*=*the* (*next-action s execs*) **and** *x*=*v*]
have *vpeq v* (*next-state s execs*) (*step* (*next-state s execs*) (*next-action s execs*))
unfolding *step-def precondition-def B-def*
by (*cases next-action s execs*,*auto*)
from *not-ifp-curr-v this locally-respects-next-state vpeq-transitive*
have *vpeq-s-ns*: *vpeq v s* (*step* (*next-state s execs*) (*next-action s execs*))
by *blast*
from *not-ifp-curr-v current-s-t current-next-state*[*THEN spec*,*THEN spec*,**where** *x1*=*t′* **and** *x*=*execs2*] *prec-t*
locally-respects[*THEN spec*,*THEN spec*,**where** *x*=*next-state t′ execs2*] *vpeq-reflexive*
have *0*: *vpeq v* (*next-state t′ execs2*) (*step* (*next-state t′ execs2*) (*next-action t′ execs2*))
unfolding *step-def precondition-def B-def*
by (*cases next-action t′ execs2*,*auto*)
from *not-ifp-curr-v current-s-t current-next-state* **have** *1*: ¬*ifp*^∗∗ (*current t′*) *v*
using *rtranclp-trans* **by** *auto*
from *0 1 locally-respects-next-state vpeq-transitive*
have *vpeq-t-nt*: *vpeq v t′* (*step* (*next-state t′ execs2*) (*next-action t′ execs2*))
by *blast*
from *vpeq-s-ns* **and** *vpeq-t-nt* **and** *vpeq-s-t* **and** *ifp-v-u* **and** *vpeq-symmetric* **and** *vpeq-transitive*
have *vpeq-ns-nt*: *vpeq v* (*step* (*next-state s execs*) (*next-action s execs*)) (*step* (*next-state t′ execs2*) (*next-action t′ execs2*))
by *blast*
}
thus *?thesis* **by** *auto*
qed

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

```
lemma unwinding-implies-view-partitioned-ind:
shows view-partitioned-ind
proof−
{
 fix execs execs2 s t n u
  have equivalent-states s t u ∧ purged-relation u execs execs2 ⟶ equivalent-states (run n s execs) (run n t
execs2) u
 proof (induct n s execs arbitrary: t u execs2 rule: run.induct)
   case (1 s execs t u execs2)
    show ?case by auto
   next
   case (2 n execs t u execs2)
    show ?case by simp
   next
   case (3 n s execs t u execs2)
   assume interrupt-s: interrupt (Suc n)
   assume IH: (⋀t u execs2.
         equivalent-states (Some (cswitch (Suc n) s)) t u ∧ purged-relation u execs execs2 ⟶
         equivalent-states (run n (Some (cswitch (Suc n) s)) execs) (run n t execs2) u)
   {
    fix t′
    assume t = Some t′
    fix rs
    assume rs: run (Suc n) (Some s) execs = Some rs
    fix rt
    assume rt: run (Suc n) (Some t′) execs2 = Some rt

    assume vpeq-s-t: ∀ v . ifp^** v u ⟶ vpeq v s t′
    assume current-s-t: current s = current t′
    assume purged-a-a2: purged-relation u execs execs2
```

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

— We prove two properties: the states rs and rt have equal active domains (current-rs-rt) and are vpeq for all domains v that may influence u (vpeq-rs-rt). Both are proven using the IH. To use the IH, we have to prove that the properties hold for the next step (in this case, a context switch). Statement current-ns-nt states that after one step states ns and nt have the same active domain. Statement vpeq-ns-nt states that after one step states ns and nt are vpeq for all domains v that may influence u (vpeq-rs-rt).

```
    from current-s-t cswitch-independent-of-state
     have current-ns-nt: current (cswitch (Suc n) s) = current (cswitch (Suc n) t′) by blast
    from cswitch-consistency vpeq-s-t
     have vpeq-ns-nt: ∀ v . ifp^** v u ⟶ vpeq v (cswitch (Suc n) s) (cswitch (Suc n) t′) by auto
     from current-ns-nt vpeq-ns-nt interrupt-s vpeq-reflexive purged-a-a2 current-s-t IH[where u=u and t=Some
(cswitch (Suc n) t′) and ?execs2.0=execs2]
    have current-rs-rt: current rs = current rt using rs rt by(auto)
    {
     fix v
     assume ia: ifp^** v u
     from current-ns-nt vpeq-ns-nt ia interrupt-s vpeq-reflexive purged-a-a2 IH[where u=u and t=Some (cswitch
(Suc n) t′) and ?execs2.0=execs2]
     have vpeq-rs-rt: vpeq v rs rt using rs rt by(auto)
    }
    from current-rs-rt and this have equivalent-states (Some rs) (Some rt) u by auto
   }
   thus ?case by(simp add:option.splits,cases t,simp+)
   next
   case (4 n execs s t u execs2)
```

assume *not-interrupt*: $\neg$*interrupt* (*Suc n*)
assume *thread-empty-s*: *thread-empty*(*execs* (*current s*))
assume *IH*: ($\bigwedge$*t u execs2*. *equivalent-states* (*Some s*) *t u* $\wedge$ *purged-relation u execs execs2* $\longrightarrow$ *equivalent-states* (*run n* (*Some s*) *execs*) (*run n t execs2*) *u*)
{
fix t'
assume *t*: *t* = *Some* t'
fix *rs*
assume *rs*: *run* (*Suc n*) (*Some s*) *execs* = *Some rs*
fix *rt*
assume *rt*: *run* (*Suc n*) (*Some* t') *execs2* = *Some rt*

assume *vpeq-s-t*: $\forall$ *v* . *ifp*^** *v u* $\longrightarrow$ *vpeq v s* t'
assume *current-s-t*: *current s* = *current* t'
assume *purged-a-a2*: *purged-relation u execs execs2*

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

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

from *ifp-reflexive* **and** *vpeq-s-t* **have** *vpeq-s-t-u*: *vpeq u s* t' **by** *auto*
from *thread-empty-s* **and** *purged-a-a2* **and** *current-s-t* **have** *purged-a-na2*: $\neg$*ifp*^** (*current* t') *u* $\longrightarrow$ *purged-relation u execs* (*next-execs* t' *execs2*)
by(*unfold next-execs-def* ,*unfold purged-relation-def* ,*auto*)
from *step-atomicity current-next-state current-s-t* **have** *current-s-nt*: *current s* = *current* (*step* (*next-state* t' *execs2*) (*next-action* t' *execs2*))
unfolding *step-def*
by (*cases next-action* t' *execs2*,*auto*)

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

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

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

hence *run* (*Suc n*) (*Some s*) *execs* = *None* **using** *not-interrupt thread-not-empty-s* **by** *simp*
thus *?case* **by**(*simp add*:*option.splits*)
next
case (*6 n execs s t u execs2*)
assume *not-interrupt*: ¬*interrupt* (*Suc n*)
assume *thread-not-empty-s*: ¬*thread-empty*(*execs* (*current s*))
assume *prec-s*: *precondition* (*next-state s execs*) (*next-action s execs*)
assume *IH*: (⋀*t u execs2*.
equivalent-states (*Some* (*step* (*next-state s execs*) (*next-action s execs*))) *t u* ∧
purged-relation u (*next-execs s execs*) *execs2* ⟶
equivalent-states
(*run n* (*Some* (*step* (*next-state s execs*) (*next-action s execs*))) (*next-execs s execs*))
(*run n t execs2*) *u*)
{
fix *t′*
assume *t*: *t* = *Some t′*
fix *rs*
assume *rs*: *run* (*Suc n*) (*Some s*) *execs* = *Some rs*
fix *rt*
assume *rt*: *run* (*Suc n*) (*Some t′*) *execs2* = *Some rt*

assume *vpeq-s-t*: ∀ *v* . *ifp*^** *v u* ⟶ *vpeq v s t′*
assume *current-s-t*: *current s* = *current t′*
assume *purged-a-a2*: *purged-relation u execs execs2*

— The following terminology is used: states rs and rt (for: run-s and run-t) are the states after a run. States ns and nt (for: next-s and next-t) are the states after one step.
— We prove two properties: the states rs and rt have equal active domains (current-rs-rt) and are vpeq for all domains v that may influence u (vpeq-rs-rt). Both are proven using the IH. To use the IH, we have to prove that the properties hold for the next step (in this case, state s executes an action). Statement current-ns-nt states that after one step states ns and nt have the same active domain. Statement vpeq-ns-nt states that after one step states ns and nt are vpeq for all domains v that may influence u (vpeq-rs-rt).

— Some lemma's used in the remainder of this case.
from *ifp-reflexive* **and** *vpeq-s-t* **have** *vpeq-s-t-u*: *vpeq u s t′* **by** *auto*
from *step-atomicity* **and** *current-s-t current-next-state*
have *current-ns-nt*: *current* (*step* (*next-state s execs*) (*next-action s execs*)) = *current* (*step* (*next-state t′ execs2*) (*next-action t′ execs2*))
unfolding *step-def*
by (*cases next-action s execs*,*cases next-action t′ execs2*,*simp*,*simp*,*cases next-action t′ execs2*,*simp*,*simp*)
from *vpeq-s-t* **have** *vpeq-curr-s-t*: *ifp*^** (*current s*) *u* ⟶ *vpeq* (*current s*) *s t′* **by** *auto*
from *prec-s involved-ifp*[*THEN spec*,*THEN spec*,**where** *x1*=*next-state s execs* **and** *x*=*next-action s execs*] *vpeq-s-t* **have** *vpeq-involved*: *ifp*^** (*current s*) *u* ⟶ (∀ *d* ∈ *involved* (*next-action s execs*) . *vpeq d s t′*)
using *current-next-state*
unfolding *involved-def precondition-def B-def*
by(*cases next-action s execs*,*simp*,*auto*,*metis converse-rtranclp-into-rtranclp*)
from *current-s-t next-execs-consistent vpeq-curr-s-t vpeq-involved*
have *next-execs-t*: *ifp*^** (*current s*) *u* ⟶ *next-execs t′ execs* = *next-execs s execs*
unfolding *next-execs-def*
by(*auto*)
from *current-s-t purged-a-a2 thread-not-empty-s next-action-consistent*[*THEN spec*,*THEN spec*,**where** *x1*=*s* **and** *x*=*t′*] *vpeq-curr-s-t vpeq-involved*
have *next-action-s-t*: *ifp*^** (*current s*) *u* ⟶ *next-action t′ execs2* = *next-action s execs*
by(*unfold next-action-def*,*unfold purged-relation-def*,*auto*)
from *purged-a-a2 current-s-t next-execs-consistent*[*THEN spec*,*THEN spec*,*THEN spec*,**where** *x2*=*s* **and** *x1*=*t′* **and** *x*=*execs*]
vpeq-curr-s-t vpeq-involved

have *purged-na-na2*: *purged-relation u* (*next-execs s execs*) (*next-execs t′ execs2*)
unfolding *next-execs-def purged-relation-def*
by(*auto*)
from *purged-a-a2* **and** *purged-relation-def* **and** *thread-not-empty-s* **and** *current-s-t* **have** *thread-not-empty-t*: *ifp^*** (*current s*) *u* ⟶ ¬*thread-empty*(*execs2* (*current t′*)) **by** *auto*
from *step-atomicity current-s-t current-next-state* **have** *current-ns-t*: *current* (*step* (*next-state s execs*) (*next-action s execs*)) = *current t′*
unfolding *step-def*
by (*cases next-action s execs*,*auto*)
from *step-atomicity* **and** *current-s-t* **have** *current-s-nt*: *current s* = *current* (*step t′* (*next-action t′ execs2*))
unfolding *step-def*
by (*cases next-action t′ execs2*,*auto*)
from *purged-a-a2* **have** *purged-na-a*: ¬*ifp^*** (*current s*) *u* ⟶ *purged-relation u* (*next-execs s execs*) *execs2*
by(*unfold next-execs-def* ,*unfold purged-relation-def* ,*auto*)

— The proof is by case distinction. If the current domain can interact with u (case curr-ifp-u), then either in state t the precondition holds (case t-prec) or not. If it holds, then lemma vpeq-ns-nt-ifp-u applies. Otherwise, the proof is trivial as the theorem holds vacuously. If the domain cannot interact with u, (case curr-not-ifp-u), then lemma vpeq-ns-nt-not-ifp-u applies.
have *current-rs-rt*: *current rs* = *current rt*
proof (*cases ifp^*** (*current s*) *u rule* :*case-split*[*case-names curr-ifp-u curr-not-ifp-u*])
case *curr-ifp-u*
show *?thesis*
proof (*cases precondition* (*next-state t′ execs2*) (*next-action t′ execs2*) *rule* :*case-split*[*case-names prec-t prec-not-t*])
case *prec-t*
have *thread-not-empty-t*: ¬*thread-empty*(*execs2* (*current t′*)) **using** *thread-not-empty-t curr-ifp-u* **by** *auto*
from
current-ns-nt next-execs-t next-action-s-t purged-a-a2
curr-ifp-u prec-t prec-s vpeq-ns-nt-ifp-u[**where** *a*=(*next-action s execs*)] *vpeq-s-t current-s-t*
have *equivalent-states* (*Some* (*step* (*next-state s execs*) (*next-action s execs*))) (*Some* (*step* (*next-state t′ execs2*) (*next-action t′ execs2*))) *u*
unfolding *purged-relation-def next-state-def*
by *auto*
from *this*
IH[**where** *u*=*u* **and** *?execs2.0*=(*next-execs t′ execs2*) **and** *t*=*Some* (*step* (*next-state t′ execs2*) (*next-action t′ execs2*))]
current-ns-nt purged-na-na2
have *equivalent-states* (*run n* (*Some* (*step* (*next-state s execs*) (*next-action s execs*))) (*next-execs s execs*)) (*run n* (*Some* (*step* (*next-state t′ execs2*) (*next-action t′ execs2*))) (*next-execs t′ execs2*)) *u*
by *auto*
from *prec-t thread-not-empty-t prec-s* **and** *this* **and** *not-interrupt* **and** *thread-not-empty-s* **and** *next-action-s-t*
show *?thesis* **using** *rs rt* **by** *auto*
next
case *prec-not-t*
from *curr-ifp-u prec-not-t thread-not-empty-t not-interrupt* **show** *?thesis* **using** *rt* **by** *simp*
qed
next
case *curr-not-ifp-u*
show *?thesis*
proof (*cases thread-empty*(*execs2* (*current t′*)) *rule* :*case-split*[*case-names t-empty t-not-empty*])
case *t-not-empty*
show *?thesis*
proof (*cases precondition* (*next-state t′ execs2*) (*next-action t′ execs2*) *rule* :*case-split*[*case-names t-prec t-not-prec*])
case *t-prec*
from *curr-not-ifp-u t-prec IH*[**where** *u*=*u* **and** *?execs2.0*=(*next-execs t′ execs2*) **and** *t*=*Some* (*step*

(*next-state t′ execs2*) (*next-action t′ execs2*))]
current-ns-nt next-execs-t purged-na-na2 vpeq-ns-nt-not-ifp-u current-s-t vpeq-s-t prec-s purged-a-a2
have *equivalent-states* (*run n* (*Some* (*step* (*next-state s execs*) (*next-action s execs*))) (*next-execs s execs*))
(*run n* (*Some* (*step* (*next-state t′ execs2*) (*next-action t′ execs2*))) (*next-execs t′ execs2*)) *u* **by** *auto*
from *this t-prec curr-not-ifp-u t-not-empty prec-s not-interrupt thread-not-empty-s* **show** *?thesis* **using** *rs rt* **by** *auto*
next
case *t-not-prec*
from *t-not-prec t-not-empty not-interrupt* **show** *?thesis* **using** *rt* **by** *simp*
qed
next
case *t-empty*
from *curr-not-ifp-u* **and** *prec-s* **and** *vpeq-s-t* **and** *locally-respects* **and** *vpeq-ns-t current-next-state locally-respects-next-state*
have *vpeq-ns-t*: (∀ *v* . *ifp*^∗∗ *v u* ⟶ *vpeq v* (*step* (*next-state s execs*) (*next-action s execs*)) *t′*)
by *blast*
from *curr-not-ifp-u IH*[**where** *t=Some t′* **and** *u=u* **and** *?execs2.0=execs2*] **and** *current-ns-t* **and** *next-execs-t* **and** *purged-na-a* **and** *vpeq-ns-t* **and** *this*
have *equivalent-states* (*run n* (*Some* (*step* (*next-state s execs*) (*next-action s execs*))) (*next-execs s execs*))
(*run n* (*Some t′*) *execs2*) *u* **by** *auto*
from *this not-interrupt thread-not-empty-s t-empty prec-s* **show** *?thesis* **using** *rs rt* **by** *auto*
qed
qed
{
fix *v*
assume *ia*: *ifp*^∗∗ *v u*

have *vpeq v rs rt*
proof (*cases ifp*^∗∗ (*current s*) *u rule* :*case-split*[*case-names curr-ifp-u curr-not-ifp-u*])
case *curr-ifp-u*
show *?thesis*
proof (*cases precondition* (*next-state t′ execs2*) (*next-action t′ execs2*) *rule* :*case-split*[*case-names t-prec t-not-prec*])
case *t-prec*
have *thread-not-empty-t*: ¬*thread-empty*(*execs2* (*current t′*)) **using** *thread-not-empty-t curr-ifp-u* **by** *auto*
from
current-ns-nt next-execs-t next-action-s-t purged-a-a2
curr-ifp-u t-prec prec-s vpeq-ns-nt-ifp-u[**where** *a=*(*next-action s execs*)] *vpeq-s-t current-s-t*
have *equivalent-states* (*Some* (*step* (*next-state s execs*) (*next-action s execs*))) (*Some* (*step* (*next-state t′ execs2*) (*next-action t′ execs2*))) *u*
unfolding *purged-relation-def next-state-def*
by *auto*
from *this*
IH[**where** *u=u* **and** *?execs2.0=*(*next-execs t′ execs2*) **and** *t=Some* (*step* (*next-state t′ execs2*) (*next-action t′ execs2*))]
current-ns-nt purged-na-na2
have *equivalent-states* (*run n* (*Some* (*step* (*next-state s execs*) (*next-action s execs*))) (*next-execs s execs*))
(*run n* (*Some* (*step* (*next-state t′ execs2*) (*next-action t′ execs2*))) (*next-execs t′ execs2*)) *u*
by *auto*
from *ia curr-ifp-u t-prec thread-not-empty-t prec-s* **and** *this* **and** *not-interrupt* **and** *thread-not-empty-s* **and** *next-action-s-t*
show *?thesis* **using** *rs rt* **by** *auto*
next

case *t-not-prec*
from *curr-ifp-u t-not-prec thread-not-empty-t not-interrupt* **show** *?thesis* **using** *rt* **by** *simp*
qed
next
case *curr-not-ifp-u*
show *?thesis*
proof (*cases thread-empty*(*execs2* (*current t′*)) *rule* :*case-split*[*case-names t-empty t-not-empty*])
case *t-not-empty*
show *?thesis*
proof (*cases precondition* (*next-state t′ execs2*) (*next-action t′ execs2*) *rule* :*case-split*[*case-names t-prec t-not-prec*])
case *t-prec*
from *curr-not-ifp-u t-prec IH*[**where** *u*=*u* **and** *?execs2.0*=(*next-execs t′ execs2*) **and** *t*=*Some* (*step* (*next-state t′ execs2*) (*next-action t′ execs2*))]
current-ns-nt next-execs-t purged-na-na2 vpeq-ns-nt-not-ifp-u current-s-t vpeq-s-t prec-s purged-a-a2
have *equivalent-states* (*run n* (*Some* (*step* (*next-state s execs*) (*next-action s execs*))) (*next-execs s execs*))
(*run n* (*Some* (*step* (*next-state t′ execs2*) (*next-action t′ execs2*))) (*next-execs t′ execs2*)) *u* **by** *auto*
from *ia this t-prec curr-not-ifp-u t-not-empty prec-s not-interrupt thread-not-empty-s* **show** *?thesis* **using** *rs rt* **by** *auto*
next
case *t-not-prec*
from *t-not-prec t-not-empty not-interrupt* **show** *?thesis* **using** *rt* **by** *simp*
qed
next
case *t-empty*
from *curr-not-ifp-u prec-s* **and** *vpeq-s-t* **and** *locally-respects* **and** *vpeq-ns-t current-next-state locally-respects-next-state*
have *vpeq-ns-t*: (∀ *v* . *ifp*^∗∗ *v u* ⟶ *vpeq v* (*step* (*next-state s execs*) (*next-action s execs*)) *t′*)
by *blast*
from *curr-not-ifp-u IH*[**where** *t*=*Some t′* **and** *u*=*u* **and** *?execs2.0*=*execs2*] **and** *current-ns-t* **and** *next-execs-t* **and** *purged-na-a* **and** *vpeq-ns-t* **and** *this*
have *equivalent-states* (*run n* (*Some* (*step* (*next-state s execs*) (*next-action s execs*))) (*next-execs s execs*))
(*run n* (*Some t′*) *execs2*) *u* **by** *auto*
from *ia this not-interrupt thread-not-empty-s t-empty prec-s* **show** *?thesis* **using** *rs rt* **by** *auto*
qed
qed
}
from *current-rs-rt* **and** *this* **have** *equivalent-states* (*Some rs*) (*Some rt*) *u* **by** *auto*
}
thus *?case* **by**(*simp add*:*option.splits*,*cases t*,*simp*+)
qed
}
thus *?thesis*
unfolding *view-partitioned-ind-def* **by** *auto*
qed

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

lemma *unwinding-implies-view-partitioned*:
shows *view-partitioned*
proof–
from *unwinding-implies-view-partitioned-ind* **have** *view-partitioned-inductive*: *view-partitioned-ind*
by *blast*
have *purged-relation*: ∀ *u execs* . *purged-relation u execs* (*purge execs u*)
by(*unfold purged-relation-def* , *unfold purge-def* , *auto*)

```
{
 fix execs s t n u
 assume 1: equivalent-states s t u
 from this view-partitioned-inductive purged-relation
  have equivalent-states (run n s execs) (run n t (purge execs u)) u
  unfolding view-partitioned-ind-def by auto
 from this ifp-reflexive
  have run n s execs ∥ run n t (purge execs u) ⇀ (λrs rt. vpeq u rs rt ∧ current rs = current rt)
  using r-into-rtranclp unfolding B-def
  by(cases run n s execs,simp,cases run n t (purge execs u),simp,auto)
}
thus ?thesis unfolding view-partitioned-def Let-def by auto
qed
```

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

```
theorem unwinding-implies-NI-unrelated:
shows NI-unrelated
proof–
 {
  fix execs a n
  from unwinding-implies-view-partitioned
   have vp: view-partitioned by blast
  from vp and vpeq-reflexive
   have 1: ∀ u . (run n (Some s0) execs
                ∥ run n (Some s0) (purge execs u)
                  ⇀ (λrs rt. vpeq u rs rt ∧ current rs = current rt))
   unfolding view-partitioned-def by auto
  have run n (Some s0) execs ⇀ (λs-f. run n (Some s0) (purge execs (current s-f)) ⇀ (λs-f2. output-f s-f a =
output-f s-f2 a ∧ current s-f = current s-f2))
  proof(cases run n (Some s0) execs)
  case None
   thus ?thesis unfolding B-def by simp
  next
  case (Some rs)
   thus ?thesis
   proof(cases run n (Some s0) (purge execs (current rs)))
   case None
    from Some this show ?thesis unfolding B-def by simp
   next
   case (Some rt)
    from ‹run n (Some s0) execs = Some rs› Some 1[THEN spec,where x=current rs]
     have vpeq: vpeq (current rs) rs rt ∧ current rs = current rt
     unfolding B-def by auto
    from this output-consistent have output-f rs a = output-f rt a
      by auto
    from this vpeq ‹run n (Some s0) execs = Some rs› Some
     show ?thesis unfolding B-def by auto
   qed
  qed
 }
 thus ?thesis unfolding NI-unrelated-def by auto
qed
```

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–
{
 fix v
 assume involved: v ∈ involved (next-action s execs)
 from this prec-s involved-ifp[THEN spec,THEN spec,where x1=next-state s execs and x=next-action s execs]
  have ifp-v-curr: ifp v (current s)
  using current-next-state
  unfolding involved-def precondition-def B-def
   by(cases next-action s execs,auto)
 have vpeq v s t′
 proof–
 {
  assume ifp v u ∧ ¬intermediary v u
  from this vpeq-s-t
   have vpeq v s t′ by (auto)
 }
 moreover
 {
  assume not-intermediary-v: intermediary v u
  from ifp-curr not-intermediary-curr ifp-v-curr not-intermediary-v have curr-is-u: current s = u
   using rtranclp-trans r-into-rtranclp
   by (metis intermediary-def)
   from curr-is-u next-action-from-execs[THEN spec,THEN spec,where x=execs and x1=s] not-intermediary-v involved
     no-gateway-comm[unfolded does-not-communicate-with-gateway-def,THEN spec,where x=the (next-action s execs)]
   have False
   unfolding involved-def B-def
   by (cases next-action s execs,auto)
  hence vpeq v s t′ by auto
 }
```

moreover
{
assume *intermediary-v*: ¬ *ifp v u*
from *ifp-curr not-intermediary-curr ifp-v-curr intermediary-v*
have *False* **unfolding** *intermediary-def* **by** *auto*
hence *vpeq v s t′* **by** *auto*
}
ultimately
show *vpeq v s t′* **unfolding** *intermediary-def* **by** *auto*
qed
}
thus *?thesis* **by** *auto*
qed

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

lemma *ipurge-l-removes-gateway-communications*:
shows *does-not-communicate-with-gateway u* (*ipurge-l execs u*)
proof–
{
fix *aseq u execs a v*
assume *1*: *aseq* ∈ *set* (*remove-gateway-communications u* (*execs u*))
assume *2*: *a* ∈ *set aseq*
assume *3*: *intermediary v u*
have *4*: *v* ∉ *involved* (*Some a*)
proof–
{
fix *a*::*′action-t*
fix *aseq u exec v*
have *aseq* ∈ *set* (*remove-gateway-communications u exec*) ∧ *a* ∈ *set aseq* ∧ *intermediary v u* ⟶ *v* ∉ *involved* (*Some a*)
by(*induct exec*,*auto*)
}
from *1 2 3 this* **show** *?thesis* **by** *metis*
qed
}
from *this*
show *?thesis*
unfolding *does-not-communicate-with-gateway-def ipurge-l-def actions-in-execution-def*
by *auto*
qed

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

lemma *iunwinding-implies-view-partitioned1*:
shows *iview-partitioned*
proof–
{
fix *u execs execs2 s t n*
have *does-not-communicate-with-gateway u execs* ∧ *iequivalent-states s t u* ∧ *ipurged-relation1 u execs execs2* ⟶ *iequivalent-states* (*run n s execs*) (*run n t execs2*) *u*
proof (*induct n s execs arbitrary*: *t u execs2 rule*: *run.induct*)
case (*1 s execs t u execs2*)
show *?case* **by** *auto*
next
case (*2 n execs t u execs2*)
show *?case* **by** *simp*
next
case (*3 n s execs t u execs2*)

assume interrupt-s: interrupt (Suc n)
assume IH: (⋀t u execs2. does-not-communicate-with-gateway u execs ∧
iequivalent-states (Some (cswitch (Suc n) s)) t u ∧ ipurged-relation1 u execs execs2 ⟶
iequivalent-states (run n (Some (cswitch (Suc n) s)) execs) (run n t execs2) u)
{
fix t′ :: ′state-t
assume t = Some t′
fix rs
assume rs: run (Suc n) (Some s) execs = Some rs
fix rt
assume rt: run (Suc n) (Some t′) execs2 = Some rt

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

from current-s-t cswitch-independent-of-state
have current-ns-nt: current (cswitch (Suc n) s) = current (cswitch (Suc n) t′)
by blast
from cswitch-consistency vpeq-s-t
have vpeq-ns-nt: ∀ v . ifp v u ∧ ¬intermediary v u ⟶ vpeq v (cswitch (Suc n) s) (cswitch (Suc n) t′)
by auto
from no-gateway-comm current-ns-nt vpeq-ns-nt interrupt-s vpeq-reflexive current-s-t purged-a-a2 IH[where u=u and t=Some (cswitch (Suc n) t′) and ?execs2.0=execs2]
have current-rs-rt: current rs = current rt using rs rt by(auto)
{
fix v
assume ia: ifp v u ∧ ¬intermediary v u
from no-gateway-comm interrupt-s current-ns-nt vpeq-ns-nt vpeq-reflexive ia current-s-t purged-a-a2 IH[where u=u and t=Some (cswitch (Suc n) t′) and ?execs2.0=execs2]
have vpeq v rs rt using rs rt by(auto)
}
from current-rs-rt and this have iequivalent-states (Some rs) (Some rt) u by auto
}
thus ?case by(simp add:option.splits,cases t,simp+)
next
case (4 n execs s t u execs2)
assume not-interrupt: ¬interrupt (Suc n)
assume thread-empty-s: thread-empty(execs (current s))

assume IH: (⋀t u execs2. does-not-communicate-with-gateway u execs ∧ iequivalent-states (Some s) t u ∧ ipurged-relation1 u execs execs2 ⟶ iequivalent-states (run n (Some s) execs) (run n t execs2) u)
{
fix t′

assume t: t = Some t′
fix rs
assume rs: run (Suc n) (Some s) execs = Some rs
fix rt
assume rt: run (Suc n) (Some t′) execs2 = Some rt

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

from *ifp-reflexive vpeq-s-t* **have** *vpeq-u-s-t*: *vpeq u s t′* **unfolding** *intermediary-def* **by** *auto*
from *step-atomicity current-next-state current-s-t* **have** *current-s-nt*: *current s = current (step (next-state t′ execs2) (next-action t′ execs2))*
unfolding *step-def*
by (*cases next-action s execs,cases next-action t′ execs2,simp,simp,cases next-action t′ execs2,simp,simp*)
from *vpeq-s-t* **have** *vpeq-curr-s-t*: *ifp (current s) u ∧ ¬intermediary (current s) u ⟶ vpeq (current s) s t′* **by** *auto*
have *iequivalent-states (run (Suc n) (Some s) execs) (run (Suc n) (Some t′) execs2) u*
proof(*cases thread-empty*(*execs2 (current t′)*))
case *True*
from *purged-a-a2* **and** *vpeq-s-t* **and** *current-s-t IH*[**where** *t=Some t′* **and** *u=u* **and** *?execs2.0=execs2*] *no-gateway-comm*
have *iequivalent-states (run n (Some s) execs) (run n (Some t′) execs2) u* **using** *rs rt* **by**(*auto*)
from *this not-interrupt True thread-empty-s*
show *?thesis* **using** *rs rt* **by**(*auto*)
next
case *False*
have *prec-t*: *precondition (next-state t′ execs2) (next-action t′ execs2)*
proof–
{
assume *not-prec-t*: *¬precondition (next-state t′ execs2) (next-action t′ execs2)*
hence *run (Suc n) (Some t′) execs2 = None* **using** *not-interrupt False not-prec-t* **by** (*simp*)
from *this* **have** *False* **using** *rt* **by**(*simp add:option.splits*)
}
thus *?thesis* **by** *auto*
qed

from *False purged-a-a2 thread-empty-s current-s-t*
have *1*: *ind-source (current t′) u ∨ unrelated (current t′) u* **unfolding** *ipurged-relation1-def intermediary-def* **by** *auto*
{
fix *v*
assume *ifp-v*: *ifp v u*
assume *v-not-intermediary*: *¬intermediary v u*

from *1 ifp-v v-not-intermediary* **have** *not-ifp-curr-v*: *¬ifp (current t′) v* **unfolding** *intermediary-def* **by** *auto*
from *not-ifp-curr-v prec-t locally-respects*[*THEN spec,THEN spec,THEN spec*,**where** *x1=next-state t′ execs2* **and** *x=v* **and** *x2=the (next-action t′ execs2)*]
current-next-state vpeq-reflexive
have *vpeq v (next-state t′ execs2) (step (next-state t′ execs2) (next-action t′ execs2))*
unfolding *step-def precondition-def B-def*
by (*cases next-action t′ execs2,auto*)
from *this vpeq-transitive not-ifp-curr-v locally-respects-next-state*
have *vpeq-t-nt*: *vpeq v t′ (step (next-state t′ execs2) (next-action t′ execs2))*
by *blast*
from *vpeq-s-t ifp-v v-not-intermediary vpeq-t-nt vpeq-transitive vpeq-symmetric vpeq-reflexive*
have *vpeq v s (step (next-state t′ execs2) (next-action t′ execs2))*
by (*metis*)
}
hence *vpeq-ns-nt*: *∀ v . ifp v u ∧ ¬intermediary v u ⟶ vpeq v s (step (next-state t′ execs2) (next-action t′ execs2))* **by** *auto*
from *False purged-a-a2 current-s-t thread-empty-s* **have** *purged-a-na2*: *ipurged-relation1 u execs (next-execs t′ execs2)*
unfolding *ipurged-relation1-def next-execs-def* **by**(*auto*)
from *vpeq-ns-nt no-gateway-comm*
and *IH*[**where** *t=Some (step (next-state t′ execs2) (next-action t′ execs2))* **and** *?execs2.0=(next-execs t′ execs2)* **and** *u=u*]

and *current-s-nt purged-a-na2*
have *eq-ns-nt*: *iequivalent-states* (*run n* (*Some s*) *execs*)
(*run n* (*Some* (*step* (*next-state t′ execs2*) (*next-action t′ execs2*))) (*next-execs t′ execs2*)) *u* **by** *auto*
from *prec-t eq-ns-nt not-interrupt False thread-empty-s*
show *?thesis* **using** *t rs rt* **by**(*auto*)
qed
}
thus *?case* **by**(*simp add*:*option.splits*,*cases t*,*simp+*)
next
case (*5 n execs s t u execs2*)
assume *not-interrupt*: ¬*interrupt* (*Suc n*)
assume *thread-not-empty-s*: ¬*thread-empty*(*execs* (*current s*))
assume *not-prec-s*: ¬ *precondition* (*next-state s execs*) (*next-action s execs*)
hence *run* (*Suc n*) (*Some s*) *execs* = *None* **using** *not-interrupt thread-not-empty-s* **by** *simp*
thus *?case* **by**(*simp add*:*option.splits*)
next
case (*6 n execs s t u execs2*)
assume *not-interrupt*: ¬*interrupt* (*Suc n*)
assume *thread-not-empty-s*: ¬*thread-empty*(*execs* (*current s*))
assume *prec-s*: *precondition* (*next-state s execs*) (*next-action s execs*)
assume *IH*: (⋀*t u execs2*. *does-not-communicate-with-gateway u* (*next-execs s execs*) ∧
iequivalent-states (*Some* (*step* (*next-state s execs*) (*next-action s execs*))) *t u* ∧
ipurged-relation1 u (*next-execs s execs*) *execs2* ⟶
iequivalent-states
(*run n* (*Some* (*step* (*next-state s execs*) (*next-action s execs*))) (*next-execs s execs*))
(*run n t execs2*) *u*)
{
fix *t′*
assume *t*: *t* = *Some t′*
fix *rs*
assume *rs*: *run* (*Suc n*) (*Some s*) *execs* = *Some rs*
fix *rt*
assume *rt*: *run* (*Suc n*) (*Some t′*) *execs2* = *Some rt*

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

from *ifp-reflexive vpeq-s-t* **have** *vpeq-u-s-t*: *vpeq u s t′* **unfolding** *intermediary-def* **by** *auto*
from *step-atomicity* **and** *current-s-t current-next-state*
have *current-ns-nt*: *current* (*step* (*next-state s execs*) (*next-action s execs*)) = *current* (*step* (*next-state t′ execs2*) (*next-action t′ execs2*))
unfolding *step-def*
by (*cases next-action s execs*,*cases next-action t′ execs2*,*simp*,*simp*,*cases next-action t′ execs2*,*simp*,*simp*)

from *step-atomicity current-next-state current-s-t* **have** *current-ns-t*: *current* (*step* (*next-state s execs*) (*next-action s execs*)) = *current t′*
unfolding *step-def*
by (*cases next-action s execs*,*auto*)
from *vpeq-s-t* **have** *vpeq-curr-s-t*: *ifp* (*current s*) *u* ∧ ¬*intermediary* (*current s*) *u* ⟶ *vpeq* (*current s*) *s t′* **unfolding** *intermediary-def* **by** *auto*
from *current-s-t purged-a-a2*
have *eq-execs*: *ifp* (*current s*) *u* ∧ ¬*intermediary* (*current s*) *u* ⟶ *execs* (*current s*) = *execs2* (*current s*)
by(*auto simp add*: *ipurged-relation1-def*)
from *vpeq-involved-domains no-gateway-comm vpeq-s-t vpeq-involved-domains prec-s*

have *vpeq-involved*: *ifp* (*current s*) *u* ∧ ¬*intermediary* (*current s*) *u* ⟶ (∀ *d* ∈ *involved* (*next-action s execs*) . *vpeq d s t*′)
by *blast*
from *current-s-t next-execs-consistent*[*THEN spec*,*THEN spec*,*THEN spec*,**where** *x2*=*s* **and** *x1*=*t*′ **and** *x*=*execs*] *vpeq-curr-s-t vpeq-involved*
have *next-execs-t*: *ifp* (*current s*) *u* ∧ ¬*intermediary* (*current s*) *u* ⟶ *next-execs t*′ *execs* = *next-execs s execs*
by(*auto simp add*: *next-execs-def*)
from *current-s-t* **and** *purged-a-a2* **and** *thread-not-empty-s next-action-consistent*[*THEN spec*,*THEN spec*,**where** *x1*=*s* **and** *x*=*t*′] *vpeq-curr-s-t vpeq-involved*
have *next-action-s-t*: *ifp* (*current s*) *u* ∧ ¬*intermediary* (*current s*) *u* ⟶ *next-action t*′ *execs2* = *next-action s execs*
by(*unfold next-action-def*,*unfold ipurged-relation1-def*,*auto*)
from *purged-a-a2* **and** *thread-not-empty-s* **and** *current-s-t*
have *thread-not-empty-t*: *ifp* (*current s*) *u* ∧ ¬*intermediary* (*current s*) *u* ⟶ ¬*thread-empty*(*execs2* (*current t*′))
unfolding *ipurged-relation1-def* **by** *auto*
have *vpeq-ns-nt-1*: ⋀ *a* . *precondition* (*next-state s execs*) *a* ∧ *precondition* (*next-state t*′ *execs*) *a* ⟹ *ifp* (*current s*) *u* ∧ ¬*intermediary* (*current s*) *u* ⟹ (∀ *v* . *ifp v u* ∧ ¬*intermediary v u* ⟶ *vpeq v* (*step* (*next-state s execs*) *a*) (*step* (*next-state t*′ *execs*) *a*))
proof–
fix *a*
assume *precs*: *precondition* (*next-state s execs*) *a* ∧ *precondition* (*next-state t*′ *execs*) *a*
assume *ifp-curr*: *ifp* (*current s*) *u* ∧ ¬*intermediary* (*current s*) *u*
from *ifp-curr precs*
next-state-consistent[*THEN spec*,*THEN spec*,**where** *x1*=*s* **and** *x*=*t*′] *vpeq-curr-s-t vpeq-s-t*
current-next-state current-s-t weakly-step-consistent[*THEN spec*,*THEN spec*,*THEN spec*,*THEN spec*,**where** *x3*=*next-state s execs* **and** *x2*=*next-state t*′ *execs* **and** *x*=*the a*]
show ∀ *v* . *ifp v u* ∧ ¬*intermediary v u* ⟶ *vpeq v* (*step* (*next-state s execs*) *a*) (*step* (*next-state t*′ *execs*) *a*)
unfolding *step-def precondition-def B-def*
by (*cases a*,*auto*)
qed
have *no-gateway-comm-na*: *does-not-communicate-with-gateway u* (*next-execs s execs*)
proof–
{
fix *a*
assume *a* ∈ *actions-in-execution* (*next-execs s execs u*)
from *this no-gateway-comm*[*unfolded does-not-communicate-with-gateway-def*,*THEN spec*,**where** *x*=*a*]
next-execs-subset[*THEN spec*,*THEN spec*,*THEN spec*,**where** *x2*=*s* **and** *x1*=*execs* **and** *x0*=*u*]
have ∀ *v*. *intermediary v u* ⟶ *v* ∉ *involved* (*Some a*)
unfolding *actions-in-execution-def*
by(*auto*)
}
thus *?thesis* **unfolding** *does-not-communicate-with-gateway-def* **by** *auto*
qed
have *iequivalent-states* (*run* (*Suc n*) (*Some s*) *execs*) (*run* (*Suc n*) (*Some t*′) *execs2*) *u*
proof (*cases ifp* (*current s*) *u* ∧ ¬*intermediary* (*current s*) *u rule* :*case-split*[*case-names T F*])
case *T*
show *?thesis*
proof (*cases thread-empty*(*execs2* (*current t*′)) *rule* :*case-split*[*case-names T2 F2*])
case *F2*
show *?thesis*
proof (*cases precondition* (*next-state t*′ *execs2*) (*next-action t*′ *execs2*) *rule* :*case-split*[*case-names T3 F3*])

case *T3*
from *T purged-a-a2 current-s-t*
next-execs-consistent[*THEN spec*,*THEN spec*,**where** *x1*=*s* **and** *x*=*t*′] *vpeq-curr-s-t vpeq-involved*
have *purged-na-na2*: *ipurged-relation1 u* (*next-execs s execs*) (*next-execs t*′ *execs2*)

```
          unfolding ipurged-relation1-def next-execs-def
          by auto
         from IH[where t=Some (step (next-state t′ execs2) (next-action t′ execs2)) and ?execs2.0=next-execs t′
execs2 and u=u]
          purged-na-na2 current-ns-nt vpeq-ns-nt-1[where a=(next-action s execs)] T T3 prec-s
          next-action-s-t  eq-execs current-s-t no-gateway-comm-na
        have eq-ns-nt: iequivalent-states (run n (Some (step (next-state s execs) (next-action s execs))) (next-execs
s execs))
                                    (run n (Some (step (next-state t′ execs2) (next-action t′ execs2))) (next-execs t′
execs2)) u
          unfolding next-state-def
          by (auto,metis)
        from this not-interrupt thread-not-empty-s prec-s F2 T3
          have current-rs-rt: current rs = current rt using rs rt by auto
        {
          fix v
          assume ia: ifp v u ∧ ¬intermediary v u
          from this eq-ns-nt not-interrupt thread-not-empty-s prec-s F2 T3
            have vpeq v rs rt using rs rt by auto
        }
        from this and current-rs-rt show ?thesis using rs rt by auto
      next
      case F3
        from F3 F2 not-interrupt show ?thesis using rt by simp
      qed
    next
    case T2
      from T2 T purged-a-a2 thread-not-empty-s current-s-t prec-s next-action-s-t vpeq-u-s-t
        have ind-source: False unfolding  ipurged-relation1-def by auto
      thus ?thesis by auto
    qed
  next
  case F
    hence 1: ind-source (current s) u ∨ unrelated (current s) u ∨ intermediary (current s) u
      unfolding intermediary-def
      by auto
    from purged-a-a2 and thread-not-empty-s
      have 2: ¬intermediary (current s) u unfolding ipurged-relation1-def by auto

    let ?nt = if thread-empty(execs2 (current t′))  then t′ else step (next-state t′ execs2) (next-action t′ execs2)
    let ?na2 = if thread-empty(execs2 (current t′)) then execs2 else next-execs t′ execs2

     have prec-t: ¬thread-empty(execs2 (current t′)) ⟹ precondition (next-state t′ execs2) (next-action t′
execs2)
    proof−
      assume thread-not-empty-t: ¬thread-empty(execs2 (current t′))
      {
        assume not-prec-t: ¬precondition (next-state t′ execs2) (next-action t′ execs2)
        hence run (Suc n) (Some t′) execs2 = None using not-interrupt thread-not-empty-t not-prec-t by (simp)
        from this  have False using rt by(simp add:option.splits)
      }
      thus ?thesis by auto
    qed

    show ?thesis
    proof−
      {
```

```
fix v
assume ifp-v: ifp v u
assume v-not-intermediary: ¬intermediary v u

have not-ifp-curr-v: ¬ifp (current s) v
proof
  assume ifp-curr-v: ifp (current s) v
  thus False
  proof–
   {
    assume ind-source (current s) u
    from this ifp-curr-v ifp-v have intermediary v u unfolding intermediary-def by auto
    from this v-not-intermediary have False unfolding intermediary-def by auto
   }
   moreover
   {
    assume unrelated: unrelated (current s) u
    from this ifp-v ifp-curr-v have False using rtranclp-trans r-into-rtranclp by metis
   }
   ultimately show ?thesis using 1 2 by auto
  qed
qed
from this current-next-state[THEN spec,THEN spec,where x1=s and x=execs] prec-s
   locally-respects[THEN spec,THEN spec,where x=next-state s execs] vpeq-reflexive
   have vpeq v (next-state s execs) (step (next-state s execs) (next-action s execs))
   unfolding step-def precondition-def B-def
   by (cases next-action s execs,auto)
from not-ifp-curr-v this locally-respects-next-state vpeq-transitive
  have vpeq-s-ns: vpeq v s (step (next-state s execs) (next-action s execs))
  by blast
from not-ifp-curr-v current-s-t current-next-state[THEN spec,THEN spec,where x1=t′ and x=execs2] prec-t
   locally-respects[THEN spec,THEN spec,where x=next-state t′ execs2]
   F vpeq-reflexive
   have 0: ¬ thread-empty (execs2 (current t′)) ⟶ vpeq v (next-state t′ execs2) (step (next-state t′ execs2) (next-action t′ execs2))
   unfolding step-def precondition-def B-def
   by (cases next-action t′ execs2,auto)
    from 0 not-ifp-curr-v current-s-t locally-respects-next-state[THEN spec,THEN spec,THEN spec,where x2=t′ and x1=v and x=execs2]
   vpeq-transitive
  have vpeq-t-nt: ¬ thread-empty (execs2 (current t′)) ⟶ vpeq v t′ (step (next-state t′ execs2) (next-action t′ execs2)) by metis
from this vpeq-reflexive
  have vpeq-t-nt: vpeq v t′ ?nt
  by auto
from vpeq-s-t ifp-v v-not-intermediary
  have vpeq v s t′ by auto
from this vpeq-s-ns vpeq-t-nt vpeq-transitive vpeq-symmetric vpeq-reflexive
  have vpeq v (step (next-state s execs) (next-action s execs)) ?nt
  by (metis (opaque-lifting, no-types))
}
 hence vpeq-ns-nt: ∀ v . ifp v u ∧ ¬intermediary v u ⟶ vpeq v (step (next-state s execs) (next-action s execs)) ?nt by auto
  from vpeq-s-t 2 F purged-a-a2 current-s-t thread-not-empty-s have purged-na-na2: ipurged-relation1 u (next-execs s execs) ?na2
  unfolding ipurged-relation1-def next-execs-def intermediary-def by(auto)
from current-ns-nt current-ns-t current-next-state have current-ns-nt:
```

current (*step* (*next-state s execs*) (*next-action s execs*)) = *current ?nt*
by *auto*
from *prec-s vpeq-ns-nt no-gateway-comm-na*
and *IH*[**where** *t=Some ?nt* **and** *?execs2.0=?na2* **and** *u=u*]
and *current-ns-nt purged-na-na2*
have *eq-ns-nt*: *iequivalent-states* (*run n* (*Some* (*step* (*next-state s execs*) (*next-action s execs*))) (*next-execs s execs*))
(*run n* (*Some ?nt*) *?na2*) *u* **by** *auto*

from *this not-interrupt thread-not-empty-s prec-t prec-s*
have *current-rs-rt*: *current rs* = *current rt* **using** *rs rt* **by** (*cases thread-empty* (*execs2* (*current t′*)),*simp*,*simp*)
{
fix *v*
assume *ia*: *ifp v u* ∧ ¬*intermediary v u*
from *this eq-ns-nt not-interrupt thread-not-empty-s prec-s prec-t*
have *vpeq v rs rt*
using *rs rt* **by** (*cases thread-empty*(*execs2* (*current t′*)),*simp*,*simp*)
}
from *current-rs-rt* **and** *this* **show** *?thesis* **using** *rs rt* **by** *auto*
qed
qed
}
thus *?case* **by**(*simp add*:*option.splits*,*cases t*,*simp+*)
qed
}
hence *iview-partitioned-inductive*: ∀ *u s t execs execs2 n*. *does-not-communicate-with-gateway u execs* ∧ *iequivalent-states s t u* ∧ *ipurged-relation1 u execs execs2* ⟶ *iequivalent-states* (*run n s execs*) (*run n t execs2*) *u*
by *blast*
have *ipurged-relation*: ∀ *u execs* . *ipurged-relation1 u* (*ipurge-l execs u*) (*ipurge-r execs u*)
by(*unfold ipurged-relation1-def* ,*unfold ipurge-l-def* ,*unfold ipurge-r-def* ,*auto*)
{
fix *execs s t n u*
assume *1*: *iequivalent-states s t u*
from *ifp-reflexive*
have *dir-source*: ∀ *u* . *ifp u u* ∧ ¬*intermediary u u* **unfolding** *intermediary-def* **by** *auto*
from *ipurge-l-removes-gateway-communications*
have *does-not-communicate-with-gateway u* (*ipurge-l execs u*)
by *auto*
from *1 this iview-partitioned-inductive ipurged-relation*
have *iequivalent-states* (*run n s* (*ipurge-l execs u*)) (*run n t* (*ipurge-r execs u*)) *u* **by** *auto*
from *this dir-source*
have *run n s* (*ipurge-l execs u*) ∥ *run n t* (*ipurge-r execs u*) ⇀ (λ*rs rt*. *vpeq u rs rt* ∧ *current rs* = *current rt*)
using *r-into-rtranclp* **unfolding** *B-def*
by(*cases run n s* (*ipurge-l execs u*),*simp*,*cases run n t* (*ipurge-r execs u*),*simp*,*auto*)
}
thus *?thesis* **unfolding** *iview-partitioned-def Let-def* **by** *auto*
qed

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

definition *mcurrents* :: ′*state-t option* ⇒ ′*state-t option* ⇒ *bool*
where *mcurrents m1 m2* ≡ *m1* ∥ *m2* ⇀ (λ *s t* . *current s* = *current t*)

Proof that switching/interrupts are purely time-based and happen independent of the actions done by the domains. As all theorems in this locale, it holds vacuously whenever one of the states is None, i.e., 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–
{
fix *n s execs t execs2*
have *mcurrents s t* ⟶ *mcurrents* (*run n s execs*) (*run n t execs2*)
proof (*induct n s execs arbitrary*: *t execs2 rule*: *run.induct*)
case (*1 s execs t execs2*)
from *this* **show** *?case* **using** *current-s-t* **unfolding** *B-def* **by** *auto*
next
case (*2 n execs t execs2*)
show *?case* **unfolding** *mcurrents-def* **by**(*auto*)
next
case (*3 n s execs t execs2*)
assume *interrupt*: *interrupt* (*Suc n*)
assume *IH*: (⋀*t execs2*. *mcurrents* (*Some* (*cswitch* (*Suc n*) *s*)) *t* ⟶ *mcurrents* (*run n* (*Some* (*cswitch* (*Suc n*) *s*)) *execs*) (*run n t execs2*))
{
fix *t′*
assume *t*: *t* = (*Some t′*)
assume *curr*: *mcurrents* (*Some s*) *t*
from *t curr cswitch-independent-of-state*[*THEN spec*,*THEN spec*,*THEN spec*,**where** *x1*=*s*] **have** *current-ns-nt*: *current* (*cswitch* (*Suc n*) *s*) = *current* (*cswitch* (*Suc n*) *t′*)
unfolding *mcurrents-def* **by** *simp*
from *current-ns-nt IH*[**where** *t*=*Some* (*cswitch* (*Suc n*) *t′*) **and** *?execs2.0*=*execs2*]
have *mcurrents-ns-nt*: *mcurrents* (*run n* (*Some* (*cswitch* (*Suc n*) *s*)) *execs*) (*run n* (*Some* (*cswitch* (*Suc n*) *t′*)) *execs2*)
unfolding *mcurrents-def* **by**(*auto*)
from *mcurrents-ns-nt interrupt t*
have *mcurrents* (*run* (*Suc n*) (*Some s*) *execs*) (*run* (*Suc n*) *t execs2*)
unfolding *mcurrents-def B2-def B-def* **by**(*cases run n* (*Some* (*cswitch* (*Suc n*) *s*)) *execs*, *cases run* (*Suc n*) *t execs2*,*auto*)
}
thus *?case* **unfolding** *mcurrents-def B2-def* **by**(*cases t*,*auto*)
next
case (*4 n execs s t execs2*)
assume *not-interrupt*: ¬*interrupt* (*Suc n*)
assume *thread-empty-s*: *thread-empty*(*execs* (*current s*))
assume *IH*: (⋀*t execs2*. *mcurrents* (*Some s*) *t* ⟶ *mcurrents* (*run n* (*Some s*) *execs*) (*run n t execs2*))
{
fix *t′*
assume *t*: *t* = (*Some t′*)
assume *curr*: *mcurrents* (*Some s*) *t*
{
assume *thread-empty-t*: *thread-empty*(*execs2* (*current t′*))
from *t curr not-interrupt thread-empty-s this IH*[**where** *?execs2.0*=*execs2* **and** *t*=*Some t′*]
have *mcurrents* (*run* (*Suc n*) (*Some s*) *execs*) (*run* (*Suc n*) *t execs2*)
by *auto*
}
moreover
{
assume *not-prec-t*: ¬*thread-empty*(*execs2* (*current t′*)) ∧ ¬*precondition* (*next-state t′ execs2*) (*next-action t′ execs2*)
from *t this not-interrupt*
have *mcurrents* (*run* (*Suc n*) (*Some s*) *execs*) (*run* (*Suc n*) *t execs2*)
unfolding *mcurrents-def* **by** (*simp add*: *rewrite-B2-cases*)

}
moreover
{
assume *step-t*: ¬*thread-empty*(*execs2* (*current t′*)) ∧ *precondition* (*next-state t′ execs2*) (*next-action t′ execs2*)
have *mcurrents* (*Some s*) (*Some* (*step* (*next-state t′ execs2*) (*next-action t′ execs2*)))
using *step-atomicity curr t current-next-state* **unfolding** *mcurrents-def*
unfolding *step-def*
by (*cases next-action t′ execs2*,*auto*)
from *t step-t curr not-interrupt thread-empty-s this IH*[**where** *?execs2.0=next-execs t′ execs2* **and** *t=Some* (*step* (*next-state t′ execs2*) (*next-action t′ execs2*))]
have *mcurrents* (*run* (*Suc n*) (*Some s*) *execs*) (*run* (*Suc n*) *t execs2*)
by *auto*
}
ultimately have *mcurrents* (*run* (*Suc n*) (*Some s*) *execs*) (*run* (*Suc n*) *t execs2*) **by** *blast*
}
thus *?case* **unfolding** *mcurrents-def B2-def* **by**(*cases t*,*auto*)
next
case (*5 n execs s t execs2*)
assume *not-interrupt-s*: ¬*interrupt* (*Suc n*)
assume *thread-not-empty-s*: ¬*thread-empty*(*execs* (*current s*))
assume *not-prec-s*: ¬ *precondition* (*next-state s execs*) (*next-action s execs*)
hence *run* (*Suc n*) (*Some s*) *execs* = *None* **using** *not-interrupt-s thread-not-empty-s* **by** *simp*
thus *?case* **unfolding** *mcurrents-def* **by**(*simp add*:*option.splits*)
next
case (*6 n execs s t execs2*)
assume *not-interrupt*: ¬*interrupt* (*Suc n*)
assume *thread-not-empty-s*: ¬*thread-empty*(*execs* (*current s*))
assume *prec-s*: *precondition* (*next-state s execs*) (*next-action s execs*)
assume *IH*: (⋀*t execs2*.
mcurrents (*Some* (*step* (*next-state s execs*) (*next-action s execs*))) *t* ⟶
mcurrents (*run n* (*Some* (*step* (*next-state s execs*) (*next-action s execs*))) (*next-execs s execs*)) (*run n t execs2*))
{
fix *t′*
assume *t*: *t* = (*Some t′*)
assume *curr*: *mcurrents* (*Some s*) *t*
{
assume *thread-empty-t*: *thread-empty*(*execs2* (*current t′*))
have *mcurrents* (*Some* (*step* (*next-state s execs*) (*next-action s execs*))) (*Some t′*)
using *step-atomicity curr t current-next-state* **unfolding** *mcurrents-def*
unfolding *step-def*
by (*cases next-action s execs*,*auto*)
from *t curr not-interrupt thread-not-empty-s prec-s thread-empty-t this IH*[**where** *?execs2.0=execs2* **and** *t=Some t′*]
have *mcurrents* (*run* (*Suc n*) (*Some s*) *execs*) (*run* (*Suc n*) *t execs2*)
by *auto*
}
moreover
{
assume *not-prec-t*: ¬*thread-empty*(*execs2* (*current t′*)) ∧ ¬*precondition* (*next-state t′ execs2*) (*next-action t′ execs2*)
from *t this not-interrupt*
have *mcurrents* (*run* (*Suc n*) (*Some s*) *execs*) (*run* (*Suc n*) *t execs2*)
unfolding *mcurrents-def B2-def* **by** (*auto*)
}
moreover

{
assume *step-t*: ¬*thread-empty*(*execs2* (*current t′*)) ∧ *precondition* (*next-state t′ execs2*) (*next-action t′ execs2*)
have *mcurrents* (*Some* (*step* (*next-state s execs*) (*next-action s execs*))) (*Some* (*step* (*next-state t′ execs2*) (*next-action t′ execs2*)))
using *step-atomicity curr t current-next-state* **unfolding** *mcurrents-def*
unfolding *step-def*
by (*cases next-action s execs*,*simp*,*cases next-action t′ execs2*,*simp*,*simp*,*cases next-action t′ execs2*,*simp*,*simp*)
from *current-next-state t step-t curr not-interrupt thread-not-empty-s prec-s this IH*[**where** *?execs2.0*=*next-execs t′ execs2* **and** *t*=*Some* (*step* (*next-state t′ execs2*) (*next-action t′ execs2*))]
have *mcurrents* (*run* (*Suc n*) (*Some s*) *execs*) (*run* (*Suc n*) *t execs2*)
by *auto*
}
ultimately have *mcurrents* (*run* (*Suc n*) (*Some s*) *execs*) (*run* (*Suc n*) *t execs2*) **by** *blast*
}
thus *?case* **unfolding** *mcurrents-def B2-def* **by**(*cases t*,*auto*)
qed
}
thus *?thesis* **using** *current-s-t* **by** *auto*
qed

theorem *unwinding-implies-NI-indirect-sources*:
shows *NI-indirect-sources*
proof–
{
fix *execs a n*
from *iunwinding-implies-view-partitioned1*
have *vp*: *iview-partitioned* **by** *blast*
from *vp* **and** *vpeq-reflexive*
have *1*: ∀ *u* . *run n* (*Some s0*) (*ipurge-l execs u*) ∥ *run n* (*Some s0*) (*ipurge-r execs u*) ⇀ (λ*rs rt*. *vpeq u rs rt* ∧ *current rs* = *current rt*)
unfolding *iview-partitioned-def* **by** *auto*

have *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*))
proof(*cases run n* (*Some s0*) *execs*)
case *None*
thus *?thesis* **unfolding** *B-def* **by** *simp*
next
case (*Some s-f*)
thus *?thesis*
proof(*cases run n* (*Some s0*) (*ipurge-l execs* (*current s-f*)))
case *None*
from *Some this* **show** *?thesis* **unfolding** *B-def* **by** *simp*
next
case (*Some s-ipurge-l*)
show *?thesis*
proof(*cases run n* (*Some s0*) (*ipurge-r execs* (*current s-f*)))
case *None*
from ‹*run n* (*Some s0*) *execs* = *Some s-f*› *Some this* **show** *?thesis* **unfolding** *B-def* **by** *simp*
next
case (*Some s-ipurge-r*)
from *cswitch-independent-of-state*
‹*run n* (*Some s0*) *execs* = *Some s-f*› ‹*run n* (*Some s0*) (*ipurge-l execs* (*current s-f*)) = *Some s-ipurge-l*›
current-independent-of-domain-actions[**where** *n*=*n* **and** *s*=*Some s0* **and** *t*=*Some s0* **and** *execs*=*execs* **and** *?execs2.0*=(*ipurge-l execs* (*current s-f*))]

```
        have 2: current s-ipurge-l = current s-f
        unfolding mcurrents-def B-def by auto
      from ‹run n (Some s0) execs = Some s-f› ‹run n (Some s0) (ipurge-l execs (current s-f)) = Some s-ipurge-l›
          Some 1[THEN spec,where x=current s-f]
        have vpeq (current s-f) s-ipurge-l s-ipurge-r ∧ current s-ipurge-l = current s-ipurge-r
        unfolding B-def by auto
       from this 2 have output-f s-ipurge-l a = output-f s-ipurge-r a
        using output-consistent by auto
      from ‹run n (Some s0) execs = Some s-f› ‹run n (Some s0) (ipurge-l execs (current s-f)) = Some s-ipurge-l›
          this Some
        show ?thesis unfolding B-def by auto
     qed
    qed
   qed
 }
 thus ?thesis unfolding NI-indirect-sources-def by auto
qed


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

end
end
```

3.3 ISK (Interruptible Separation Kernel)

```
theory ISK
 imports SK
begin
```

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

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

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

If action sequence 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

$(aseqs \neq [] \wedge hd\ aseqs \neq [] \wedge \neg aborting\ s'\ d\ (the\ a) \wedge \neg\ waiting\ s'\ d\ (the\ a) \wedge (a,aseqs') = (Some\ (hd\ (hd\ aseqs)), (tl\ (hd\ aseqs))\#(tl\ aseqs))) \vee$ — Execute the first action of the current action sequence
$(aseqs \neq [] \wedge hd\ aseqs \neq [] \wedge waiting\ s'\ d\ (the\ a) \wedge (a,aseqs',s') = (Some\ (hd\ (hd\ aseqs)),aseqs,s)) \vee$ — Nothing happens, waiting to execute the next action
$(a,aseqs') = (None,tl\ aseqs)$
and *next-action-after-cswitch*: $\forall\ s\ n\ d\ aseqs\ .\ fst\ (control\ (cswitch\ n\ s)\ d\ aseqs) = fst\ (control\ s\ d\ aseqs)$
and *next-action-after-next-state*: $\forall\ s\ execs\ d\ .\ current\ s \neq d \longrightarrow fst\ (control\ (next\text{-}state\ s\ execs)\ d\ (execs\ d)) = None \vee fst\ (control\ (next\text{-}state\ s\ execs)\ d\ (execs\ d)) = fst\ (control\ s\ d\ (execs\ d))$
and *next-action-after-step*: $\forall\ s\ a\ d\ aseqs\ .\ current\ s \neq d \longrightarrow fst\ (control\ (step\ s\ a)\ d\ aseqs) = fst\ (control\ s\ d\ aseqs)$
and *next-state-precondition*: $\forall\ s\ d\ a\ execs.\ AS\text{-}precondition\ s\ d\ a \longrightarrow AS\text{-}precondition\ (next\text{-}state\ s\ execs)\ d\ a$
and *next-state-invariant*: $\forall\ s\ execs\ .\ invariant\ s \longrightarrow invariant\ (next\text{-}state\ s\ execs)$
and *spec-of-waiting*: $\forall\ s\ a\ .\ waiting\ s\ (current\ s)\ a \longrightarrow kstep\ s\ a = s$
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* :: $time\text{-}t \Rightarrow 'state\text{-}t \Rightarrow ('dom\text{-}t \Rightarrow 'action\text{-}t\ execution) \Rightarrow 'state\text{-}t$
where $run\text{-}total\ 0\ s\ execs = s$
$|\ interrupt\ (Suc\ n) \Longrightarrow run\text{-}total\ (Suc\ n)\ s\ execs = run\text{-}total\ n\ (cswitch\ (Suc\ n)\ s)\ execs$
$|\ \neg interrupt\ (Suc\ n) \Longrightarrow thread\text{-}empty(execs\ (current\ s)) \Longrightarrow run\text{-}total\ (Suc\ n)\ s\ execs = run\text{-}total\ n\ s\ execs$
$|\ \neg interrupt\ (Suc\ n) \Longrightarrow \neg thread\text{-}empty(execs\ (current\ s)) \Longrightarrow$
$run\text{-}total\ (Suc\ n)\ s\ execs = run\text{-}total\ n\ (step\ (next\text{-}state\ s\ execs)\ (next\text{-}action\ s\ execs))\ (next\text{-}execs\ s\ execs)$
using *not0-implies-Suc* **by** (*metis prod-cases3*,*auto*)
termination by *lexicographic-order*

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

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

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* :: $'action\text{-}t\ list \Rightarrow bool$
where $realistic\text{-}AS\text{-}partial\ aseq \equiv \exists\ n\ aseq'\ .\ n \leq length\ aseq' \wedge aseq' \in AS\text{-}set \wedge aseq = lastn\ n\ aseq'$
definition *realistic-executions-ind* :: $('dom\text{-}t \Rightarrow 'action\text{-}t\ execution) \Rightarrow bool$
where $realistic\text{-}executions\text{-}ind\ execs \equiv \forall\ d\ .\ (case\ execs\ d\ of\ [] \Rightarrow True\ |\ (aseq\#aseqs) \Rightarrow realistic\text{-}AS\text{-}partial\ aseq \wedge set\ aseqs \subseteq AS\text{-}set)$

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* :: $'state\text{-}t \Rightarrow ('dom\text{-}t \Rightarrow 'action\text{-}t\ execution) \Rightarrow bool$
where $precondition\text{-}ind\ s\ execs \equiv invariant\ s \wedge (\forall\ d\ .\ fst(control\ s\ d\ (execs\ d)) \rightharpoonup AS\text{-}precondition\ s\ d)$

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*: $next\text{-}execs\ s\ execs\ d = aseq\ \#\ aseqs$

```
  and d-is-curr: d = current s
  and realistic: realistic-executions-ind execs
  and thread-not-empty: ¬thread-empty(execs (current s))
shows realistic-AS-partial aseq ∧ set aseqs ⊆ AS-set
proof−
let ?c = control s (current s) (execs (current s))
{
 assume c-empty: let (a,aseqs',s') = ?c in
          (a,aseqs') = (None,[])
 from na-def d-is-curr c-empty
  have ?thesis
  unfolding realistic-executions-ind-def next-execs-def by (auto)
}
moreover
{
 let ?ct= execs (current s)
 let ?execs' = (tl (hd ?ct))#(tl ?ct)
 let ?a' = Some (hd (hd ?ct))
 assume hd-thread-not-empty: hd (execs (current s)) ≠ []
 assume c-executing: let (a,aseqs',s') = ?c in
                (a,aseqs') = (?a', ?execs')
 from na-def c-executing d-is-curr
  have as-defs: aseq = tl (hd ?ct) ∧ aseqs = tl ?ct
  unfolding next-execs-def by (auto)
 from realistic[unfolded realistic-executions-ind-def,THEN spec,where x=d] d-is-curr
  have subset: set (tl ?execs') ⊆ AS-set
  unfolding Let-def realistic-AS-partial-def
  by (cases execs d,auto)
 from d-is-curr thread-not-empty hd-thread-not-empty realistic[unfolded realistic-executions-ind-def,THEN spec,where
x=d]
  obtain n aseq' where n-aseq': n ≤ length aseq' ∧ aseq' ∈ AS-set ∧ hd ?ct = lastn n aseq'
  unfolding realistic-AS-partial-def
  by (cases execs d,auto)
 from this hd-thread-not-empty have n > 0 unfolding lastn-def by(cases n,auto)
 from this n-aseq' lastn-one-less[where n=n and x=aseq' and a=hd (hd ?ct) and y=tl (hd ?ct)] hd-thread-not-empty
  have n − 1 ≤ length aseq' ∧ aseq' ∈ AS-set ∧ tl (hd ?ct) = lastn (n − 1) aseq'
  by auto
 from this as-defs subset have ?thesis
  unfolding realistic-AS-partial-def
  by auto
}
moreover
{
 let ?ct= execs (current s)
 let ?execs' = ?ct
 let ?a' = Some (hd (hd ?ct))
 assume c-waiting: let (a,aseqs',s') = ?c in
                (a,aseqs') = (?a', ?execs')
 from na-def c-waiting d-is-curr
  have as-defs: aseq = hd ?execs' ∧ aseqs = tl ?execs'
  unfolding next-execs-def by (auto)
  from realistic[unfolded realistic-executions-ind-def,THEN spec,where x=d] d-is-curr set-tl-is-subset[where
x=?execs']
  have subset: set (tl ?execs') ⊆ AS-set
  unfolding Let-def realistic-AS-partial-def
  by (cases execs d,auto)
 from na-def c-waiting d-is-curr
```

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

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

```
lemma run-total-equals-run:
  assumes realistic-exec: realistic-executions execs
      and invariant: invariant s
    shows strict-equal (run n (Some s) execs) (run-total n s execs)
proof−
{
  fix n ms s execs
   have strict-equal ms s ∧ realistic-executions-ind execs ∧ precondition-ind s execs ⟶ strict-equal (run n ms
execs) (run-total n s execs)
  proof (induct n ms execs arbitrary: s rule: run.induct)
  case (1 s execs sa)
    show ?case by auto
  next
  case (2 n execs s)
```

show *?case* **unfolding** *strict-equal-def* **by** *auto*
next
case (*3 n s execs sa*)
assume *interrupt*: *interrupt* (*Suc n*)
assume *IH*: (⋀*sa*. *strict-equal* (*Some* (*cswitch* (*Suc n*) *s*)) *sa* ∧ *realistic-executions-ind execs* ∧ *precondition-ind sa execs* ⟶
strict-equal (*run n* (*Some* (*cswitch* (*Suc n*) *s*)) *execs*) (*run-total n sa execs*))
{
assume *equal-s-sa*: *strict-equal* (*Some s*) *sa*
assume *realistic*: *realistic-executions-ind execs*
assume *inv-sa*: *precondition-ind sa execs*
have *inv-nsa*: *precondition-ind* (*cswitch* (*Suc n*) *sa*) *execs*
proof–
{
fix *d*
have *fst* (*control* (*cswitch* (*Suc n*) *sa*) *d* (*execs d*)) ⇀ *AS-precondition* (*cswitch* (*Suc n*) *sa*) *d*
using *next-action-after-cswitch inv-sa*[*unfolded precondition-ind-def*,*THEN conjunct2*,*THEN spec*,**where** *x*=*d*]
precondition-after-cswitch
unfolding *Let-def B-def precondition-ind-def*
by(*cases fst* (*control* (*cswitch* (*Suc n*) *sa*) *d* (*execs d*)),*auto*)
}
thus *?thesis* **using** *inv-sa invariant-after-cswitch* **unfolding** *precondition-ind-def* **by** *auto*
qed
from *equal-s-sa realistic inv-nsa inv-sa IH*[**where** *sa*=*cswitch* (*Suc n*) *sa*]
have *equal-ns-nt*: *strict-equal* (*run n* (*Some* (*cswitch* (*Suc n*) *s*)) *execs*) (*run-total n* (*cswitch* (*Suc n*) *sa*) *execs*)
unfolding *strict-equal-def* **by**(*auto*)
}
from *this interrupt* **show** *?case* **by** *auto*
next
case (*4 n execs s sa*)
assume *not-interrupt*: ¬*interrupt* (*Suc n*)
assume *thread-empty*: *thread-empty*(*execs* (*current s*))
assume *IH*: (⋀*sa*. *strict-equal* (*Some s*) *sa* ∧ *realistic-executions-ind execs* ∧ *precondition-ind sa execs* ⟶ *strict-equal* (*run n* (*Some s*) *execs*) (*run-total n sa execs*))
have *current-s-sa*: *strict-equal* (*Some s*) *sa* ⟶ *current s* = *current sa* **unfolding** *strict-equal-def* **by** *auto*
{
assume *equal-s-sa*: *strict-equal* (*Some s*) *sa*
assume *realistic*: *realistic-executions-ind execs*
assume *inv-sa*: *precondition-ind sa execs*
from *equal-s-sa realistic inv-sa IH*[**where** *sa*=*sa*]
have *equal-ns-nt*: *strict-equal* (*run n* (*Some s*) *execs*) (*run-total n sa execs*)
unfolding *strict-equal-def* **by**(*auto*)
}
from *this current-s-sa thread-empty not-interrupt* **show** *?case* **by** *auto*
next
case (*5 n execs s sa*)
assume *not-interrupt*: ¬*interrupt* (*Suc n*)
assume *thread-not-empty*: ¬*thread-empty*(*execs* (*current s*))
assume *not-prec*: ¬ *precondition* (*next-state s execs*) (*next-action s execs*)
— In locale ISK, the precondition can be proven to hold at all times. This case cannot happen, and we can prove False.
{
assume *equal-s-sa*: *strict-equal* (*Some s*) *sa*
assume *realistic*: *realistic-executions-ind execs*
assume *inv-sa*: *precondition-ind sa execs*

from *equal-s-sa* **have** *s-sa*: *s = sa* **unfolding** *strict-equal-def* **by** *auto*
from *inv-sa* **have**
next-action sa execs ⇀ AS-precondition sa (current sa)
unfolding *precondition-ind-def B-def next-action-def*
by (*cases next-action sa execs,auto*)
from *this next-state-precondition*
have *next-action sa execs ⇀ AS-precondition (next-state sa execs) (current sa)*
unfolding *precondition-ind-def B-def*
by (*cases next-action sa execs,auto*)
from *inv-sa this s-sa next-state-invariant current-next-state*
have *prec-s*: *precondition (next-state s execs) (next-action s execs)*
unfolding *precondition-ind-def kprecondition-def precondition-def B-def*
by (*cases next-action sa execs,auto*)
from *this not-prec* **have** *False* **by** *auto*
}
thus *?case* **by** *auto*
next
case (*6 n execs s sa*)
assume *not-interrupt*: *¬interrupt (Suc n)*
assume *thread-not-empty*: *¬thread-empty(execs (current s))*
assume *prec*: *precondition (next-state s execs) (next-action s execs)*
assume *IH*: (*⋀sa. strict-equal (Some (step (next-state s execs) (next-action s execs))) sa ∧*
realistic-executions-ind (next-execs s execs) ∧ precondition-ind sa (next-execs s execs) ⟶
strict-equal (run n (Some (step (next-state s execs) (next-action s execs))) (next-execs s execs)) (run-total n sa (next-execs s execs)))
have *current-s-sa*: *strict-equal (Some s) sa ⟶ current s = current sa* **unfolding** *strict-equal-def* **by** *auto*
{
assume *equal-s-sa*: *strict-equal (Some s) sa*
assume *realistic*: *realistic-executions-ind execs*
assume *inv-sa*: *precondition-ind sa execs*

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

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

have *equal-ns-nsa*: *strict-equal (Some ?ns) ?ns* **unfolding** *strict-equal-def* **by** *auto*
from *inv-sa equal-s-sa* **have** *inv-s*: *invariant s* **unfolding** *strict-equal-def precondition-ind-def* **by** *auto*

— Two things are proven inductive. First, the assumptions that the execution is realistic (statement realistic-na). This proof uses lemma next-execution-is-realistic-partial. Secondly, the precondition: if the precondition holds for the current action, then it holds for the next action (statement invariant-na).
have *realistic-na*: *realistic-executions-ind ?na*
proof–
{
fix *d*
have *case ?na d of [] ⇒ True | aseq # aseqs ⇒ realistic-AS-partial aseq ∧ set aseqs ⊆ AS-set*
proof(*cases ?na d,simp,rename-tac aseq aseqs,simp,cases d = current s*)
case *False*
fix *aseq aseqs*
assume *next-execs s execs d = aseq # aseqs*
from *False this realistic*[*unfolded realistic-executions-ind-def,THEN spec,***where** *x=d*]
show *realistic-AS-partial aseq ∧ set aseqs ⊆ AS-set*
unfolding *next-execs-def* **by** *simp*
next

case *True*
fix *aseq aseqs*
assume *na-def*: *next-execs s execs d* = *aseq* # *aseqs*
from *next-execution-is-realistic-partial na-def True realistic thread-not-empty*
show *realistic-AS-partial aseq* ∧ *set aseqs* ⊆ *AS-set* **by** *blast*
qed
}
thus *?thesis* **unfolding** *realistic-executions-ind-def* **by** *auto*
qed
have *invariant-na*: *precondition-ind ?ns ?na*
proof−
from *spec-of-invariant inv-sa next-state-invariant s-sa* **have** *inv-ns*: *invariant ?ns*
unfolding *precondition-ind-def step-def*
by (*cases next-action sa execs*,*auto*)
have ∀ *d*. *fst* (*control ?ns d* (*?na d*)) ⇀ *AS-precondition ?ns d*
proof−
{
fix *d*
{
let *?a′* = *fst* (*control ?ns d* (*?na d*))
assume *snd-action-not-none*: *?a′* ≠ *None*
have *AS-precondition ?ns d* (*the ?a′*)
proof (*cases d* = *current s*)
case *True*
{
have *?thesis*
proof (*cases ?a*)
case (*Some a*)
— Assuming that the current domain executes some action a, and assuming that the action a' after that is not None (statement snd-action-not-none), we prove that the precondition is inductive, i.e., it will hold for a'. Two cases arise: either action a is delayed (case waiting) or not (case executing).
show *?thesis*
proof(*cases ?na d* = *execs* (*current s*) *rule* :*case-split*[*case-names waiting executing*])
case *executing* — The kernel is executing two consecutive actions a and a'. We show that [a,a'] is a subsequence in some action in AS-set. The PO's ensure that the precondition is inductive.
from *executing True Some control-spec*[*THEN spec*,*THEN spec*,*THEN spec*,**where** *x2*=*s* **and** *x1*=*d* **and** *x*=*execs d*]
have *a-def*: *a* = *hd* (*hd* (*execs* (*current s*))) ∧ *?na d* = (*tl* (*hd* (*execs* (*current s*))))#(*tl* (*execs* (*current s*)))
unfolding *next-action-def next-execs-def Let-def*
by(*auto*)
from *a-def True snd-action-not-none control-spec*[*THEN spec*,*THEN spec*,*THEN spec*,**where** *x2*=*?ns* **and** *x1*=*d* **and** *x*=*?na d*]
second-elt-is-hd-tl[**where** *x*= *hd* (*execs* (*current s*)) **and** *a*=*hd*(*tl*(*hd* (*execs* (*current s*)))) **and** *x′*=*tl* (*tl*(*hd* (*execs* (*current s*))))]
have *na-def*: *the ?a′* = (*hd* (*execs* (*current s*)))!*1*
unfolding *next-execs-def*
by(*auto*)
from *Some realistic*[*unfolded realistic-executions-ind-def* ,*THEN spec*,**where** *x*=*d*] *True thread-not-empty*
obtain *n aseq′* **where** *witness*: *n* ≤ *length aseq′* ∧ *aseq′* ∈ *AS-set* ∧ *hd*(*execs d*) = *lastn n aseq′*
unfolding *realistic-AS-partial-def* **by** (*cases execs d*,*auto*)
from *True executing length-lt-2-implies-tl-empty*[**where** *x*=*hd* (*execs* (*current s*))]
Some control-spec[*THEN spec*,*THEN spec*,*THEN spec*,**where** *x2*=*s* **and** *x1*=*d* **and** *x*=*execs d*]
snd-action-not-none control-spec[*THEN spec*,*THEN spec*,*THEN spec*,**where** *x2*=*?ns* **and** *x1*=*d* **and** *x*=*?na d*]
have *in-action-sequence*: *length* (*hd* (*execs* (*current s*))) ≥ 2
unfolding *next-action-def next-execs-def*

by *auto*
from *this witness consecutive-is-sub-seq*[**where** *a=a* **and** *b=the ?a′* **and** *n=n* **and** *y=aseq′* **and** *x=tl* (*tl* (*hd* (*execs* (*current s*))))]
a-def na-def True in-action-sequence
x-is-hd-snd-tl[**where** *x=hd* (*execs* (*current s*))]
have *1*: ∃ *aseq′* ∈ *AS-set* . *is-sub-seq a* (*the ?a′*) *aseq′*
by(*auto*)
from *True Some inv-sa*[*unfolded precondition-ind-def*,*THEN conjunct2*,*THEN spec*,**where** *x=current s*] *s-sa*
have 2: *AS-precondition s* (*current s*) *a*
unfolding *strict-equal-def next-action-def B-def* **by** *auto*
from *executing True Some control-spec*[*THEN spec*,*THEN spec*,*THEN spec*,**where** *x2=s* **and** *x1=d* **and** *x=execs d*]
have *not-aborting*: ¬*aborting* (*next-state s execs*) (*current s*) (*the ?a*)
unfolding *next-action-def next-state-def next-execs-def*
by *auto*
from *executing True Some control-spec*[*THEN spec*,*THEN spec*,*THEN spec*,**where** *x2=s* **and** *x1=d* **and** *x=execs d*]
have *not-waiting*: ¬*waiting* (*next-state s execs*) (*current s*) (*the ?a*)
unfolding *next-action-def next-state-def next-execs-def*
by *auto*
from *True this*
1 2 inv-s
sub-seq-in-prefixes[**where** *X=AS-set*] *Some next-state-invariant*
current-next-state[*THEN spec*,*THEN spec*,**where** *x1=s* **and** *x=execs*]
AS-prec-after-step[*THEN spec*,*THEN spec*,*THEN spec*,**where** *x2=next-state s execs* **and** *x1=a* **and** *x=the ?a′*]
next-state-precondition not-aborting not-waiting
show *?thesis*
unfolding *step-def*
by *auto*
next
case *waiting* — The kernel is delaying action a. Thus the action after a, which is a', is equal to a.
from *tl-hd-x-not-tl-x*[**where** *x=execs d*] *True waiting control-spec*[*THEN spec*,*THEN spec*,*THEN spec*,**where** *x2=s* **and** *x1=d* **and** *x=execs d*] *Some*
have *a-def*: *?na d* = *execs* (*current s*) ∧ *next-state s execs* = *s* ∧ *waiting s d* (*the ?a*)
unfolding *next-action-def next-execs-def next-state-def*
by(*auto*)
from *Some waiting a-def True snd-action-not-none control-spec*[*THEN spec*,*THEN spec*,*THEN spec*,**where** *x2=?ns* **and** *x1=d* **and** *x=?na d*]
have *na-def*: *the ?a′* = *hd* (*hd* (*execs* (*current s*)))
unfolding *next-action-def next-execs-def*
by(*auto*)
from *spec-of-waiting a-def True*
have *no-step*: *step s ?a* = *s* **unfolding** *step-def* **by** (*cases next-action s execs*,*auto*)
from *no-step Some True a-def*
inv-sa[*unfolded precondition-ind-def*,*THEN conjunct2*,*THEN spec*,**where** *x=current s*] *s-sa*
have 2: *AS-precondition s* (*current s*) (*the ?a′*)
unfolding *next-action-def B-def*
by(*auto*)
from *a-def na-def this True Some no-step*
show *?thesis*
unfolding *step-def*
by(*auto*)
qed
next
case *None*

— Assuming that the current domain does not execute an action, and assuming that the action a' after that is not None (statement snd-action-not-none), we prove that the precondition is inductive, i.e., it will hold for a'. This holds, since the control mechanism will ensure that action a' is the start of a new action sequence in AS-set.

from *None True snd-action-not-none control-spec*[*THEN spec*,*THEN spec*,*THEN spec*,**where** *x2=?ns* **and** *x1=d* **and** *x=?na d*]
control-spec[*THEN spec*,*THEN spec*,*THEN spec*,**where** *x2=s* **and** *x1=d* **and** *x=execs d*]
have *na-def*: *the ?a′ = hd (hd (tl (execs (current s)))) ∧ ?na d = tl (execs (current s))*
unfolding *next-action-def next-execs-def*
by(*auto*)
from *True None snd-action-not-none control-spec*[*THEN spec*,*THEN spec*,*THEN spec*,**where** *x2=?ns* **and** *x1=d* **and** *x=?na d*]
this
have *1*: *tl (execs (current s)) ≠ [] ∧ hd (tl (execs (current s))) ≠ []*
by *auto*
from *this realistic*[*unfolded realistic-executions-ind-def*,*THEN spec*,**where** *x=d*] *True thread-not-empty*
have *hd (tl (execs (current s))) ∈ AS-set*
by (*cases execs d*,*auto*)
from *True snd-action-not-none this*
inv-ns this na-def 1
AS-prec-first-action[*THEN spec*,*THEN spec*,*THEN spec*,**where** *x2=?ns* **and** *x=hd (tl (execs (current s)))* **and** *x1=d*]
show *?thesis* **by** *auto*
qed
}
thus *?thesis*
using *control-spec*[*THEN spec*,*THEN spec*,*THEN spec*,**where** *x2=?ns* **and** *x1=current s* **and** *x=?na (current s)*]
thread-not-empty True snd-action-not-none
by (*auto simp add*: *Let-def*)
next
case *False*
from *False* **have** *equal-na-a*: *?na d = execs d*
unfolding *next-execs-def* **by** *auto*
from *this False current-next-state next-action-after-step*
have *?a′ = fst (control (next-state s execs) d (next-execs s execs d))*
unfolding *next-action-def* **by** *auto*
from *inv-sa*[*unfolded precondition-ind-def*,*THEN conjunct2*,*THEN spec*,**where** *x=d*] *s-sa equal-na-a this*
next-action-after-next-state[*THEN spec*,*THEN spec*,*THEN spec*,**where** *x=d* **and** *x2=s* **and** *x1=execs*]
snd-action-not-none False
have *AS-precondition s d (the ?a′)*
unfolding *precondition-ind-def next-action-def B-def* **by** (*cases fst (control sa d (execs d))*,*auto*)
from *equal-na-a False this next-state-precondition current-next-state*
AS-prec-dom-independent[*THEN spec*,*THEN spec*,*THEN spec*,*THEN spec*,**where** *x3=next-state s execs* **and** *x2=d* **and** *x=the ?a* **and** *x1=the ?a′*]
show *?thesis*
unfolding *step-def*
by (*cases next-action s execs*,*auto*)
qed
}
hence *fst (control ?ns d (?na d)) ⇀ AS-precondition ?ns d* **unfolding** *B-def*
by (*cases fst (control ?ns d (?na d))*,*auto*)
}
thus *?thesis* **by** *auto*
qed
from *this inv-ns* **show** *?thesis*
unfolding *precondition-ind-def B-def Let-def*

```
      by (auto)
    qed
    from equal-ns-nsa realistic-na invariant-na s-sa IH[where sa=?ns]
      have equal-ns-nt: strict-equal (run n (Some ?ns) ?na) (run-total n (step (next-state sa execs) (next-action
sa execs)) (next-execs sa execs))
      by(auto)
  }
  from this current-s-sa thread-not-empty not-interrupt prec show ?case by auto
 qed
}
hence thm-inductive: ∀ m s execs n . strict-equal m s ∧ realistic-executions-ind execs ∧ precondition-ind s execs
⟶ strict-equal (run n m execs) (run-total n s execs) by blast
have 1: strict-equal (Some s) s unfolding strict-equal-def by simp
have 2: realistic-executions-ind execs
 proof–
 {
  fix d
  have case execs d of [] ⇒ True | aseq # aseqs ⇒ realistic-AS-partial aseq ∧ set aseqs ⊆ AS-set
  proof(cases execs d,simp)
  case (Cons aseq aseqs)
   from Cons realistic-exec[unfolded realistic-executions-def,THEN spec,where x=d]
    have 0: length aseq ≤ length aseq ∧ aseq ∈ AS-set ∧ aseq = lastn (length aseq) aseq
    unfolding lastn-def realistic-execution-def by auto
   hence 1: realistic-AS-partial aseq unfolding realistic-AS-partial-def by auto
   from Cons realistic-exec[unfolded realistic-executions-def,THEN spec,where x=d]
    have 2: set aseqs ⊆ AS-set
    unfolding realistic-execution-def by auto
   from Cons 1 2 show ?thesis by auto
  qed
 }
 thus ?thesis unfolding realistic-executions-ind-def by auto
 qed
have 3: precondition-ind s execs
 proof–
  {
   fix d
   {
   assume not-empty: fst (control s d (execs d)) ≠ None
   from not-empty realistic-exec[unfolded realistic-executions-def,THEN spec,where x=d]
    have current-aseq-is-realistic: hd (execs d) ∈ AS-set
    using control-spec[THEN spec,THEN spec,THEN spec,where x=execs d and x1=d and x2=s]
    unfolding realistic-execution-def by(cases execs d,auto)
    from not-empty current-aseq-is-realistic invariant AS-prec-first-action[THEN spec,THEN spec,THEN spec,
where x2=s and x1=d and x=hd (execs d)]
    have AS-precondition s d (the (fst (control s d (execs d))))
    using control-spec[THEN spec,THEN spec,THEN spec,where x=execs d and x1=d and x2=s]
    by auto
   }
   hence fst (control s d (execs d)) ⇀ AS-precondition s d
    unfolding B-def
    by (cases fst (control s d (execs d)),auto)
  }
  from this invariant show ?thesis unfolding precondition-ind-def by auto
 qed
from thm-inductive 1 2 3 show ?thesis by auto
qed
```

Theorem unwinding_implies_isecure gives security for all realistic executions. For unrealistic 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–
{
 fix *execs d*
 assume *realistic-executions execs*
 hence *realistic-executions* (*purge execs d*)
  using *someI*[**where** *P*=*realistic-execution* **and** *x*=*execs d*]
  unfolding *realistic-executions-def purge-def* **by**(*simp*)
}
thus *?thesis* **by** *auto*
qed

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

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

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

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

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

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

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

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

theorem unwinding-implies-isecure-total:
shows isecure-total
proof–
 from unwinding-implies-isecure have secure-partial: NI-unrelated unfolding isecure-def by blast
 from unwinding-implies-isecure have isecure1-partial: NI-indirect-sources unfolding isecure-def by blast

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

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

have *let s-f = run-total n s0 execs in output-f (run-total n s0 (ipurge-l execs (current s-f))) a = output-f (run-total n s0 (ipurge-r execs (current s-f))) a*
proof (*cases run n* (*Some s0*) *execs*)
case *None*
thus *?thesis* **using** *1* **unfolding** *NI-unrelated-total-def strict-equal-def* **by** *auto*
next
case (*Some s-f*)
from *realistic-ipurge-l invariant-s0 realistic run-total-equals-run*[**where** *n=n* **and** *s=s0* **and** *execs=ipurge-l execs* (*current s-f*)]
have *2*: *strict-equal* (*run n* (*Some s0*) (*ipurge-l execs* (*current s-f*))) (*run-total n s0* (*ipurge-l execs* (*current s-f*)))
by *auto*
from *realistic-ipurge-r invariant-s0 realistic run-total-equals-run*[**where** *n=n* **and** *s=s0* **and** *execs=ipurge-r execs* (*current s-f*)]
have *3*: *strict-equal* (*run n* (*Some s0*) (*ipurge-r execs* (*current s-f*))) (*run-total n s0* (*ipurge-r execs* (*current s-f*)))
by *auto*

show *?thesis* **proof**(*cases run n* (*Some s0*) (*ipurge-l execs* (*current s-f*)))
case *None*
from *2 None* **show** *?thesis* **using** *2* **unfolding** *NI-unrelated-total-def strict-equal-def* **by** *auto*
next
case (*Some s-ipurge-l*)
show *?thesis*
proof(*cases run n* (*Some s0*) (*ipurge-r execs* (*current s-f*)))
case *None*
from *3 None* **show** *?thesis* **using** *2* **unfolding** *NI-unrelated-total-def strict-equal-def* **by** *auto*
next
case (*Some s-ipurge-r*)
from ‹*run n* (*Some s0*) *execs = Some s-f*› ‹*run n* (*Some s0*) (*ipurge-l execs* (*current s-f*)) = *Some s-ipurge-l*›
Some 1 2 3 isecure1-partial[*unfolded NI-indirect-sources-def* ,*THEN spec*,*THEN spec*,*THEN spec*,**where** *x=n* **and** *x2=execs*]
show *?thesis*
unfolding *strict-equal-def NI-unrelated-def*
by(*simp add*: *Let-def B-def B2-def*)
qed
qed
qed
}
thus *?thesis* **unfolding** *NI-indirect-sources-total-def* **by** *auto*
qed
from *NI-unrelated-total NI-indirect-sources-total* **show** *?thesis* **unfolding** *isecure-total-def* **by** *auto*
qed

end
end

3.4 CISK (Controlled Interruptible Separation Kernel)

theory *CISK*
imports *ISK*
begin

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

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

locale *Controllable-Interruptible-Separation-Kernel* = — CISK
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
by *pat-completeness auto*
termination by *lexicographic-order*

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

abbreviation *next-action*::*'state-t* ⇒ (*'dom-t* ⇒ *'action-t execution*) ⇒ *'action-t option*
where *next-action* ≡ *Kernel.next-action current CISK-control*
abbreviation *next-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*)
using *not0-implies-Suc* **by** (*metis prod-cases3*,*auto*)
termination by *lexicographic-order*

3.4.2 Formulations of security

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

abbreviation *kprecondition*
where *kprecondition s a ≡ invariant s ∧ AS-precondition s (current s) a*
definition *realistic-execution*
where *realistic-execution aseq ≡ set aseq ⊆ AS-set*
definition *realistic-executions* :: (*'dom-t ⇒ 'action-t execution*) ⇒ *bool*
where *realistic-executions execs ≡ ∀ d. realistic-execution (execs d)*
abbreviation *involved* **where** *involved ≡ Kernel.involved 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–
{
fix *s t execs*
assume *vpeq*: *vpeq (current s) s t*
assume *vpeq-involved*: *∀ d ∈ involved (next-action s execs) . vpeq d s t*
assume *current-s-t*: *current s = current t*
from *aborting-consistent current-s-t vpeq*
have *aborting t (current s) = aborting s (current s)* **by** *auto*
from *current-s-t this waiting-consistent vpeq-involved*
have *next-action s execs = next-action t execs*
unfolding *Kernel.next-action-def*
by(*cases (s,(current s),execs (current s)) rule*: *CISK-control.cases,auto*)
}
thus *?thesis* **by** *auto*
qed

lemma *next-execs-consistent*:
shows ∀ *s t execs* . *vpeq* (*current s*) *s t* ∧ (∀ *d* ∈ *involved* (*next-action s execs*) . *vpeq d s t*) ∧ *current s* = *current t* ⟶ *fst* (*snd* (*CISK-control s* (*current s*) (*execs* (*current s*)))) = *fst* (*snd* (*CISK-control t* (*current s*) (*execs* (*current s*))))
proof–
{
fix *s t execs*
assume *vpeq*: *vpeq* (*current s*) *s t*
assume *vpeq-involved*: ∀ *d* ∈ *involved* (*next-action s execs*) . *vpeq d s t*
assume *current-s-t*: *current s* = *current t*
from *aborting-consistent current-s-t vpeq*
have *1*: *aborting t* (*current s*) = *aborting s* (*current s*) **by** *auto*
from *1 vpeq current-s-t vpeq-involved waiting-consistent*[*THEN spec*,*THEN spec*,*THEN spec*,*THEN spec*,**where** *x3*=*s* **and** *x2*=*t* **and** *x1*=*current s* **and** *x*=*the* (*next-action s execs*)]
have *fst* (*snd* (*CISK-control s* (*current s*) (*execs* (*current s*)))) = *fst* (*snd* (*CISK-control t* (*current s*) (*execs* (*current s*))))
unfolding *Kernel.next-action-def Kernel.involved-def*
by(*cases* (*s*,(*current s*),*execs* (*current s*)) *rule*: *CISK-control.cases*,*auto split*: *if-split-asm*)
}
thus *?thesis* **by** *auto*
qed

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

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

lemma *locally-respects-next-state*:
shows ∀ *s u execs*. ¬*ifp* (*current s*) *u* ⟶ *vpeq u s* (*next-state s execs*)
proof–
{

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

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

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

lemma *next-action-after-next-state*:
shows ∀ *s execs d* . *current s* ≠ *d* ⟶ *fst* (*CISK-control* (*next-state s execs*) *d* (*execs d*)) = *None* ∨ *fst* (*CISK-control* (*next-state s execs*) *d* (*execs d*)) = *fst* (*CISK-control s d* (*execs d*))
proof–
{
fix *s execs d aseqs*
assume *1*: *current s* ≠ *d*
have *fst* (*CISK-control* (*next-state s execs*) *d aseqs*) = *None* ∨ *fst* (*CISK-control* (*next-state s execs*) *d aseqs*) = *fst* (*CISK-control s d aseqs*)
proof(*cases* (*s*,*d*,*aseqs*) *rule*: *CISK-control.cases*,*simp*,*simp*,*simp*)

```
    case (4 sa da a as execs')
      thus ?thesis
        unfolding Kernel.next-state-def
        using aborting-error-update waiting-error-update 1
        by(cases (sa,current sa,execs (current sa)) rule: CISK-control.cases,auto split: if-split-asm)
    qed
}
thus ?thesis by auto
qed

lemma next-action-after-step:
shows ∀ s a d aseqs . current s ≠ d ⟶ fst (CISK-control (step s a) d aseqs) = fst (CISK-control s d aseqs)
proof–
{
  fix s a d aseqs
  assume 1: current s ≠ d
  from this aborting-after-step
    have fst (CISK-control (step s a) d aseqs) = fst (CISK-control s d aseqs)
    unfolding Kernel.step-def
    by(cases (s,d,aseqs) rule: CISK-control.cases,simp,simp,simp,cases a,auto)
}
thus ?thesis by auto
qed

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

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

lemma next-action-from-execs:
shows ∀ s execs . next-action s execs ⇀ (λ a . a ∈ actions-in-execution (execs (current s)))
proof–
{
  fix s execs
```

```
  {
    fix a
    assume 1: next-action s execs = Some a
    from 1 have a ∈ actions-in-execution (execs (current s))
      unfolding Kernel.next-action-def actions-in-execution-def
      by (cases (s,(current s),execs (current s)) rule: CISK-control.cases,auto split: if-split-asm)
  }
  hence next-action s execs ⇀ (λ a . a ∈ actions-in-execution (execs (current s)))
    unfolding B-def
    by (cases next-action s execs,auto)
}
thus ?thesis unfolding B-def by (auto)
qed

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


theorem unwinding-implies-isecure-CISK:
shows isecure
proof–
  interpret int: Interruptible-Separation-Kernel kstep output-f s0 current cswitch interrupt kprecondition realistic-execution CISK-control kinvolved ifp vpeq AS-set invariant AS-precondition aborting waiting
    proof (unfold-locales)
      show ∀ a b c u. vpeq u a b ∧ vpeq u b c ⟶ vpeq u a c
        using vpeq-transitive by blast
      show ∀ a b u. vpeq u a b ⟶ vpeq u b a
        using vpeq-symmetric by blast
      show ∀ a u. vpeq u a a
        using vpeq-reflexive by blast
      show ∀ u. ifp u u
        using ifp-reflexive by blast
      show ∀ s t u a. vpeq u s t ∧ vpeq (current s) s t ∧ kprecondition s a ∧ kprecondition t a ∧ current s = current t ⟶ vpeq u (kstep s a) (kstep t a)
        using weakly-step-consistent by blast
      show ∀ a s u. ¬ifp (current s) u ∧ kprecondition s a ⟶ vpeq u s (kstep s a)
        using locally-respects by blast
      show ∀ a s t. vpeq (current s) s t ∧ current s = current t ⟶ (output-f s a) = (output-f t a)
        using output-consistent by blast
      show ∀ s a . current (kstep s a) = current s
        using step-atomicity by blast
      show ∀ n s t . current s = current t ⟶ current (cswitch n s) = current (cswitch n t)
        using cswitch-independent-of-state by blast
      show ∀ u s t n . vpeq u s t ⟶ vpeq u (cswitch n s) (cswitch n t)
        using cswitch-consistency by blast
      show ∀ s t execs. vpeq (current s) s t ∧ (∀ d ∈ involved (next-action s execs) . vpeq d s t) ∧ current s = current t ⟶ next-action s execs = next-action t execs
        using next-action-consistent by blast
```

show ∀ *s t execs*.
vpeq (*current s*) *s t* ∧ (∀ *d* ∈ *involved* (*next-action s execs*) . *vpeq d s t*) ∧ *current s* = *current t* ⟶
fst (*snd* (*CISK-control s* (*current s*) (*execs* (*current s*)))) = *fst* (*snd* (*CISK-control t* (*current s*) (*execs* (*current s*))))
using *next-execs-consistent* **by** *blast*
show ∀ *s t u execs*. *vpeq* (*current s*) *s t* ∧ *vpeq u s t* ∧ *current s* = *current t* ⟶ *vpeq u* (*next-state s execs*) (*next-state t execs*)
using *next-state-consistent* **by** *auto*
show ∀ *s execs*. *current* (*next-state s execs*) = *current s*
using *current-next-state* **by** *auto*
show ∀ *s u execs*. ¬ *ifp* (*current s*) *u* ⟶ *vpeq u s* (*next-state s execs*)
using *locally-respects-next-state* **by** *auto*
show [] ∈ *AS-set*
using *empty-in-AS-set* **by** *blast*
show ∀ *s n* . *invariant s* ⟶ *invariant* (*cswitch n s*)
using *invariant-after-cswitch* **by** *blast*
show ∀ *s d n a*. *AS-precondition s d a* ⟶ *AS-precondition* (*cswitch n s*) *d a*
using *precondition-after-cswitch* **by** *blast*
show *invariant s0*
using *invariant-s0* **by** *blast*
show ∀ *s d aseq* . *invariant s* ∧ *aseq* ∈ *AS-set* ∧ *aseq* ≠ [] ⟶ *AS-precondition s d* (*hd aseq*)
using *AS-prec-first-action* **by** *blast*
show ∀ *s a a′*. (∃ *aseq*∈*AS-set*. *is-sub-seq a a′ aseq*) ∧ *invariant s* ∧ *AS-precondition s* (*current s*) *a* ∧ ¬ *aborting s* (*current s*) *a* ∧ ¬ *waiting s* (*current s*) *a* ⟶
AS-precondition (*kstep s a*) (*current s*) *a′*
using *AS-prec-after-step* **by** *blast*
show ∀ *s d a a′* . *current s* ≠ *d* ∧ *AS-precondition s d a* ⟶ *AS-precondition* (*kstep s a′*) *d a*
using *AS-prec-dom-independent* **by** *blast*
show ∀ *s a* . *invariant s* ⟶ *invariant* (*kstep s a*)
using *spec-of-invariant* **by** *blast*
show ⋀*s a*. *kprecondition s a* ≡ *kprecondition s a*
by *auto*
show ⋀*aseq*. *realistic-execution aseq* ≡ *set aseq* ⊆ *AS-set*
unfolding *realistic-execution-def*
by *auto*
show ∀ *s a*. ∀ *d* ∈ *involved a*. *kprecondition s* (*the a*) ⟶ *ifp d* (*current s*)
using *involved-ifp* **unfolding** *Kernel.involved-def* **by** (*auto split*: *option.splits*)
show ∀ *s execs*. *next-action s execs* ⇀ (λ*a*. *a* ∈ *actions-in-execution* (*execs* (*current s*)))
using *next-action-from-execs* **by** *blast*
show ∀ *s execs u*. *actions-in-execution* (*next-execs s execs u*) ⊆ *actions-in-execution* (*execs u*)
using *next-execs-subset* **by** *blast*
show ∀ *s d aseqs*.
case CISK-control s d aseqs of
(*a*, *aseqs′*, *s′*) ⇒
thread-empty aseqs ∧ (*a*, *aseqs′*) = (*None*, []) ∨
aseqs ≠ [] ∧ *hd aseqs* ≠ [] ∧ ¬ *aborting s′ d* (*the a*) ∧ ¬ *waiting s′ d* (*the a*) ∧ (*a*, *aseqs′*) = (*Some* (*hd* (*hd aseqs*)), *tl* (*hd aseqs*) # *tl aseqs*) ∨
aseqs ≠ [] ∧ *hd aseqs* ≠ [] ∧ *waiting s′ d* (*the a*) ∧ (*a*, *aseqs′*, *s′*) = (*Some* (*hd* (*hd aseqs*)), *aseqs*, *s*) ∨ (*a*, *aseqs′*) = (*None*, *tl aseqs*)
using *CISK-control-spec* **by** *blast*
show ∀ *s n d aseqs*. *fst* (*CISK-control* (*cswitch n s*) *d aseqs*) = *fst* (*CISK-control s d aseqs*)
using *next-action-after-cswitch* **by** *auto*
show ∀ *s execs d*.
current s ≠ *d* ⟶
fst (*CISK-control* (*next-state s execs*) *d* (*execs d*)) = *None* ∨ *fst* (*CISK-control* (*next-state s execs*) *d* (*execs d*)) = *fst* (*CISK-control s d* (*execs d*))
using *next-action-after-next-state* **by** *auto*

show ∀ *s a d aseqs*. *current s* ≠ *d* ⟶ *fst* (*CISK-control* (*step s a*) *d aseqs*) = *fst* (*CISK-control s d aseqs*)
using *next-action-after-step* **by** *auto*
show ∀ *s d a execs*. *AS-precondition s d a* ⟶ *AS-precondition* (*next-state s execs*) *d a*
using *next-state-precondition* **by** *auto*
show ∀ *s execs*. *invariant s* ⟶ *invariant* (*next-state s execs*)
using *next-state-invariant* **by** *auto*
show ∀ *s a*. *waiting s* (*current s*) *a* ⟶ *kstep s a* = *s*
using *spec-of-waiting* **by** *blast*
qed

note *interpreted* = *int.Interruptible-Separation-Kernel-axioms*
note *run-total-induct* = *Interruptible-Separation-Kernel.run-total.induct*[*of kstep output-f s0 current cswitch kprecondition realistic-execution*
CISK-control kinvolved ifp vpeq AS-set invariant AS-precondition aborting waiting - interrupt]
have *run-equals-run-total*:
⋀ *n s execs* . *run n s execs* ≡ *Interruptible-Separation-Kernel.run-total kstep current cswitch interrupt CISK-control n s execs*
proof−
fix *n s execs*
show *run n s execs* ≡ *Interruptible-Separation-Kernel.run-total kstep current cswitch interrupt CISK-control n s execs*
using *interpreted int.step-def*
by(*induct n s execs rule*: *run-total-induct*,*auto split*: *option.splits*)
qed
from *interpreted*
have *0*: *Interruptible-Separation-Kernel.isecure-total kstep output-f s0 current cswitch interrupt realistic-execution CISK-control kinvolved ifp*
by (*metis int.unwinding-implies-isecure-total*)
from *0 run-equals-run-total*
have *1*: *NI-unrelated*
by (*metis realistic-executions-def int.isecure-total-def int.realistic-executions-def int.NI-unrelated-total-def NI-unrelated-def*)
from *0 run-equals-run-total*
have *2*: *NI-indirect-sources*
by (*metis realistic-executions-def int.NI-indirect-sources-total-def int.isecure-total-def int.realistic-executions-def NI-indirect-sources-def*)
from *1 2* **show** *?thesis* **unfolding** *isecure-def* **by** *auto*
qed

end
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 (unfold-locales)
  show sp-spec-file-provider:
    ∀ p1 p2 f m1 m2 .
       sp-spec-subj-obj p1 (FILEP f) m1 ∧
       sp-spec-subj-obj p2 (FILEP f) m2 ⟶ sp-spec-subj-subj p1 p2
    unfolding sp-spec-subj-subj-def by auto
  show sp-spec-no-wronly-pages:
    ∀ p x . sp-spec-subj-obj p (PAGE x) WRITE ⟶ sp-spec-subj-obj p (PAGE x) READ
    unfolding sp-spec-subj-obj-def by auto
  show ifp-reflexive:
    ∀ p . ifp p p
    unfolding ifp-def by auto
  show ifp-compatible-with-sp-spec:
    ∀ a b . sp-spec-subj-subj a b ⟶ ifp a b ∧ ifp b a
    unfolding ifp-def by auto
  show ifp-compatible-with-ipc:
    ∀ a b c x . (sp-spec-subj-subj a b
          ∧ sp-spec-subj-obj b (PAGE x) WRITE ∧ sp-spec-subj-obj c (PAGE x) READ)
         ⟶ ifp a c
    unfolding ifp-def by auto
  qed
end

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

interpretation Policy: abstract-policy-derivation configured-subj-obj.
interpretation Policy-properties: policy-axioms Policy.sp-spec-subj-obj Policy.sp-spec-subj-subj Policy.ifp
  using Policy.correct by auto

lemma example-how-to-use-properties-in-proofs:
  shows ∀ p . Policy.ifp p p
  using Policy-properties.ifp-reflexive by auto

end
```

4.3 Separation kernel state and atomic step function

```
theory Step
  imports Step-policies
begin
```

4.3.1 Interrupt points

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

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

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

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

4.3.2 System state

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 –
show *?thesis*
unfolding *atomic-step-invariant-def atomic-step-precondition-def ipc-precondition-def sp-subset-def set-object-value-def Let-def*
by (*simp split*: *int-point-t.splits ipc-stage-t.splits ipc-direction-t.splits*)
qed

lemma *set-thread-value-invariant*:
shows *atomic-step-invariant s* = *atomic-step-invariant* (*s* (| *thread* := *thrst* |))
proof –
show *?thesis*
unfolding *atomic-step-invariant-def atomic-step-precondition-def ipc-precondition-def sp-subset-def set-object-value-def Let-def*
by (*simp split*: *int-point-t.splits ipc-stage-t.splits ipc-direction-t.splits*)
qed

lemma *atomic-ipc-preserves-invariants*:
fixes *s* :: *state-t*
and *tid* :: *thread-id-t*
assumes *atomic-step-invariant s*
shows *atomic-step-invariant* (*atomic-step-ipc tid dir stage partner page s*)
proof –
show *?thesis*
proof (*cases stage*)
case *PREP*
from *this assms* **show** *?thesis*

```
      unfolding atomic-step-ipc-def atomic-step-invariant-def by auto
    next
    case WAIT
     from this assms show ?thesis
      unfolding atomic-step-ipc-def atomic-step-invariant-def by auto
    next
    case BUF
     show ?thesis
      using assms BUF set-object-value-invariant
      unfolding atomic-step-ipc-def
      by (simp split: ipc-direction-t.splits)
    qed
qed

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

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

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

4.4.2 Summary theorems on atomic step invariants

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

```
theorem atomic-step-preserves-invariants:
 fixes s :: state-t
  and tid :: thread-id-t
 assumes atomic-step-invariant s
 shows atomic-step-invariant (atomic-step s a)
proof (cases a)
```

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

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

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

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

end

4.5 The view-partitioning equivalence relation

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

The view consists of

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

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

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

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

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

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

4.5.1 Elementary properties

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

Auxiliary equivalence relation.

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

qed

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

theorem *cswitch-consistency-and-respect*:
 fixes *u* :: *partition-id-t*
  and *s* :: *state-t*
  and *new-current* :: *thread-id-t*
 assumes *atomic-step-invariant s*
 shows *vpeq u s* (*s* (| *current* := *new-current* |))
proof –
 show *?thesis*
  unfolding *vpeq-def vpeq-obj-def vpeq-subj-subj-def vpeq-subj-obj-def vpeq-local-def*
  by *auto*
qed

end

4.6 Atomic step locally respects the information flow policy

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

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

4.6.1 Locally respects of atomic step functions

lemma *ipc-respects-policy*:
 assumes *no*: ¬ *Policy.ifp* (*partition tid*) *u*
  and *inv*: *atomic-step-invariant s*
  and *prec*: *atomic-step-precondition s tid* (*SK-IPC dir stage partner pag*)
  and *ipt-case*: *ipt* = *SK-IPC dir stage partner page*
 shows *vpeq u s* (*atomic-step-ipc tid dir stage partner page s*)
 proof(*cases stage*)
  case *PREP*
  thus *?thesis*
  unfolding *atomic-step-ipc-def*
  using *vpeq-refl* **by** *simp*
  next
  case *WAIT*
  thus *?thesis*
  unfolding *atomic-step-ipc-def*
  using *vpeq-refl* **by** *simp*
  next case (*BUF mypage*)
  show *?thesis*
  proof(*cases dir*)
  case *RECV*
   thus *?thesis*
   unfolding *atomic-step-ipc-def*
   using *vpeq-refl BUF* **by** *simp*
  next
  case *SEND*
   have *Policy.sp-spec-subj-subj* (*partition tid*) (*partition partner*)
    and *Policy.sp-spec-subj-obj* (*partition partner*) (*PAGE mypage*) *WRITE*

```
        using BUF SEND inv prec ipt-case
        unfolding atomic-step-invariant-def sp-subset-def
        unfolding atomic-step-precondition-def ipc-precondition-def opposite-ipc-direction-def
        by auto
      hence ¬ Policy.sp-spec-subj-obj u (PAGE mypage) READ
        using no Policy-properties.ifp-compatible-with-ipc
        by auto
      thus ?thesis
        using BUF SEND assms
        unfolding atomic-step-ipc-def set-object-value-def
        unfolding vpeq-def vpeq-obj-def vpeq-subj-obj-def vpeq-subj-subj-def vpeq-local-def
        by auto
    qed
  qed

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

lemma ev-wait-all-respects-policy:
  assumes no: ¬ Policy.ifp (partition tid) u
    and inv: atomic-step-invariant s
    and prec: atomic-step-precondition s tid ipt
    and ipt-case: ipt = SK-EV-WAIT ev-wait-stage EV-CONSUME-ALL
  shows vpeq u s (atomic-step-ev-wait-all tid s)
  proof –
  from assms have 1:(partition tid) ≠ u
  unfolding Policy.ifp-def
   by simp
  then have 2:vpeq-local u s (atomic-step-ev-wait-all tid s)
  unfolding vpeq-local-def atomic-step-ev-wait-all-def
```

```
   by simp
  have 3:vpeq-obj u s (atomic-step-ev-wait-all tid s)
  unfolding vpeq-obj-def atomic-step-ev-wait-all-def
   by simp
  have 4:vpeq-subj-subj u s (atomic-step-ev-wait-all tid s)
  unfolding vpeq-subj-subj-def atomic-step-ev-wait-all-def
   by simp
  have 5:vpeq-subj-obj u s (atomic-step-ev-wait-all tid s)
  unfolding vpeq-subj-obj-def atomic-step-ev-wait-all-def
   by simp
  with 2 3 4 5 show ?thesis
  unfolding vpeq-def
   by simp
qed

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

4.6.2 Summary theorems on view-partitioning locally respects

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

```
theorem atomic-step-respects-policy:
  assumes 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 –
  show ?thesis
    using assms ipc-respects-policy vpeq-refl
          ev-signal-respects-policy ev-wait-one-respects-policy
          ev-wait-all-respects-policy
    unfolding atomic-step-def
```

by (*auto split*: *int-point-t.splits ev-consume-t.splits ev-wait-stage-t.splits ev-signal-stage-t.splits*)
qed

end

4.7 Weak step consistency

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

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

4.7.1 Weak step consistency of auxiliary functions

lemma *ipc-precondition-weakly-step-consistent*:
assumes *eq-tid*: *vpeq* (*partition tid*) *s1 s2*
and *inv1*: *atomic-step-invariant s1*
and *inv2*: *atomic-step-invariant s2*
shows *ipc-precondition tid dir partner page s1* = *ipc-precondition tid dir partner page s2*
proof –
let *?sender* = *case dir of SEND* ⇒ *tid* | *RECV* ⇒ *partner*
let *?receiver* = *case dir of SEND* ⇒ *partner* | *RECV* ⇒ *tid*
let *?local-access-mode* = *case dir of SEND* ⇒ *READ* | *RECV* ⇒ *WRITE*
let *?A* = *sp-impl-subj-subj s1* (*partition ?sender*) (*partition ?receiver*)
= *sp-impl-subj-subj s2* (*partition ?sender*) (*partition ?receiver*)
let *?B* = *sp-impl-subj-obj s1* (*partition tid*) (*PAGE page*) *?local-access-mode*
= *sp-impl-subj-obj s2* (*partition tid*) (*PAGE page*) *?local-access-mode*

have *A*: *?A*
proof (*cases Policy.sp-spec-subj-subj* (*partition ?sender*) (*partition ?receiver*))
case *True*
thus *?A*
using *eq-tid* **unfolding** *vpeq-def vpeq-subj-subj-def*
by (*simp split*: *ipc-direction-t.splits*)
next case *False*
have *sp-subset s1* **and** *sp-subset s2*
using *inv1 inv2* **unfolding** *atomic-step-invariant-def sp-subset-def* **by** *auto*
hence ¬ *sp-impl-subj-subj s1* (*partition ?sender*) (*partition ?receiver*)
and ¬ *sp-impl-subj-subj s2* (*partition ?sender*) (*partition ?receiver*)
using *False* **unfolding** *sp-subset-def* **by** *auto*
thus *?A* **by** *auto*
qed
have *B*: *?B*
proof (*cases Policy.sp-spec-subj-obj* (*partition tid*) (*PAGE page*) *?local-access-mode*)
case *True*
thus *?B*
using *eq-tid* **unfolding** *vpeq-def vpeq-subj-obj-def*
by (*simp split*: *ipc-direction-t.splits*)
next case *False*
have *sp-subset s1* **and** *sp-subset s2*
using *inv1 inv2* **unfolding** *atomic-step-invariant-def sp-subset-def* **by** *auto*
hence ¬ *sp-impl-subj-obj s1* (*partition tid*) (*PAGE page*) *?local-access-mode*
and ¬ *sp-impl-subj-obj s2* (*partition tid*) (*PAGE page*) *?local-access-mode*
using *False* **unfolding** *sp-subset-def* **by** *auto*
thus *?B* **by** *auto*

```
    qed
  show ?thesis using A B unfolding ipc-precondition-def by auto
qed

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


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

unfolding *set-object-value-def* **by** *auto*
thus *vpeq-subj-subj u ?s1′ ?s2′*
using *eq-obs* **unfolding** *vpeq-def vpeq-subj-subj-def* **by** *auto*
qed
have *E5*: *vpeq-subj-obj u ?s1′ ?s2′*
proof –
have *sp-impl-subj-obj ?s1′ = sp-impl-subj-obj s1*
and *sp-impl-subj-obj ?s2′ = sp-impl-subj-obj s2*
unfolding *set-object-value-def* **by** *auto*
thus *vpeq-subj-obj u ?s1′ ?s2′*
using *eq-obs* **unfolding** *vpeq-def vpeq-subj-obj-def* **by** *auto*
qed
from *eq-obs* **have** *E6*: *vpeq-local u ?s1′ ?s2′*
unfolding *vpeq-def vpeq-local-def set-object-value-def*
by *simp*
from *E1 E4 E5 E6*
show *?thesis* **unfolding** *vpeq-def*
by *auto*
qed

4.7.2 Weak step consistency of atomic step functions

lemma *ipc-weakly-step-consistent*:
assumes *eq-obs*: *vpeq u s1 s2*
and *eq-act*: *vpeq (partition tid) s1 s2*
and *inv1*: *atomic-step-invariant s1*
and *inv2*: *atomic-step-invariant s2*
and *prec1*: *atomic-step-precondition s1 tid ipt*
and *prec2*: *atomic-step-precondition s1 tid ipt*
and *ipt-case*: *ipt = SK-IPC dir stage partner page*
shows *vpeq u*
(atomic-step-ipc tid dir stage partner page s1)
(atomic-step-ipc tid dir stage partner page s2)
proof –
have ⋀ *mypage* . ⟦ *dir = SEND*; *stage = BUF mypage* ⟧ ⟹ *?thesis*
proof –
fix *mypage*
assume *dir-send*: *dir = SEND*
assume *stage-buf*: *stage = BUF mypage*
have *Policy.sp-spec-subj-obj (partition tid) (PAGE page) READ*
using *inv1 prec1 dir-send stage-buf ipt-case*
unfolding *atomic-step-invariant-def sp-subset-def*
unfolding *atomic-step-precondition-def ipc-precondition-def opposite-ipc-direction-def*
by *auto*
hence *obj s1 (PAGE page) = obj s2 (PAGE page)*
using *eq-act* **unfolding** *vpeq-def vpeq-obj-def vpeq-local-def*
by *auto*
thus *vpeq u*
(atomic-step-ipc tid dir stage partner page s1)
(atomic-step-ipc tid dir stage partner page s2)
using *dir-send stage-buf eq-obs set-object-value-consistent*
unfolding *atomic-step-ipc-def*
by *auto*
qed
thus *?thesis*
using *eq-obs* **unfolding** *atomic-step-ipc-def*
by *(cases stage, auto, cases dir, auto)*
qed

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

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

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

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

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

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

lemma *extend-subj-subj-consistent*:
fixes *f* :: *partition-id-t* ⇒ *partition-id-t* ⇒ *bool*
assumes *vpeq u s1 s2*

```
 shows vpeq u (extend-subj-subj f s1) (extend-subj-subj f s2)
proof -
 let ?g1 = sp-impl-subj-subj s1 and ?g2 = sp-impl-subj-subj s2
 have ∀ v . Policy.sp-spec-subj-subj u v ⟶ ?g1 u v = ?g2 u v
  and ∀ v . Policy.sp-spec-subj-subj v u ⟶ ?g1 v u = ?g2 v u
  using assms unfolding vpeq-def vpeq-subj-subj-def by auto
 hence ∀ v . Policy.sp-spec-subj-subj u v ⟶ extend-f f ?g1 u v = extend-f f ?g2 u v
  and ∀ v . Policy.sp-spec-subj-subj v u ⟶ extend-f f ?g1 v u = extend-f f ?g2 v u
  unfolding extend-f-def by auto
 hence 1: vpeq-subj-subj u (extend-subj-subj f s1) (extend-subj-subj f s2)
  unfolding vpeq-subj-subj-def extend-subj-subj-def
  by auto
 have 2: vpeq-obj u (extend-subj-subj f s1) (extend-subj-subj f s2)
  using assms unfolding vpeq-def vpeq-obj-def extend-subj-subj-def by fastforce
 have 3: vpeq-subj-obj u (extend-subj-subj f s1) (extend-subj-subj f s2)
  using assms unfolding vpeq-def vpeq-subj-obj-def extend-subj-subj-def by fastforce
 have 4: vpeq-local u (extend-subj-subj f s1) (extend-subj-subj f s2)
  using assms unfolding vpeq-def vpeq-local-def extend-subj-subj-def by fastforce
 from 1 2 3 4 show ?thesis
  using assms unfolding vpeq-def by fast
qed
```

4.7.3 Summary theorems on view-partitioning weak step consistency

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

```
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 -
 show ?thesis
  using assms
     ipc-weakly-step-consistent
     ev-wait-all-weakly-step-consistent
     ev-wait-one-weakly-step-consistent
     ev-signal-weakly-step-consistent
     vpeq-refl
  unfolding atomic-step-def
  apply (cases ipt, auto)
  apply (simp split: ev-consume-t.splits ev-wait-stage-t.splits)
  by (simp split: ev-signal-stage-t.splits)
 qed
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 –
 have *sp-subset s0*
  unfolding *sp-subset-def s0-def* **by** *auto*
 thus *?thesis*
  unfolding *atomic-step-invariant-def* **by** *auto*
qed

4.8.2 Types for instantiation of the generic model

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

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

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

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

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

lemma *rstate-up-down*[*simp*]:
assumes *atomic-step-invariant s*
shows (↓↑*s*) = *s*
using *assms Abs-rstate-t-inverse* **unfolding** *rep-def abs-def* **by** *auto*

A CISK action is identified with an interrupt point.

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

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

definition *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*
unfolding *rvpeq-def* **using** *vpeq-rel* **by** *metis+*

lemma *inst-ifp-refl*:
shows ∀ *u* . *rifp u u*
unfolding *rifp-def* **using** *Policy-properties.ifp-reflexive* **by** *fast*

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

```
using atomic-step-does-not-change-current-thread rstate-up-down rstate-invariant atomic-step-preserves-invariants
  by auto


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


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


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

by *auto*
thus *routput-f s a p = routput-f t a p*
using *2* **unfolding** *routput-f-def* **by** *simp*
qed }
thus $\forall$ *a s t. rvpeq (rcurrent s) s t* $\wedge$ *rcurrent s = rcurrent t* $\longrightarrow$ *routput-f s a = routput-f t a*
by *auto*
qed
thus *?thesis* **using** *assms* **by** *auto*
qed

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

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

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

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

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

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

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

lemma *prec-first-EV-SIGNAL-action*:
assumes *is-SK-EV-SIGNAL aseq*
shows *rprecondition s d (hd aseq)*
using *assms*
unfolding *is-SK-EV-SIGNAL-def rprecondition-def atomic-step-precondition-def ev-signal-precondition-def*
by *auto*

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

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

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

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

```
  fix consume

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

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

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

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

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

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

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

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

lemma *prec-ev-wait-all-dom-independent*:
assumes *current s* ≠ *d*
and *atomic-step-invariant s*
and *atomic-step-precondition s d a*
shows *atomic-step-precondition* (*atomic-step-ev-wait-all* (*current s*) *s*) *d a*
using *assms on-set-object-value*
unfolding *atomic-step-precondition-def atomic-step-ev-wait-all-def ipc-precondition-def*

```
     ev-signal-precondition-def set-object-value-def
     by (auto split: int-point-t.splits ipc-stage-t.splits ipc-direction-t.splits
         ev-consume-t.splits ev-wait-stage-t.splits ev-signal-stage-t.splits)

lemma prec-dom-independent:
shows ∀ s d a a′ . rcurrent s ≠ d ∧ rprecondition s d a ⟶ rprecondition (rstep s a′) d a
using atomic-step-preserves-invariants
rstate-invariant prec-IPC-dom-independent prec-ev-signal-dom-independent
prec-ev-wait-all-dom-independent prec-ev-wait-one-dom-independent
unfolding rcurrent-def rprecondition-def rstep-def atomic-step-def
by(auto split: int-point-t.splits  ev-consume-t.splits ev-wait-stage-t.splits ev-signal-stage-t.splits)

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

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

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

```
 hence rwaiting (rcswitch n s) = rwaiting s by auto
}
thus ?thesis by auto
qed

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

lemma waiting-after-IPC-step:
assumes d1 ≠ d2
shows waiting (atomic-step-ipc d1 dir stage partner page s) d2 a = waiting s d2 a
unfolding atomic-step-ipc-def waiting-def set-object-value-def ipc-precondition-def
by(auto split: int-point-t.splits ipc-stage-t.splits ipc-direction-t.splits
             ev-wait-stage-t.splits)


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

lemma aborting-dom-independent:
 assumes rcurrent s ≠ d
  shows raborting (rstep s a) d a′ = raborting s d a′
proof −
 have ⋀ tid dir partner page s . ipc-precondition tid dir partner page s = ipc-precondition tid dir partner page
(atomic-step s a)
                  ∧ ev-signal-precondition tid partner s = ev-signal-precondition tid partner (atomic-step s a)

  proof −
  fix tid dir partner page s
  let ?s = atomic-step s a
```

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

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

lemma *ipc-precondition-of-partner-consistent*:
assumes *vpeq*: ∀ *d* ∈ *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–
from *assms ipc-precondition-weakly-step-consistent rstate-invariant*
show *?thesis*
unfolding *rvpeq-def rkinvolved-def*
by *auto*
qed

lemma *ev-signal-precondition-of-partner-consistent*:
assumes *vpeq*: ∀ *d* ∈ *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–
from *assms ev-signal-precondition-weakly-step-consistent rstate-invariant*
show *?thesis*
unfolding *rvpeq-def rkinvolved-def*
by *auto*
qed

lemma *waiting-consistent*:
shows ∀ *s t u a* . *rvpeq* (*rcurrent s*) *s t* ∧ (∀ *d* ∈ *rkinvolved a* . *rvpeq d s t*)
∧ *rvpeq u s t*
⟶ *rwaiting s u a* = *rwaiting t u a*
proof–
{
fix *s t u a*
assume *vpeq*: *rvpeq* (*rcurrent s*) *s t*
assume *vpeq-involved*: ∀ *d* ∈ *rkinvolved a* . *rvpeq d s t*
assume *vpeq-u*: *rvpeq u s t*
have *rwaiting s u a* = *rwaiting t u a* **proof** (*cases a*)
case *SK-IPC*
thus *rwaiting s u a* = *rwaiting t u a*
using *ipc-precondition-of-partner-consistent vpeq-involved*
unfolding *waiting-def* **by** (*auto split*: *ipc-stage-t.splits*)
next case *SK-EV-WAIT*
thus *rwaiting s u a* = *rwaiting t u a*

```
      using ev-signal-precondition-of-partner-consistent
      vpeq-involved vpeq vpeq-u
      unfolding waiting-def rkinvolved-def ev-signal-precondition-def
            rvpeq-def vpeq-def vpeq-local-def
      by (auto split: ipc-stage-t.splits ev-wait-stage-t.splits ev-consume-t.splits)
    qed (simp add: waiting-def , simp add: waiting-def )
}
thus ?thesis by auto
qed

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

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

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

lemma spec-of-waiting-ev:
shows ∀ s a. rwaiting s (rcurrent s) (SK-EV-WAIT EV-FINISH EV-CONSUME-ALL)
         ⟶ rstep s a = s
 unfolding waiting-def
 by auto
```

lemma *spec-of-waiting-ev-w*:
shows ∀ *s a*. *rwaiting s* (*rcurrent s*) (*SK-EV-WAIT EV-WAIT EV-CONSUME-ALL*)
⟶ *rstep s* (*SK-EV-WAIT EV-WAIT EV-CONSUME-ALL*) = *s*
unfolding *rstep-def atomic-step-def*
by (*auto split*: *int-point-t.splits ipc-stage-t.splits ev-wait-stage-t.splits*)

lemma *spec-of-waiting*:
shows ∀ *s a*. *rwaiting s* (*rcurrent s*) *a* ⟶ *rstep s a* = *s*
unfolding *waiting-def rstep-def atomic-step-def atomic-step-ipc-def*
atomic-step-ev-signal-def atomic-step-ev-wait-all-def
atomic-step-ev-wait-one-def
by(*auto split*: *int-point-t.splits ipc-stage-t.splits ev-wait-stage-t.splits*)
end

4.9 Link implementation to CISK: the specific separation kernel is an interpretation of the generic model.

theory *Link-separation-kernel-model-to-CISK*
imports *Separation-kernel-model*
begin

We show that the separation kernel instantiation satisfies the specification of CISK.

theorem *CISK-proof-obligations-satisfied*:
shows
Controllable-Interruptible-Separation-Kernel
rstep
routput-f
(↑*s0*)
rcurrent
rcswitch
rkinvolved
rifp
rvpeq
rAS-set
rinvariant
rprecondition
raborting
rwaiting
rset-error-code
proof (*unfold-locales*)
— show that rvpeq is equivalence relation
show ∀ *a b c u*. (*rvpeq u a b* ∧ *rvpeq u b c*) ⟶ *rvpeq u a c*
and ∀ *a b u*. *rvpeq u a b* ⟶ *rvpeq u b a*
and ∀ *a u*. *rvpeq u a a*
using *inst-vpeq-rel* **by** *metis+*
— show output consistency
show ∀ *a s t*. *rvpeq* (*rcurrent s*) *s t* ∧ *rcurrent s* = *rcurrent t* ⟶ *routput-f s a* = *routput-f t a*
using *inst-output-consistency* **by** *metis*
— show reflexivity of ifp
show ∀ *u* . *rifp u u*
using *inst-ifp-refl* **by** *metis*
— show step consistency
show ∀ *s t u a*. *rvpeq u s t* ∧ *rvpeq* (*rcurrent s*) *s t* ∧ *True* ∧ *rprecondition s* (*rcurrent s*) *a* ∧ *True* ∧ *rprecondition t* (*rcurrent t*) *a* ∧ *rcurrent s* = *rcurrent t* ⟶
rvpeq u (*rstep s a*) (*rstep t a*)
using *inst-weakly-step-consistent* **by** *blast*
— show step atomicity

show $\forall$ *s a* . *rcurrent* (*rstep s a*) = *rcurrent s*
  using *inst-step-atomicity* **by** *metis*
show $\forall$ *a s u*. $\neg$ *rifp* (*rcurrent s*) *u* $\wedge$ *True* $\wedge$ *rprecondition s* (*rcurrent s*) *a* $\longrightarrow$ *rvpeq u s* (*rstep s a*)
  using *inst-local-respect* **by** *blast*
— show cswitch is independent of state
show $\forall$ *n s t*. *rcurrent s* = *rcurrent t* $\longrightarrow$ *rcurrent* (*rcswitch n s*) = *rcurrent* (*rcswitch n t*)
  using *inst-cswitch-independent-of-state* **by** *metis*
— show cswitch consistency
show $\forall$ *u s t n*. *rvpeq u s t* $\longrightarrow$ *rvpeq u* (*rcswitch n s*) (*rcswitch n t*)
  using *inst-cswitch-consistency* **by** *metis*
— Show the empt action sequence is in *AS-set*
show [] $\in$ *rAS-set*
  unfolding *rAS-set-def*
  by *auto*
— The invariant for the initial state, already encoded in *rstate-t*
show *True*
  by *auto*
— Step function of the invariant, already encoded in *rstate-t*
show $\forall$ *s n*. *True* $\longrightarrow$ *True*
  by *auto*
— The precondition does not change with a context switch
show $\forall$ *s d n a*. *rprecondition s d a* $\longrightarrow$ *rprecondition* (*rcswitch n s*) *d a*
  using *precondition-after-cswitch* **by** *blast*
— The precondition holds for the first action of each action sequence
show $\forall$ *s d aseq*. *True* $\wedge$ *aseq* $\in$ *rAS-set* $\wedge$ *aseq* $\neq$ [] $\longrightarrow$ *rprecondition s d* (*hd aseq*)
  using *prec-first-IPC-action prec-first-EV-WAIT-action prec-first-EV-SIGNAL-action*
  unfolding *rAS-set-def is-sub-seq-def*
  by *auto*
— The precondition holds for the next action in an action sequence, assuming the sequence is not aborted or delayed
show $\forall$ *s a a'*. ($\exists$*aseq*$\in$*rAS-set*. *is-sub-seq a a' aseq*) $\wedge$ *True* $\wedge$ *rprecondition s* (*rcurrent s*) *a* $\wedge$ $\neg$ *raborting s* (*rcurrent s*) *a* $\wedge$ $\neg$ *rwaiting s* (*rcurrent s*) *a* $\longrightarrow$
    rprecondition (*rstep s a*) (*rcurrent s*) *a'*
  using *prec-after-IPC-step prec-after-EV-SIGNAL-step prec-after-EV-WAIT-step*
  unfolding *rAS-set-def is-sub-seq-def*
  by *auto*
— Steps of other domains do not influence the precondition
show $\forall$ *s d a a'*. *rcurrent s* $\neq$ *d* $\wedge$ *rprecondition s d a* $\longrightarrow$ *rprecondition* (*rstep s a'*) *d a*
  using *prec-dom-independent* **by** *blast*
— The invariant
show $\forall$ *s a*. *True* $\longrightarrow$ *True*
  by *auto*
— Aborting does not depend on a context switch
show $\forall$ *n s*. *raborting* (*rcswitch n s*) = *raborting s*
  using *aborting-switch-independent* **by** *auto*
— Aborting does not depend on actions of other domains
show $\forall$ *s a d*. *rcurrent s* $\neq$ *d* $\longrightarrow$ *raborting* (*rstep s a*) *d* = *raborting s d*
  using *aborting-dom-independent* **by** *auto*
— Aborting is consistent
show $\forall$ *s t u*. *rvpeq u s t* $\longrightarrow$ *raborting s u* = *raborting t u*
  using *raborting-consistent* **by** *auto*
— Waiting does not depend on a context switch
show $\forall$ *n s*. *rwaiting* (*rcswitch n s*) = *rwaiting s*
  using *waiting-switch-independent* **by** *auto*
— Waiting is consistent
show $\forall$ *s t u a*. *rvpeq* (*rcurrent s*) *s t* $\wedge$ ($\forall$ *d* $\in$ *rkinvolved a* . *rvpeq d s t*)
    $\wedge$ *rvpeq u s t*

    ⟶ *rwaiting s u a* = *rwaiting t u a*
  unfolding *Kernel.involved-def*
  using *waiting-consistent* **by** *auto*
 — Domains that are involved in an action may influence the domain of the action
 show ∀ *s a*. ∀ *d* ∈ *rkinvolved a*. *rprecondition s* (*rcurrent s*) *a* ⟶ *rifp d* (*rcurrent s*)
  using *involved-ifp* **by** *blast*
 — An action that is waiting does not change the state
 show ∀ *s a*. *rwaiting s* (*rcurrent s*) *a* ⟶ *rstep s a* = *s*
  using *spec-of-waiting* **by** *blast*
 — Proof obligations for *set-error-code*. Right now, they are all trivial
 show ∀ *s d a′ a*. *rcurrent s* ≠ *d* ∧ *raborting s d a* ⟶ *raborting* (*rset-error-code s a′*) *d a*
  unfolding *rset-error-code-def*
  by *auto*
 show ∀ *s t u a*. *rvpeq u s t* ⟶ *rvpeq u* (*rset-error-code s a*) (*rset-error-code t a*)
  unfolding *rset-error-code-def*
  by *auto*
 show ∀ *s u a*. ¬ *rifp* (*rcurrent s*) *u* ⟶ *rvpeq u s* (*rset-error-code s a*)
  unfolding *rset-error-code-def*
  by (*metis* ‹∀ *a u*. *rvpeq u a a*›)
 show ∀ *s a*. *rcurrent* (*rset-error-code s a*) = *rcurrent s*
  unfolding *rset-error-code-def*
  by *auto*
 show ∀ *s d a a′*. *rprecondition s d a* ∧ *raborting s* (*rcurrent s*) *a′* ⟶ *rprecondition* (*rset-error-code s a′*) *d a*
  unfolding *rset-error-code-def*
  by *auto*
 show ∀ *s d a′ a*. *rcurrent s* ≠ *d* ∧ *rwaiting s d a* ⟶ *rwaiting* (*rset-error-code s a′*) *d a*
  unfolding *rset-error-code-def*
  by *auto*
qed

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

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

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

theorem *risecure*:
 Inst.isecure
using *Inst.unwinding-implies-isecure-CISK*
by *blast*

end

5 Related Work

We consider various definitions of intransitive (I) 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.