Part Number 84-0902045A000 Rev A

Version Date 27 June 1997

# DTOS COMPOSABILITY STUDY

## CONTRACT NO. MDA904-93-C-4209
## CDRL SEQUENCE NO. A020

**Prepared for:**
**Maryland Procurement Office**

**Prepared by:**

**Secure Computing Corporation**
**2675 Long Lake Road**
**Roseville, Minnesota 55113**

Authenticated by ______________________ Approved by ______________________
(Contracting Agency) (Contractor)

Date ______________________ Date ______________________

Technical Note

# DTOS COMPOSABILITY STUDY

Secure Computing Corporation

*Abstract*

This report describes a study into techniques for specifying and verifying modular systems using composability.

| | |
|---|---|
| **Part Number** | 84-0902045A000 Rev A |
| **Created** | 19 September 1995 |
| **Revised** | 27 June 1997 |
| **Done for** | Maryland Procurement Office |
| **Distribution** | Secure Computing and U.S. Government |
| **CM** | `/home/cmt/rev/dtos/docs/compose/RCS/compose.vdd,v 1.9 27 June 1997` |

This document was produced using the TEX document formatting system and the LATEX style macros.

# Contents

*Section* 1
# Scope

## 1.1 Identification

This report describes the results of a study into composability techniques performed on the Distributed Trusted Operating System (DTOS) program, contract MDA904-93-C-4209. The goal of the study is to assess existing techniques for specifying and verifying modular systems by composing specifications and proofs for the individual system components and to develop new techniques as necessary.

## 1.2 System Overview

The DTOS prototype is an enhanced version of the CMU Mach 3.0 kernel that provides support for a wide variety of security policies by enforcing access decisions provided to it by a *security server*. By developing different security servers, a wide range of policies can be supported by the same DTOS kernel. By developing a security server that allows all accesses, the DTOS kernel behaves essentially the same as the CMU Mach 3.0 kernel. Although this is uninteresting from a security standpoint, it demonstrates the compatibility of DTOS with Mach 3.0.

By using appropriately developed security servers, the DTOS kernel can support interesting security policies such as MLS (multi-level security) and type enforcement. The first security server planned for development is one that enforces a combination of MLS and type enforcement.

Ideally, the evaluation of the resulting system can be done on a component-by-component basis. This would allow system components to be replaced by new components without invalidating the formal analysis as long as the new components satisfy the same requirements. The end goal of the work described in this report is to assess the degree to which this can be accomplished. The results of the study will provide insight into the feasibility of assuring a DTOS-like architecture.

## 1.3 Document Overview

The report is structured as follows:

- Section 1, **Scope**, defines the scope and this overview of the document.
- Section 2, **Applicable Documents**, describes other documents that are relevant to this document.
- Section 3, **Introduction**, provides a brief introduction.
- Section 4, **State Views**, defines the notion of the visible portion of a system state.
- Section 5, **Components**, defines a framework for specifying system components.
- Section 6, **Behaviors**, discusses the notion of a system behavior.

- Section 7, **Satisfaction**, discusses the notion of a component satisfying a property.
- Section 8, **State and Action Predicates**, describes a variant of TLA based on the given definitions of components and behaviors.
- Section 9, **Composition**, provides a definition of specification composition.
- Section 10, **Composition Theorem**, states the composition theorem that allows one to conclude that a composite system satisfies any property satisfied by at least one of its components.
- Section 11, **Distinction between *hidd* and *rely***, provides an example to help clarify the distinction between the *hidd* and *rely* fields in the definition of a component and the reason for having both fields.
- Section 12, **Correctness of Definition**, discusses the correctness of the proposed definition of composition.
- Section 13, **Proving Liveness**, discusses helpful rules for proving liveness properties.
- Section 14, **State and Agent Translation**, discusses a technical detail concerning the type-compatibility of component specifications.
- Section 15, **Composing Two Components**, illustrates the use of translator functions to compose two components having different state and agent types. This section shows that the definition of composition of pairs of components given in previous versions of this report is a special case of the definition of composition of a set of components given here.
- Section 16, **An Example**, introduces a Synergy-like system that we use as an example of how to specify and analyze a system within the composition framework.
- Section 17, **Kernel**, provides a specification of a DTOS-like kernel.
- Section 18, **Common Transitions**, defines various utility functions used in the subsequent component specifications.
- Section 19, **Security Server**, provides a specification of a DTOS-like Security Server.
- Section 20, **Overview of the Cryptographic Subsystem**, provides an overview of the design of the Cryptographic Subsystem that forms the bulk of our example.
- Section 21, **Cryptographic Controller**, provides a specification of the Cryptographic Controller component.
- Section 22, **Protection Tasks**, provides a specification of the Protection Tasks component.
- Section 23, **Key Servers**, provides a specification of the Key Servers component.
- Section 24, **Security Service Usage Policy Server**, provides a specification of the Security Service Usage Policy Server component.
- Section 25, **Cryptographic Client**, provides a specification of the Cryptographic Client component.
- Section 26, **Composing the Components**, demonstrates the application of the framework to compose the components of the Cryptographic Subsystem into a single component specifying the entire system. This section also demonstrates the analysis of the components and system.

- Section 27, **Conclusion**, summarizes the contents and conclusions of this document.
- Section 28, **Notes**, contains a list of acronyms and a glossary for this document.
- Appendix A, **Bibliography**, provides the bibliographical information for the documents referenced in this document.
- Appendix B, **Additional PVS Theories**, includes several simple utility theories used in the report.

In summary, Sections 4–15 provide the general framework, Sections 16–25 provide example component specifications and Section 26 provides an example of how components are composed.

---

*Editorial Note:*
An earlier draft of this report contained a Z specification of an authentication server. This specification was not translated into PVS.

---

# Section 2 Applicable Documents

The following document provides a high level description of the Mach microkernel:

- OSF Mach Kernel Principles [6]

The following documents provide a detailed description of the Mach and DTOS microkernels:

- *OSF Mach 3 Kernel Interface* [5]
- *DTOS Kernel Interface Document* [10]

The following document provides a description of the overall Synergy system:

- Synergy: A Distributed, Microkernel-based Security Architecture [9]

The information in the Cryptographic Subsystem example is extracted from

- R23 Web Pages on the Cryptographic Subsystem [8]

The following documents discuss approaches for composing specifications:

- Conjoining Specifications [1]
- A Lazy Approach to Compositional Verification [11]

A short, and slightly out-of-date, introduction to this report is given in the following research paper:

- A Framework for Composition [4]

The following documents provide background on PVS:

- The PVS Specification Language [7]
- A Tutorial Introduction to PVS [3]

Although an understanding of these documents is desirable, such an understanding is not necessary to understand the majority of this document.

*Section* 3
# Introduction

In this document, we describe a variation of Lamport's TLA specification language[1] and provide a framework for composition of specifications based on the work of Abadi and Lamport[1] and Shankar[11]. Composition is a technique for constructing more complex specifications by building upon simpler specifications. Viewed from the other direction, the composition framework allows the specification and verification of a complex system to be decomposed into the specification and verification of simpler components. Benefits of this approach to assurance are similar to those realized when using a modular approach to software development. In particular, complex reasoning about an overall system can be reduced to simpler reasoning about a collection of components and reusable system components can be defined. After describing the framework, we provide an example of the use of the framework to specify and partially analyze an example.

The framework and example have been formalized in the PVS specification language and the PVS prover has been used to prove all of the stated theorems (with one exception noted in the report). The PVS representation of the framework is generic and can be used to specify and verify other systems as well as the example provided here.

*Section* 4

# State Views

In the Abadi-Lamport theory of composition, a state represents the state of the "entire" universe at a given point in time. Generally, only a small subset of the state is relevant to a given specification. We refer to the relevant portion of the state as the *view* for that specification. Each view is required to be an equivalence relation.[1] We use *VIEWS[X]* to denote the set of all equivalence relations on elements of type *X*.

## THEORY *views*

```
views[X: NONEMPTY_TYPE]: THEORY
  BEGIN

  BASE_RELATIONS: TYPE = [X, X -> bool]

  x, x1, x2, x3, x4: VAR X

  br: VAR BASE_RELATIONS

  VIEWS(br): bool =
    ((FORALL x: br(x, x))
          AND (FORALL x1, x2: br(x1, x2) IMPLIES br(x2, x1))
            AND
          (FORALL x1, x2, x3: br(x1, x2) AND br(x2, x3) IMPLIES br(x1, x3)))

  v1, v2: VAR (VIEWS)

  view_and_prop: THEOREM VIEWS(intersection(v1, v2))

  refl_view: LEMMA v1(x, x)

  sym_view: LEMMA v1(x1, x2) => v1(x2, x1)

  trans_view: LEMMA v1(x1, x2) AND v1(x2, x3) => v1(x1, x3)

  trans_sym_view: LEMMA v1(x1, x2) AND v1(x1, x3) => v1(x2, x3)

  square_view: LEMMA v1(x1, x2) AND v1(x1, x3) AND v1(x2, x4) => v1(x3, x4)

  eq_view1: LEMMA VIEWS(LAMBDA x1, x2: x1 = x2)

  eq_view2: LEMMA (FORALL x1, x2: br(x1, x2) IFF x1 = x2) IMPLIES VIEWS(br)

  END views
```

[1] An equivalence relation is a relation that satisfies the reflexivity, symmetry, and transitivity properties.

*Section* 5
# Components

Abadi and Lamport usually specify components in the following normal form:

$$\exists\, v : \mathit{Init} \wedge \Box N \wedge F$$

where:

- $v$ is a sequence of variables that are internal to the component.
- $\mathit{Init}$ is a state predicate characterizing the initial state,
- $N$ is a disjunction of action predicates characterizing valid transitions (including a no-op step to allow "stuttering"),
- $\Box N$ means predicate $N$ holds for all time, and
- $F$ is a fairness condition that is the conjunction of "weak" and "strong" fairness conditions on steps comprising $N$.

Abadi and Lamport have proven that any property can be written in this form[2] and that the fairness condition in such a specification does not add any safety properties beyond those defined by $\mathit{Init}$ and $N$.

Following the approach in [11], we have chosen to place more structure on the definition of components. The structure we use to represent components consists of:

- *init* — the set of valid starting states for a component; this is directly analogous to Abadi-Lamport's $\mathit{Init}$
- *guar* — the set of transitions representing a component's functionality; each transition is a triple $(st_1, st_2, ag)$ with $st_1$ and $st_2$ denoting the start and end states for the transition and $ag$ indicating the *agent* causing the transition

  An agent is simply a tag associated with transitions. The writer of a specification is free to make use of agents as he sees fit. In the simplest case, there would be a single agent associated with each component that is used to distinguish transitions by one component from transitions by a second component. Another possibility would be to associate agents with threads within the implementation of a component. Yet another possibility would be to associate agents with different operations supported by the component.[3]

[2] The exact statement of this theorem is unclear. This theorem only seems to be true if the property depends on only the visible portion of the state. For example, consider a state having integer fields $x$ and $y$ and suppose the property is that the number of times $x$ has previously changed from a non-zero value to a zero value is less than $y$. Without having a field of the state that records how many times $x$ has become zero, it is not possible to represent this property as simply a set of allowed transitions. For example, to determine whether a transition is allowed from $(x, y) = (2, 3)$ to $(0, 3)$ it is necessary to know whether $x$ has already become 0 three times.

[3] For example, the agent for a open request sent to a file server to open file $f$ could be $(fs, open, f)$.

- *rely* — the set of transitions by the component's environment that can be "tolerated" by the component; in other words, the intent is to prove that the component achieves its desired functionality as long as its environment performs no transitions other than those allowed by *rely*

  The union of *guar* and *rely* is directly analogous to Abadi-Lamport's $N$.

- *cags* — the set of agents associated with the component; the remaining agents are associated with the component's environment

  Although *cags* can be derived from *guar* by simply including the agent for each transition in *guar*, we find it convenient to have the set of component agents explicitly defined.

- *view* — an equivalence relation indicating which portions of the state are visible to the component; any two states related by *view* are equivalent from the standpoint of the component

  In many cases, we define the state so it contains exactly the data visible to the component. Then, *view* can be defined simply as equality of states.

- *hidd* — a set of transitions specifying constraints on the interface the component provides to other components

  *hidd* is analogous to *rely* in that it states assumptions on changes other components make to the system state. Typically, we use *hidd* to capture "syntactic" restrictions such as "no other component accesses data structure $x$" or "other than the component's agents, only agents $ag_1$ and $ag_2$ can access data structure $y$". These constraints describe how portions of a component's state are shared with other components. In contrast, *rely* is typically used to capture "semantic" restrictions. For example, suppose *hidd* indicates that data structures $l$ and $d$ are accessible to a second component. If $l$ denotes a lock protecting $d$, *rely* might be defined so that transitions changing $d$ are only allowed if $l$ is clear. This would capture the semantics of a locking protocol in which $d$ cannot be accessed by others when the component has acquired the lock.

  The distinction between *hidd* and *rely* will be further described in Section 11 were we provide an extended example of why it is valuable to have both concepts in our framework.

- *wfar* — the set of transition classes for which "weak" fairness assumptions are required; the meaning and use of *wfar* is explained in Section 7

  Each transition class is a set of transitions. So, *wfar* is a set of sets of transitions.

- *sfar* — the set of transition classes for which "strong" fairness assumptions are required; the meaning and use of *sfar* is explained in Section 7

  *wfar* and *sfar* are representations of Abadi-Lamport's $F$. They are only needed when the analyst wishes to state and prove "liveness" properties.

We require the following relationships to hold between the various fields:

- *view* is an equivalence relation.

- *init* is non-empty (function *init_restriction*).

  If *init* is empty, then the component can never really execute since it has no valid starting state.

- The agent for each transition in *guar* is an element of *cags* (function $guar\_restriction$).

- Each transition in *rely* is also in *hidd* (function *rely_hidd_restriction*).

  Although *rely* is intended to specify finer-grained assumptions than *hidd*, the assumptions captured by each must be consistent. In particular, *rely* cannot allow a transition that violates the weaker restrictions captured by *hidd*.

- The agent for each transition in *hidd* is not an element of *cags* (function *hidd_restriction*).

  This merely reflects that *hidd* places restrictions on how environment rather than component agents interface with the component. Since *rely* is required to be a subset of *hidd*, this restriction implies that each transition in *rely* is by an agent not in *cags*.

- *cags* is non-empty (function *cags_restriction*).

  If *cags* is empty, then the component can never really execute since it has no agents to cause transitions.

- *rely*, *hidd*, *guar*, and each of the transition sets in *wfar* and *sfar* are each well-defined with respect to *view* (functions *view_rely_restriction*, *view_hidd_restriction*, *view_guar_restriction*, *view_wfar_restriction*, and *view_sfar_restriction*).

  By a set of transitions being well-defined with respect to *view*, we mean that whenever:

  - $(st_1, st_2, ag)$ is in the set of transitions,
  - $st_1$ and $st_3$ are equivalent with respect to *view*, and
  - $st_2$ and $st_4$ are equivalent with respect to *view*,

  then $(st_3, st_4, ag)$ is in the set of transitions, too. Intuitively, the requirement is that any transition that appears the same as one in the set of transitions is itself in the set of transitions.

- *init* is well-defined with respect to *view* (function *view_init_restriction*).

  Here well-defined means that any state that is equivalent (with respect to *view*) to a state in *init* must itself be in *init*.

  If the various elements of a component are not all well-defined with respect to the component's *view*, then the *view* is really not defined correctly. The component's behavior must be completely determined by the data structures visible to it.

- *guar* and *rely* contain all of the *stuttering* steps (functions *guar_stuttering_restriction* and *rely_stuttering_restriction*).

  A stuttering step is a transition in which the start and finish state for the transition are equivalent with respect to the component's *view*. From the component's standpoint, these transitions are no-ops that appear as the system stuttering stuck in a given state. Requiring stuttering steps is important because:

  - In some situations, it is desirable to model a component at varying levels of abstraction and show that the lower-level models are refinements of the higher-level models. Stuttering steps in high-level models serve as placeholders for transitions in low-level models that manipulate data that is not visible at the high-level. So as to not preclude the possibility of later refinement of components, it makes sense to require stuttering steps be included.
  - In the case of *rely* transitions, environment actions that change data visible to the environment but not visible to the component appear as stuttering steps to the component.

We will interpret the state transitions allowed by a component as follows. A transition $(st_1, st_2, ag)$ is allowed by a component $cmp$ if

- $(st_1, st_2, ag)$ is in $guar(cmp)$ (i.e., the component can perform the step), or
- $(st_1, st_2, ag)$ is in $rely(cmp)$ (i.e., the component allows it environment to perform the step).

We use $steps(cmp)$ to denote the set of transitions the component allows.

Note that we define a transition to be a state-state-agent triple rather than a state-state pair. Although the Abadi-Lamport work allows for transitions to be specified in this form, the examples they typically provide specify transitions simply as relations between a starting and final state. The Shankar work completely ignores agents. Our primary area of application is security, and we have found that specifying the agent for each transition is critical to security analysis. When only correctness is of concern, the component that performs a step is irrelevant as long as it is correctly performed. When security is a concern, who causes a transition is just as important as whether the transition is performed correctly.

Note that in the PVS specification, we often use $\lambda$-expressions. These are equivalent to set comprehensions. For example, consider the set:

$$\{ x : T \mid P(x) \}$$

In PVS, a set is equivalent to the predicate that evaluates to true exactly on elements of the set. Thus, the above set is equivalent to:

$$\lambda(x : T) : P(x)$$

Typically, the $\lambda$-expressions appearing in the following specifications are such set comprehensions.

## THEORY *component*

```
component[ST: NONEMPTY_TYPE, AG: NONEMPTY_TYPE]: THEORY
  BEGIN

  IMPORTING views[ST]

  transition: TYPE = [ST, ST, AG]

  TRANSITION_CLASS: TYPE = setof[transition]

  st, st1, st2, st3, st4: VAR ST

  base_comp_t:
      TYPE =
        [# init: setof[ST],
           guar: setof[transition],
           rely: setof[transition],
           hidd: setof[transition],
           cags: setof[AG],
           view: (VIEWS),
           wfar: setof[TRANSITION_CLASS],
           sfar: setof[TRANSITION_CLASS] #]
```

```
bc: VAR base_comp_t

ag: VAR AG

init_restriction(bc): bool = (init(bc) /= emptyset)

guar_restriction(bc): bool =
  (FORALL st1, st2, ag:
     member((st1, st2, ag), guar(bc)) IMPLIES member(ag, cags(bc)))

cags_restriction(bc): bool = (cags(bc) /= emptyset)

rely_restriction(bc): bool =
  (FORALL st1, st2, ag:
     member((st1, st2, ag), rely(bc)) IMPLIES NOT member(ag, cags(bc)))

hidd_restriction(bc): bool =
  (FORALL st1, st2, ag:
     member((st1, st2, ag), hidd(bc)) IMPLIES NOT member(ag, cags(bc)))

tranc: VAR TRANSITION_CLASS

v: VAR (VIEWS)

gen_view_restriction(tranc, v): bool =
  (FORALL ag, st1, st2, st3, st4:
     v(st1, st3) AND v(st2, st4) AND member((st1, st2, ag), tranc)
        IMPLIES member((st3, st4, ag), tranc))

view_rely_restriction(bc): bool = gen_view_restriction(rely(bc), view(bc))

view_hidd_restriction(bc): bool = gen_view_restriction(hidd(bc), view(bc))

view_guar_restriction(bc): bool = gen_view_restriction(guar(bc), view(bc))

view_init_restriction(bc): bool =
  (FORALL st1, st2:
     view(bc)(st1, st2) AND member(st1, init(bc))
        IMPLIES member(st2, init(bc)))

view_wfar_restriction(bc): bool =
  (FORALL tranc:
     member(tranc, wfar(bc)) IMPLIES gen_view_restriction(tranc, view(bc)))

view_sfar_restriction(bc): bool =
  (FORALL tranc:
     member(tranc, sfar(bc)) IMPLIES gen_view_restriction(tranc, view(bc)))

ag_set: VAR setof[AG]

gen_stuttering_restriction(ag_set, tranc, v): bool =
  (FORALL ag, st1, st2:
     member(ag, ag_set) AND v(st1, st2)
        IMPLIES member((st1, st2, ag), tranc))

guar_stuttering_restriction(bc): bool =
  gen_stuttering_restriction(cags(bc), guar(bc), view(bc))

rely_stuttering_restriction(bc): bool =
  gen_stuttering_restriction(complement(cags(bc)), rely(bc), view(bc))

hidd_stuttering_restriction(bc): bool =
  gen_stuttering_restriction(complement(cags(bc)), hidd(bc), view(bc))
```

```
rely_hidd_restriction(bc): bool = subset?(rely(bc), hidd(bc))

comp_t(bc): bool =
  init_restriction(bc)
      AND guar_restriction(bc)
        AND rely_hidd_restriction(bc)
          AND hidd_restriction(bc)
            AND cags_restriction(bc)
              AND view_rely_restriction(bc)
                AND view_hidd_restriction(bc)
                  AND view_guar_restriction(bc)
                    AND view_init_restriction(bc)
                      AND view_wfar_restriction(bc)
                        AND view_sfar_restriction(bc)
                          AND guar_stuttering_restriction(bc)
                            AND rely_stuttering_restriction(bc)

steps(bc): setof[[ST, ST, AG]] =
  (LAMBDA st1, st2, ag: guar(bc)(st1, st2, ag) OR rely(bc)(st1, st2, ag))

c: VAR (comp_t)

component_init: THEOREM init_restriction(c)

component_guar: THEOREM guar_restriction(c)

component_rely_hidd: THEOREM rely_hidd_restriction(c)

component_hidd: THEOREM hidd_restriction(c)

component_rely: THEOREM rely_restriction(c)

component_cags: THEOREM cags_restriction(c)

component_view_rely: THEOREM view_rely_restriction(c)

component_view_hidd: THEOREM view_hidd_restriction(c)

component_view_guar: THEOREM view_guar_restriction(c)

component_view_init: THEOREM view_init_restriction(c)

component_view_wfar: THEOREM view_wfar_restriction(c)

component_view_sfar: THEOREM view_sfar_restriction(c)

component_guar_stuttering: THEOREM guar_stuttering_restriction(c)

component_rely_stuttering: THEOREM rely_stuttering_restriction(c)

component_hidd_stuttering: THEOREM hidd_stuttering_restriction(c)

END component
```

We impose an ordering on components as follows; $cmp\_contains(cmp_1, cmp_2)$ is said to hold whenever:

- each of *init*, *rely*, *hidd*, and *view* for $cmp_1$ is a subset of the analogous entity for $cmp_2$ (for example, $init(cmp_1) \subseteq init(cmp_2)$),

- each of *cags*, *wfar*, and *sfar* for $cmp_2$ is a subset of the analogous entity for $cmp_1$ (for example, $cags(cmp_2) \subseteq cags(cmp_1)$), and
- $cmp_1$'s *guar* is a subset of the set of steps for $cmp_2$ (which is the union of *guar* and *rely* for $cmp_2$).

This imposes a partial order on components. In particular, a component always "contains" itself. We later show that this definition of containment is such that whenever $cmp_1$ is contained in $cmp_2$, then any property that holds for $cmp_2$ holds for $cmp_1$, too. Before doing so, we must first define what is meant by a "property" and what it means for a component to "satisfy" a property.

## THEORY *cmp_contains*

```
cmp_contains[ST: NONEMPTY_TYPE, AG: NONEMPTY_TYPE]: THEORY
  BEGIN

  IMPORTING component[ST, AG]

  cmp1, cmp2, cmp3: VAR (comp_t)

  cmp_contains(cmp1, cmp2): bool =
    subset?(init(cmp1), init(cmp2))
          AND subset?(cags(cmp2), cags(cmp1))
            AND subset?(guar(cmp1), steps(cmp2))
              AND subset?(wfar(cmp2), wfar(cmp1))
                AND subset?(sfar(cmp2), sfar(cmp1))
                  AND subset?(rely(cmp1), rely(cmp2))
                    AND subset?(hidd(cmp1), hidd(cmp2))
                      AND subset?(view(cmp1), view(cmp2))

  cmp_contains_reflexive: THEOREM cmp_contains(cmp1, cmp1)

  cmp_contains_as_guar: THEOREM
        cmp_contains(cmp1, cmp2) AND cmp_contains(cmp2, cmp1)
            IMPLIES subset?(guar(cmp1), guar(cmp2))

  cmp_contains_antisymmetric: THEOREM
        cmp_contains(cmp1, cmp2) AND cmp_contains(cmp2, cmp1)
            IMPLIES cmp1 = cmp2

  cmp_contains_tr_guar: THEOREM
        cmp_contains(cmp1, cmp2) AND cmp_contains(cmp2, cmp3)
            IMPLIES subset?(guar(cmp1), steps(cmp3))

  cmp_contains_transitive: THEOREM
        cmp_contains(cmp1, cmp2) AND cmp_contains(cmp2, cmp3)
            IMPLIES cmp_contains(cmp1, cmp3)

  cmp_contains_po: THEOREM partial_order?(cmp_contains)

  END cmp_contains
```

We define a number of functions and theorems for simplifying the specification of components and the demonstration that the restrictions on a component are satisfied. First, we define $gen\_class(tranc, v)$ to denote the set of transitions that look the same, with respect to a view $v$, as some transition in a set of transitions $tranc$. As an example of how this function might be used, suppose the state type is a record with fields $a$, $b$, $c$, $d$, and $e$ and the view is such that only $a$ and $b$ are visible. Suppose we wish to specify an operation that increments $a$ and leaves $b$ unchanged. A naive approach would be to model a function $f(st)$ that returns a new state in which only $a$ is altered and $a$'s new value is 1 more than its previous value. This specification does not fit with the framework, though, since it constrains the values of $c$, $d$, and $e$ which are not visible to the component. By constraining values that are not visible, the specification violates the requirement that it be well-defined with respect to the view. By applying $gen\_class$ to the set of transitions specified by $f$, any other transitions that are equivalent to those specified by $f$ are added. The resulting set is well-defined with respect to the view. In summary, using $gen\_class$ allows the specifier to write the simpler, naive specification and then extend it to a specification that is well-defined with respect to the view.

Similarly, a naive approach to specification would be to only specify operations that have some effect on the system state. Doing so would not satisfy the requirement that a component be stuttering closed. We use $add\_stuttering(ag\_set, tranc, v)$ to denote the set of transitions that are either:

- elements of $tranc$, or
- stuttering steps (with respect to $v$) by agents in $ag\_set$

Given a set of transitions, a specifier can obtain a stuttering closed set of transitions by using $add\_stuttering$. The function $add\_stuttering\_and\_gen$ combines $add\_stuttering$ and $gen\_class$ by first adding stuttering steps and then adding all equivalent transitions. We prove that:

- ($gen\_class\_view$)[4] $gen\_class(tranc, v)$ returns a set of transitions that is well-defined with respect to $v$

  Proof: $gen\_class$ adds any transitions needed to ensure the set of transitions is well-defined with respect to $v$.

- ($add\_stuttering\_guar$) $add\_stuttering(cags(cmp), tranc, view(cmp))$ returns a set of transitions that satisfies the stuttering requirement on *guar*

  Proof: $add\_stuttering$ adds any missing stuttering steps by component agents.

- ($add\_stuttering\_rely$) $add\_stuttering(complement(cags(cmp)), tranc, view(cmp))$ returns a set of transitions that satisfies the stuttering requirement on *rely*

  Proof: $add\_stuttering$ adds any missing stuttering steps by environment agents.

- ($gen\_class\_preserves\_stuttering$) If a set of transitions, $tranc$, contains all of the stuttering steps with respect to $v$, then $gen\_class(tranc, v)$ also contains all of the stuttering steps with respect to $v$.

  Proof: $gen\_class$ does not remove any transitions.

- ($asag\_stuttering$) $add\_stuttering\_and\_gen(ag\_set, tranc, v)$ contains all stuttering steps with respect to $v$

  Proof: $add\_stuttering$ adds in all of the stuttering steps and $gen\_class$ does not remove any.

[4] The name in parentheses at the beginning of a bulleted item is the name of the theorem in the PVS specification. The remaining text in the first paragraph for a bulleted item is the statement of the theorem. The second paragraph of text for a bulleted item is a sketch of the proof.

- ($asag\_view$) $add\_stuttering\_and\_gen(ag\_set, tranc, v)$ is well-defined with respect to $v$

  Proof: $gen\_class$ always returns a set of transitions that is well-defined with respect to the view.

In summary, the functions defined above can be used to extend a naive specification to a specification that is stuttering closed and well-defined, and the theorems stated above guarantee the results on stuttering closure and well-definedness.

Often, we define a different state type for each of the components we specify. Then, the entire state is visible to the component, and the view relation for the component is simply equality. In other words, two states look the same to the component only if they are exactly the same state. In this case, the stuttering closure and well-definedness requirements (*view_rely_restriction*, *view_hidd_restriction*, *view_guar_restriction*, *view_init_restriction*, *view_wfar_restriction*, and *view_sfar_restriction*) trivially hold. The theorem *component_view_eq_red* asserts this fact.

Often *guar*, *rely*, and *hidd* are specified as a union of different transition classes. For example, *guar* is often specified as the union of the individual transition classes representing the different operations supported by the component. Then, proving the requirements placed on components typically requires stepping through each of the transition classes. Since there are multiple requirements on *guar*, *rely*, and *hidd*, this can require stepping through the individual transition classes multiple times. We now define functions and theorems that allow the requirements to be proved by stepping through the transition classes only once. The general approach is to generate a single condition that is sufficient to establish all of the requirements and then prove that condition for each of the transition classes. The functions are as follows:

- $guar\_reqs\_hold(st_1, st_2, ag, cmp)$ returns true when:
  - $ag$ is an element of $cmp$'s *cags*, and
  - for any $st_3$ and $st_4$ that are equivalent to, respectively, $st_1$ and $st_2$ with respect to $cmp$'s *view*, $(st_3, st_4, ag)$ is an element of $cmp$'s *guar*

  For $cmp$ to be a valid component, these conditions must hold whenever $(st_1, st_2, ag)$ is an element of $cmp$'s *guar*.

- $rely\_reqs\_hold(st_1, st_2, ag, cmp)$ returns true when:
  - $(st_1, st_2, ag)$ is an element of $cmp$'s *hidd*, and
  - for any $st_3$ and $st_4$ that are equivalent to, respectively, $st_1$ and $st_2$ with respect to $cmp$'s *view*, $(st_3, st_4, ag)$ is an element of $cmp$'s *rely*

  For $cmp$ to be a valid component, these conditions must hold whenever $(st_1, st_2, ag)$ is an element of $cmp$'s *rely*.

- $hidd\_reqs\_hold(st_1, st_2, ag, cmp)$ returns true when:
  - $ag$ is not an element of $cmp$'s *cags*, and
  - for any $st_3$ and $st_4$ that are equivalent to, respectively, $st_1$ and $st_2$ with respect to $cmp$'s *view*, $(st_3, st_4, ag)$ is an element of $cmp$'s *hidd*

  For $cmp$ to be a valid component, these conditions must hold whenever $(st_1, st_2, ag)$ is an element of $cmp$'s *hidd*.

The associated theorems are:

- (*guar_reqs_hold_thm*) *guar_reqs_hold* is true for each transition in a transition class, *tranc*, exactly when it is true for each transition in *gen_class*(*tranc*, *view*(*cmp*)).
- (*rely_reqs_hold_thm*) *rely_reqs_hold* is true for each transition in a transition class, *tranc*, exactly when it is true for each transition in *gen_class*(*tranc*, *view*(*cmp*)).
- (*hidd_reqs_hold_thm*) *hidd_reqs_hold* is true for each transition in a transition class, *tranc*, exactly when it is true for each transition in *gen_class*(*tranc*, *view*(*cmp*)).

  These theorems allow reasoning about *guar*, *rely*, and *hidd* for *tranc* to be related to reasoning about *guar*, *rely*, and *hidd* for an extension to make *tranc* well-defined with respect to the view.
- (*guar_reqs_sufficient*) *guar_reqs_hold* is true for each transition in a component's *guar* exactly when the component satisfies the *view_guar_restriction* and *guar_restriction* requirements on components.
- (*rely_reqs_sufficient*) *rely_reqs_hold* is true for each transition in a component's *rely* exactly when the component satisfies the *view_rely_restriction* and *rely_hidd_restriction* requirements on components.
- (*hidd_reqs_sufficient*) *hidd_reqs_hold* is true for each transition in a component's *hidd* exactly when the component satisfies the *view_hidd_restriction* and *hidd_restriction* requirements on components.

  These theorems allow proofs of some of the requirements on components to be reduced to proofs of *guar_reqs_sufficient*, *rely_reqs_sufficient*, and *hidd_reqs_sufficient*.

## THEORY *component_aux*

```
component_aux[ST: NONEMPTY_TYPE, AG: NONEMPTY_TYPE]: THEORY
  BEGIN

  IMPORTING component[ST, AG]

  bc: VAR base_comp_t

  st1, st2, st3, st4: VAR ST

  ag: VAR AG

  ag_set: VAR setof[AG]

  tranc: VAR setof[transition]

  v: VAR (VIEWS)

  gen_class(tranc, v): setof[transition] =
    (LAMBDA st1, st2, ag:
       (EXISTS st3, st4:
          member((st3, st4, ag), tranc) AND v(st1, st3) AND v(st2, st4)))

  gen_class_view: THEOREM gen_view_restriction(gen_class(tranc, v), v)

  add_stuttering(ag_set, tranc, v): setof[transition] =
    (LAMBDA st1, st2, ag:
       member((st1, st2, ag), tranc) OR (member(ag, ag_set) AND v(st1, st2)))
```

```
add_stuttering_guar: THEOREM
      guar(bc) = add_stuttering(cags(bc), tranc, view(bc))
          IMPLIES guar_stuttering_restriction(bc)

add_stuttering_rely: THEOREM
      rely(bc) = add_stuttering(complement(cags(bc)), tranc, view(bc))
          IMPLIES rely_stuttering_restriction(bc)

gen_class_preserves_stuttering: THEOREM
      gen_stuttering_restriction(ag_set, tranc, v)
          IMPLIES gen_stuttering_restriction(ag_set, gen_class(tranc, v), v)

add_stuttering_and_gen(ag_set, tranc, v): setof[transition] =
  gen_class(add_stuttering(ag_set, tranc, v), v)

asag_stuttering: THEOREM
      gen_stuttering_restriction(ag_set,
                                 add_stuttering_and_gen(ag_set, tranc, v),
                                 v)

asag_view: THEOREM
      gen_view_restriction(add_stuttering_and_gen(ag_set, tranc, v), v)

view_eq(bc) : bool =
  (FORALL st1, st2: view(bc)(st1, st2) IFF st1 = st2)

component_view_eq_red: THEOREM
      view_eq(bc)
          IMPLIES view_rely_restriction(bc)
            AND view_hidd_restriction(bc)
              AND view_guar_restriction(bc)
                AND view_init_restriction(bc)
                  AND view_wfar_restriction(bc) AND view_sfar_restriction(bc)

component_view_eq_thm: THEOREM
  view_eq(bc)
   AND init_restriction(bc)
      AND guar_restriction(bc)
        AND rely_hidd_restriction(bc)
          AND hidd_restriction(bc)
            AND cags_restriction(bc)
              AND guar_stuttering_restriction(bc)
                AND rely_stuttering_restriction(bc)
  => comp_t(bc)

guar_reqs_hold(st1, st2, ag, bc): bool =
  (member(ag, cags(bc))
       AND
     (FORALL st3, st4:
        view(bc)(st1, st3) AND view(bc)(st2, st4)
            IMPLIES member((st3, st4, ag), guar(bc))))

guar_reqs_hold_thm: THEOREM
      (FORALL st1, st2, ag:
         member((st1, st2, ag), tranc)
             IMPLIES guar_reqs_hold(st1, st2, ag, bc))
          IFF
        (FORALL st1, st2, ag:
           member((st1, st2, ag), gen_class(tranc, view(bc)))
               IMPLIES guar_reqs_hold(st1, st2, ag, bc))

guar_reqs_sufficient: THEOREM
      (FORALL st1, st2, ag:
```

```
      member((st1, st2, ag), guar(bc))
          IMPLIES guar_reqs_hold(st1, st2, ag, bc))
       IFF (view_guar_restriction(bc) AND guar_restriction(bc))

rely_reqs_hold(st1, st2, ag, bc): bool =
  (member((st1, st2, ag), hidd(bc))
       AND
     (FORALL st3, st4:
        view(bc)(st1, st3) AND view(bc)(st2, st4)
            IMPLIES member((st3, st4, ag), rely(bc))))

rely_reqs_sufficient: THEOREM
     (FORALL st1, st2, ag:
        member((st1, st2, ag), rely(bc))
            IMPLIES rely_reqs_hold(st1, st2, ag, bc))
       IFF (view_rely_restriction(bc) AND rely_hidd_restriction(bc))

hidd_reqs_hold(st1, st2, ag, bc): bool =
  (NOT member(ag, cags(bc))
       AND
     (FORALL st3, st4:
        view(bc)(st1, st3) AND view(bc)(st2, st4)
            IMPLIES member((st3, st4, ag), hidd(bc))))

hidd_reqs_hold_thm: THEOREM
     (FORALL st1, st2, ag:
        member((st1, st2, ag), tranc)
            IMPLIES hidd_reqs_hold(st1, st2, ag, bc))
       IFF
      (FORALL st1, st2, ag:
         member((st1, st2, ag), gen_class(tranc, view(bc)))
             IMPLIES hidd_reqs_hold(st1, st2, ag, bc))

rely_reqs_hold_thm: THEOREM
     view_hidd_restriction(bc)
       IMPLIES
      ((FORALL st1, st2, ag:
          member((st1, st2, ag), tranc)
              IMPLIES rely_reqs_hold(st1, st2, ag, bc))
          IFF
        (FORALL st1, st2, ag:
           member((st1, st2, ag), gen_class(tranc, view(bc)))
               IMPLIES rely_reqs_hold(st1, st2, ag, bc)))

hidd_reqs_sufficient: THEOREM
     (FORALL st1, st2, ag:
        member((st1, st2, ag), hidd(bc))
            IMPLIES hidd_reqs_hold(st1, st2, ag, bc))
       IFF (view_hidd_restriction(bc) AND hidd_restriction(bc))

END component_aux
```

*Section* 6
# Behaviors

The basic construct in TLA is *behaviors*. A behavior consists of an infinite sequence of states $st_0, st_1, st_2, \ldots$ and an infinite sequence of agents $ag_0, ag_1, \ldots$. Each element of the sequence of states represents a snapshot of the system state as time progresses. The sequence of agents indicates the entity responsible for each given state transition. In the specification of the framework, we represent each sequence as a function from the set of natural numbers to the elements of the sequence. We define the type *trace_t* to denote a record containing the following fields:

- *sts* — denotes the sequence of states; *sts(i)* is the $i^{th}$ state
- *ags* — denotes the sequence of agents; *ags(i)* is the agent causing the transition from the $i^{th}$ state to the $i{+}1^{th}$ state

A *behavior predicate* is an assertion about a behavior. We represent each predicate by the set of behaviors satisfying the predicate. We use *prop_t* to denote the set of all behavior predicates.

## THEORY *props*

```
props[ST: NONEMPTY_TYPE, AG: NONEMPTY_TYPE]: THEORY
  BEGIN

  trace_t: TYPE = [# sts: [nat -> ST], ags: [nat -> AG] #]

  prop_t: TYPE = setof[trace_t]

  END props
```

10

# Section 7 Satisfaction

A component is modeled as a set of behaviors. A predicate $p$ holds in a behavior $beh$ when $beh \in p$. A system component $cmp$ is said to satisfy a behavior predicate if each element of the set of behaviors modeling the system component satisfies the behavior predicate. We write *satisfies(cmp,p)* to indicate that $cmp$ satisfies $p$. We use *prop_for(cmp)* to denote the set of behaviors modeling $cmp$. This set consists of all behaviors such that:

- the behavior starts in a state belonging to $init$ for $cmp$ (function *initial_okay*),
- each transition in the behavior is either an element of *rely* or *guar* for *cmp* (function *steps_okay*),
- and the behavior satisfies the weak and strong fairness assumptions stated in *wfar* and *sfar* for *cmp* (functions *is_wfar* and *is_sfar*).

The weak fairness assumptions are denoted by the transition classes comprising *wfar*. Given a transition class *tranc*, assuming weak fairness of *tranc* means each behavior of the component must be such that either:

- *tranc* is infinitely often disabled in the behavior, or
- *tranc* occurs infinitely often.

A transition class is said to be enabled in a state $st$ if there exists a $st_2$ and $ag$ such that $(st, st_2, ag)$ is an element of the transition class. Intuitively, a transition class is enabled in a state if some element of the transition class has that state as its starting state. A transition class is disabled in a state whenever it is not enabled in that state. A transition class is said to occur at some point in a behavior if the transition at that point of the behavior is an element of the transition class.

So, a behavior satisfies weak fairness for a transition class if points are repeatedly reached in the behavior where either the transition class is disabled or occurs. The motivation for this notion of fairness is that if a transition class is not forever disabled, it should eventually occur.

Strong fairness represents a stronger notion of fairness in which a behavior is acceptable only if each of the transition classes specified in *sfar* is either:

- eventually stuck disabled forever, or
- occurs infinitely often.

To understand the difference between weak and strong fairness, consider a behavior that oscillates between states in which a transition class is enabled and states in which it is disabled. Suppose the transition class does not occur infinitely often in the behavior. Then:

- The behavior is consistent with the weak fairness assumption for the transition class because the class is repeatedly disabled.

- The behavior is inconsistent with the strong fairness assumption for the transition class because it is never stuck disabled forever (since it repeatedly becomes enabled) and the transition class does not occur infinitely often.

In summary, *wfar* and *sfar* filter out certain behaviors that are judged to be unfair because transition classes that are prepared to occur are denied their opportunity to occur. Their use in the definition of the set of behaviors for a component results in those behaviors being "fair" as defined by the analyst. The notion of fairness is only important when proving *liveness* properties. Intuitively, a *liveness* property requires that some condition eventually hold. Typically, these properties are proven by demonstrating a transition class that results in the condition holding. To complete the proof, though, it is necessary to know the transition class is eventually given an opportunity to occur. In particular, a behavior which stutters forever is unlikely to result in any interesting conditions holding. By excluding such behaviors as unfair, liveness properties can be proven. The fairness assumptions must somehow be justified when the model is mapped to the implementation. For example, an argument might be given that the scheduler for events that occur in the implementation schedules events fairly.

Now that we have completed the discussion of what it means for a component to satisfy a property, we can state the following theorem:

> *Suppose* $cmp_1$ *and* $cmp_2$ *are components such* $cmp_2$ *"contains"* $cmp_1$. *Then, any property satisfied by* $cmp_2$ *is also satisfied by* $cmp_1$.

Here, "contains" is component containment as defined in Section 5. The proof of the theorem is as follows:

- If a property $p$ is satisfied by $cmp_2$, then every behavior of $cmp_2$ satisfies $p$.
- Since $cmp_2$ contains $cmp_1$, the set of behaviors for $cmp_2$ is at least as big as the set of behaviors of $cmp_1$.
- Thus, every behavior of $cmp_1$ satisfies $p$. By definition, this means $cmp_1$ satisfies $p$.

This theorem is the key step in the proof of the composition theorem in Section 10. The composition theorem identifies conditions sufficient to ensure that properties of system components are preserved as the components are composed with other components. Given the preceding theorem, the composition theorem can be proven by demonstrating that the sufficient conditions ensure the composite is "contained" in the components comprising the composition.

## THEORY *cprops*

```
cprops[ST: NONEMPTY_TYPE, AG: NONEMPTY_TYPE]: THEORY
  BEGIN

  IMPORTING cmp_contains[ST, AG]

  IMPORTING props[ST, AG]

  cmp, cmp1, cmp2: VAR (comp_t)

  t: VAR trace_t

  n, i, j, k, l: VAR nat
```

```
p: VAR prop_t

st, st1, st2: VAR ST

ag: VAR AG

tranc, tranc1: VAR TRANSITION_CLASS

initial_okay(cmp, t): bool = member((sts(t)(0)), init(cmp))

steps_okay(cmp, t): bool =
  (FORALL n: member((sts(t)(n), sts(t)(n + 1), ags(t)(n)), steps(cmp)))

enabled(tranc, st1): bool = (EXISTS st2, ag: member((st1, st2, ag), tranc))

is_wfar(cmp, t): bool =
  (FORALL tranc:
     member(tranc, wfar(cmp))
        IMPLIES
      (FORALL i:
         (EXISTS j:
            j > i
               AND
             (NOT enabled(tranc, sts(t)(j))
                 OR
               member((sts(t)(j), sts(t)(j + 1), ags(t)(j)), tranc)))))

is_sfar(cmp, t): bool =
  (FORALL tranc:
     member(tranc, sfar(cmp))
        IMPLIES
      (FORALL i:
         (EXISTS j:
            j > i
               AND
             ((FORALL k: k >= j IMPLIES NOT enabled(tranc, sts(t)(k)))
                 OR
               (EXISTS l:
                  l >= j
                     AND
                   member((sts(t)(l), sts(t)(l + 1), ags(t)(l)),
                          tranc))))))

prop_for(cmp): prop_t =
  (LAMBDA t:
     initial_okay(cmp, t)
        AND steps_okay(cmp, t) AND is_wfar(cmp, t) AND is_sfar(cmp, t))

satisfies(cmp, p): bool = (FORALL t: prop_for(cmp)(t) IMPLIES p(t))

initial_okay_prop: THEOREM
      (FORALL st: member(st, init(cmp1)) IMPLIES member(st, init(cmp2)))
         AND initial_okay(cmp1, t)
         IMPLIES initial_okay(cmp2, t)

steps_okay_prop: THEOREM
      (FORALL st1, st2, ag:
         member((st1, st2, ag), steps(cmp1))
            IMPLIES member((st1, st2, ag), steps(cmp2)))
         AND steps_okay(cmp1, t)
         IMPLIES steps_okay(cmp2, t)

is_wfar_prop: THEOREM
```

```
        (FORALL tranc:
           member(tranc, wfar(cmp2)) IMPLIES member(tranc, wfar(cmp1)))
            AND is_wfar(cmp1, t)
            IMPLIES is_wfar(cmp2, t)

  is_sfar_prop: THEOREM
        (FORALL tranc:
           member(tranc, sfar(cmp2)) IMPLIES member(tranc, sfar(cmp1)))
            AND is_sfar(cmp1, t)
            IMPLIES is_sfar(cmp2, t)

  satisfies_prop: THEOREM
        (FORALL st1, st2, ag:
           member((st1, st2, ag), steps(cmp1))
               IMPLIES member((st1, st2, ag), steps(cmp2)))
            AND
          (FORALL st: member(st, init(cmp1)) IMPLIES member(st, init(cmp2)))
              AND
            (FORALL tranc:
               member(tranc, wfar(cmp2)) IMPLIES member(tranc, wfar(cmp1)))
                AND
              (FORALL tranc:
                 member(tranc, sfar(cmp2)) IMPLIES member(tranc, sfar(cmp1)))
                  AND satisfies(cmp2, p)
            IMPLIES satisfies(cmp1, p)

  satisfies_contains_prop: THEOREM
        satisfies(cmp2, p) AND cmp_contains(cmp1, cmp2)
            IMPLIES satisfies(cmp1, p)

  END cprops
```

---

It is interesting to note that $prop\_for(cmp)$ is closed with respect to $cmp$'s $view$. To formalize this, we define $beh\_equiv(v)$ to be an equivalence relation on behaviors such that $b_1$ is considered equivalent to $b_2$ whenever the following hold for each $i$:

- agent $i$ of $b_1$ is the same as agent $i$ of $b_2$,
- state $i$ of $b_1$ is equivalent to state $i$ of $b_2$ with respect to $v$.

Then, the closure property is asserted in the theorem:

> ($beh\_equiv\_prop$) If $b_1$ and $b_2$ are equivalent with respect to $cmp$'s $view$, then $b_1$ is an element of $prop\_for(cmp)$ if and only if $b_2$ is, too.

This theorem is nice since it implies that properties of a component are dependent only on portion's of the state visible to the component. It would be disconcerting if it were possible to prove a component guaranteed properties about portions of the state that are not visible to the component.

## THEORY *beh_equiv*

---

```
beh_equiv[ST: NONEMPTY_TYPE, AG: NONEMPTY_TYPE]: THEORY
  BEGIN
```

```
IMPORTING cprops[ST, AG]

IMPORTING views[trace_t[ST, AG]]

b, b1, b2, b3: VAR trace_t

p: VAR prop_t

v: VAR (VIEWS[ST])

i: VAR nat

cmp: VAR (comp_t)

tranc: VAR setof[transition]

beh_equiv(v)(b1, b2): bool =
  (FORALL i: v(sts(b1)(i), sts(b2)(i)) AND ags(b1)(i) = ags(b2)(i))

beh_equiv_is_refl: THEOREM beh_equiv(v)(b, b)

beh_equiv_is_sym: THEOREM beh_equiv(v)(b1, b2) IMPLIES beh_equiv(v)(b2, b1)

beh_equiv_is_trans: THEOREM
      beh_equiv(v)(b1, b2) AND beh_equiv(v)(b2, b3)
          IMPLIES beh_equiv(v)(b1, b3)

beh_equiv_is_equiv: THEOREM VIEWS(beh_equiv(v))

beh_equiv_init: THEOREM
      beh_equiv(view(cmp))(b1, b2) AND initial_okay(cmp, b1)
          IMPLIES initial_okay(cmp, b2)

beh_equiv_gen_steps: THEOREM
      beh_equiv(v)(b1, b2)
          AND gen_view_restriction(tranc, v)
            AND member((sts(b1)(i), sts(b1)(i + 1), ags(b1)(i)), tranc)
          IMPLIES member((sts(b2)(i), sts(b2)(i + 1), ags(b2)(i)), tranc)

beh_equiv_steps: THEOREM
      beh_equiv(view(cmp))(b1, b2) AND steps_okay(cmp, b1)
          IMPLIES steps_okay(cmp, b2)

beh_equiv_enabled: THEOREM
      beh_equiv(v)(b1, b2)
          AND gen_view_restriction(tranc, v) AND enabled(tranc, sts(b1)(i))
          IMPLIES enabled(tranc, sts(b2)(i))

beh_equiv_wfar: THEOREM
      beh_equiv(view(cmp))(b1, b2) AND is_wfar(cmp, b1)
          IMPLIES is_wfar(cmp, b2)

beh_equiv_sfar: THEOREM
      beh_equiv(view(cmp))(b1, b2) AND is_sfar(cmp, b1)
          IMPLIES is_sfar(cmp, b2)

beh_equiv_prop_help: THEOREM
      beh_equiv(view(cmp))(b1, b2) AND member(b1, prop_for(cmp))
          IMPLIES member(b2, prop_for(cmp))

beh_equiv_prop: THEOREM
      beh_equiv(view(cmp))(b1, b2)
          IMPLIES (member(b1, prop_for(cmp)) IFF member(b2, prop_for(cmp)))
```

```
property(p, v): bool =
  (FORALL b1, b2:
     beh_equiv(v)(b1, b2) IMPLIES (member(b1, p) IFF member(b2, p)))

cmp_property(p, cmp): bool = property(p, view(cmp))

END beh_equiv
```

*Section* 8
# State and Action Predicates

In general, we attempt to perform analysis in terms of *state predicates* and *action predicates* and use functions defined below to translate the analyzed predicates into behavior predicates.

A state predicate is an assertion about a state. We represent each predicate by the set of states satisfying the predicate. The type $STATE\_PRED$ denotes the set of all state predicates. We use $init\_satisfies(cmp, sp)$ to denote that $sp$ holds in each of $cmp$'s initial states.

An action predicate is an assertion about state transitions. We represent each predicate by the set of triples $(st_1, st_2, ag)$ satisfying the predicate. Intuitively, the meaning of $(st_1, st_2, ag)$ belonging to the set representing an action predicate is that the action predicate allows an action by $ag$ to cause a state transition from $st_1$ to $st_2$. The type $ACTION\_PRED[ST, AG]$ denotes the set of all action predicates. We use $steps\_satisfy(cmp, ap)$ to denote that each transition ($guar$ and $rely$) allowed by $cmp$ satisfies $ap$.

We say that a state predicate is held *stable* by a transition if whenever it holds in a given state, it holds in any state reachable from that state by the transition. Given a state predicate $sp$, there is an associated action predicate $stable(sp)$ denoting the set of transitions that hold $sp$ stable.

Given a behavior predicate $p$, we use $always(p)$ to denote the behavior predicate representing that $p$ "always holds". The formal definition is that $always(p)$ contains a behavior $t$ if each "tail" of $t$ satisfies $p$. A tail of $t$ is any behavior resulting from the removal of a finite number of steps from the beginning of $t$. Similarly, we use $eventually(p)$ to denote the behavior predicate representing that $p$ "eventually holds". Rather than requiring $p$ hold for every tail, it requires that $p$ hold for at least one tail.

To allow us to reduce reasoning about behavior predicates to reasoning about state and action predicates, we define:

- $stbp(sp)$ to denote the behavior predicate representing that state predicate $sp$ holds in the initial state
- $atbp(ap)$ to denote the behavior predicate representing that action predicate $ap$ is satisfied by each transition

Given these functions, we can define:

- $alwayss(sp)$ to denote $always(stbp(sp))$

  We prove ($alwayss\_prop$) that $alwayss(sp)$ denotes the set of behaviors such that each state in the behavior satisfies $sp$.
- $eventuallys(sp)$ to denote $eventually(stbp(sp))$

  We prove ($eventuallys\_prop$) that $eventuallys(sp)$ denotes the set of behaviors such that some state in the behavior satisfies $sp$.
- $alwaysa(ap)$ to denote $always(atbp(ap))$

  We prove ($alwaysa\_prop$) that $alwaysa(ap)$ denotes the set of behaviors such that each transition in the behavior satisfies $ap$.

- $eventuallya(ap)$ to denote $eventually(atbp(ap))$

  We prove ($eventuallya\_prop$) that $eventuallya(ap)$ denotes the set of behaviors such that some transition in the behavior satisfies $ap$.

The standard logical operators can be defined on the various types of predicates. For example, $aandas(ap, sp)$ can be defined as the action predicate representing that $ap$ holds for a transition and $sp$ holds for the starting state of the transition. We define the following functions in addition to $aandas$:

- $aand(ap_1, ap_2)$ denotes the action predicate representing that both $ap_1$ and $ap_2$ hold.
- $aimplies(ap_1, ap_2)$ denotes the action predicate representing that any transition satisfying $ap_1$ also satisfies $ap_2$.
- $sand(sp_1, sp_2)$ denotes the state predicate representing that both $sp_1$ and $sp_2$ hold.
- $sor(sp_1, sp_2)$ denotes the state predicate representing that at least one of $sp_1$ and $sp_2$ hold.
- $simplies(sp_1, sp_2)$ denotes the state predicate representing that any state satisfying $sp_1$ also satisfies $sp_2$.

We prove the following theorems for reasoning about predicates:[5]

- ($inv1$) *If sp holds initially and is stable, then sp always holds.*
- ($inv2$) *If ap is satisfied by each transition, then ap always holds.*
- ($inv3$ and $inv4$) *If ap always holds and sp always holds, then aandas(ap,sp) always holds.*
- ($inv5$) *If ap1 always holds and ap2 always holds, then aand(ap1,ap2) always holds.*
- ($inv6$) *If sp1 always holds and sp2 always holds, then sand(sp1,sp2) always holds.*
- ($always\_and$) *If behavior predicates p1 and p2 both hold, then their intersection holds.*
- ($always\_aimplies$) *If ap1 always holds and ap1 implies ap2, then ap2 always holds.*

Note that $inv1$ is the proof rule commonly used to prove that every reachable state of a system satisfies a given state predicate. First, the state predicate is shown to hold in any initial state. Then, the state predicate is shown to be held invariant (stable) by each possible transition.

## THEORY *preds*

```
preds[ST: NONEMPTY_TYPE, AG: NONEMPTY_TYPE]: THEORY
  BEGIN

  IMPORTING cprops[ST, AG]

  STATE_PRED: TYPE = setof[ST]

  sp, sp1, sp2: VAR STATE_PRED
```

[5] Similar theorems could be stated and proved to cover all of the logical operators and types of predicates. Our approach has been to add such theorems as necessary for the examples we work rather than attempt to define a complete set.

```
cmp: VAR (comp_t[ST, AG])

st, st1, st2: VAR ST

ag: VAR AG

init_satisfies(cmp, sp): bool = (FORALL st: init(cmp)(st) IMPLIES sp(st))

ACTION_PRED: TYPE = setof[[ST, ST, AG]]

ap, ap1, ap2: VAR ACTION_PRED

steps_satisfy(cmp, ap): bool =
  (FORALL st1, st2, ag:
      (guar(cmp)(st1, st2, ag) OR rely(cmp)(st1, st2, ag))
          IMPLIES ap(st1, st2, ag))

stable(sp): ACTION_PRED = (LAMBDA st1, st2, ag: sp(st1) IMPLIES sp(st2))

t: VAR trace_t

p: VAR prop_t

i, j: VAR nat

shift(i, t): trace_t =
  (# sts := (LAMBDA j: sts(t)(i + j)), ags := (LAMBDA j: ags(t)(i + j)) #)

always(p): prop_t = (LAMBDA t: (FORALL i: p(shift(i, t))))

eventually(p): prop_t = (LAMBDA t: (EXISTS i: p(shift(i, t))))

stbp(sp): prop_t = (LAMBDA t: sp(sts(t)(0)))

atbp(ap): prop_t = (LAMBDA t: ap(sts(t)(0), sts(t)(1), ags(t)(0)))

alwayss(sp): prop_t = always(stbp(sp))

eventuallys(sp): prop_t = eventually(stbp(sp))

alwayss_prop: THEOREM alwayss(sp) = (LAMBDA t: (FORALL i: sp(sts(t)(i))))

eventuallys_prop: THEOREM
        eventuallys(sp) = (LAMBDA t: (EXISTS i: sp(sts(t)(i))))

alwaysa(ap): prop_t = always(atbp(ap))

eventuallya(ap): prop_t = eventually(atbp(ap))

alwaysa_prop: THEOREM
        alwaysa(ap)
            = (LAMBDA t: (FORALL i: ap(sts(t)(i), sts(t)(i + 1), ags(t)(i))))

eventuallya_prop: THEOREM
        eventuallya(ap)
            = (LAMBDA t: (EXISTS i: ap(sts(t)(i), sts(t)(i + 1), ags(t)(i))))

inv1: THEOREM
        init_satisfies(cmp, sp) AND steps_satisfy(cmp, stable(sp))
            IMPLIES satisfies(cmp, alwayss(sp))

inv2: THEOREM steps_satisfy(cmp, ap) IMPLIES satisfies(cmp, alwaysa(ap))
```

```
aandas(ap, sp): ACTION_PRED =
  (LAMBDA st1, st2, ag: ap(st1, st2, ag) AND sp(st1))

inv3: THEOREM
      intersection(alwaysa(ap), alwayss(sp)) = alwaysa(aandas(ap, sp))

inv4: THEOREM
      intersection(alwayss(sp), alwaysa(ap)) = alwaysa(aandas(ap, sp))

aand(ap1, ap2): ACTION_PRED =
  (LAMBDA st1, st2, ag: ap1(st1, st2, ag) AND ap2(st1, st2, ag))

aimplies(ap1, ap2): ACTION_PRED =
  (LAMBDA st1, st2, ag: ap1(st1, st2, ag) IMPLIES ap2(st1, st2, ag))

inv5: THEOREM
      intersection(alwaysa(ap1), alwaysa(ap2)) = alwaysa(aand(ap1, ap2))

sand(sp1, sp2): STATE_PRED = (LAMBDA st: sp1(st) AND sp2(st))

sor(sp1, sp2): STATE_PRED = (LAMBDA st: sp1(st) OR sp2(st))

simplies(sp1, sp2): STATE_PRED = (LAMBDA st: sp1(st) IMPLIES sp2(st))

inv6: THEOREM
      intersection(alwayss(sp1), alwayss(sp2)) = alwayss(sand(sp1, sp2))

p1, p2: VAR prop_t

always_and: THEOREM
      (satisfies(cmp, p1) AND satisfies(cmp, p2))
          = satisfies(cmp, intersection(p1, p2))

always_aimplies: THEOREM
      satisfies(cmp, alwaysa(ap1))
          AND (FORALL st1, st2, ag: aimplies(ap1, ap2)(st1, st2, ag))
          IMPLIES satisfies(cmp, alwaysa(ap2))

END preds
```

---

Theory *more_preds* generalizes the concepts of stability to conditional stability (*stable_assuming*). It also provides several theorems for reasoning with conditional stability.

## THEORY *more_preds*

---

```
more_preds[ST: NONEMPTY_TYPE, AG: NONEMPTY_TYPE]: THEORY
BEGIN

  IMPORTING unity[ST,AG]


  sp, sp1, sp2: VAR STATE_PRED

  cmp: VAR (comp_t[ST, AG])

  st, st1, st2: VAR ST

  p, p1, p2: VAR prop_t[ST,AG]
```

```
  ag: VAR AG

  stable_assuming(sp1, sp2): ACTION_PRED
    = (LAMBDA st1, st2, ag: sp1(st1) and sp1(st2) and sp2(st1)
        IMPLIES sp2(st2))

  pimplies_always: THEOREM
    init_satisfies(cmp, simplies(sp1, sp2))
        AND steps_satisfy(cmp, stable_assuming(sp1, sp2))
    => satisfies(cmp, pimplies(alwayss(sp1),alwayss(sp2)))

  init_simplies: THEOREM
    init_satisfies(cmp, sp2)
        => init_satisfies(cmp, simplies(sp1, sp2))

  satisfies_modus_ponens: THEOREM
    satisfies(cmp, p1) AND satisfies(cmp, pimplies(p1, p2))
      => satisfies(cmp, p2)

END more_preds
```

Section 9
# Composition

First we discuss how our definition of composition relates to the Abadi-Lamport and Shankar approaches. In doing so, we will allude to our as yet unstated definition of composition. After the discussion, we explicitly state our definition of composition.

## 9.1 Relation to Prior Work

The approach we use for combining specifications is a hybrid of the approaches used by Abadi-Lamport and Shankar. In the Abadi-Lamport work, components with no fairness assumptions are simply properties with the normal form $\exists\, v : I \wedge \Box N$. Composition is defined simply to be conjunction; the composition of $\exists\, v_1 : I_1 \wedge \Box N_1$ with $\exists\, v_2 : I_2 \wedge \Box N_2$ is

$$(\exists\, v_1 : I_1 \wedge \Box N_1) \wedge (\exists\, v_2 : I_2 \wedge \Box N_2).$$

Abadi and Lamport demonstrate that under the correct assumptions regarding free and quantified variables in the component specifications, this conjunction can essentially be rewritten in the form

$$\exists\, v : (I_1 \wedge I_2) \wedge \Box[N_1 \wedge N_2]$$

and in the form

$$\exists\, v : (I_1 \wedge I_2) \wedge \Box[\widehat{N_1} \vee \widehat{N_2}]$$

where $\widehat{N_i}$ is a formula denoting an $N_i$ transition where the variables in $v_{3-i}$ do not change value.

In the Shankar approach, components are specified in terms of a tuple $(init, guar, rely)$ and composition is defined as:

$$(init_1 \wedge init_2, (guar_1 \vee rely_2) \wedge (guar_2 \vee rely_1), rely_1 \wedge rely_2)$$

$N$ in the Abadi-Lamport approach corresponds to $guar \vee rely$ in the Shankar approach. Thus, $N_1 \wedge N_2$ corresponds to:

$$(guar_1 \wedge rely_2) \vee (guar_2 \wedge rely_1) \vee (rely_1 \wedge rely_2) \vee (guar_1 \wedge guar_2)$$

The first two terms correspond to Shankar's definition of $guar$ for the composite while the third term corresponds to Shankar's definition of $rely$ for the composite. So, other than the last term, both definitions are essentially the same. Typically, the steps by each component are disjoint so the last term does not contribute anything. In these cases, the two definitions are essentially the same.

Our definition of composition is similar but slightly different. Reasons for the differences include:

- While the Shankar approach ignores fairness, we follow the Abadi-Lamport approach for addressing fairness.

- We have made *view*, *cags*, and *hidd* explicit parts of the definition of a component, so we need to define *view*, *cags*, and *hidd* for the composite system in terms of the component systems.

- In the definition of *guar* for the composite, we replace $rely_1$ in Shankar's definition with $hidd_1$ and $rely_2$ with $hidd_2$.

  The *hidd* sets are related to *conditional implementation* in Abadi-Lamport where disjointness conditions are added asserting that the outputs of different components do not change simultaneously. Along with concerns about serializability, this reflects the fact that when combining two implemented systems the behavior of the combination depends upon how they were combined. For example, we would expect very different behavior in the case where the data segments of the two components are overlaid versus that where the address spaces are entirely separate. Abadi and Lamport assert that such assumptions should not be included in the specification of a component of an open system; we want to be able to combine the component with other components in more or less arbitrary ways and then reason about the results. By making *hidd* part of a component, we have violated this goal of Abadi and Lamport.

  Our main motivation for making *hidd* part of the definition of a component is to ensure the well-definedness of a composite's *guar* with respect to its *view*. Since well-definedness of *guar* is part of the definition of a component, this is essential to ensuring that the composite of a collection of components is itself a component. When *hidd* is separate from a component, then the proper choice of *hidd* is often dependent on what the component will be composed with. Then, each time a component is composed with different components, it could be necessary to show that the resulting composition's *guar* is well-defined with respect to its *view*. In our approach where *hidd* is part of the definition of a component, the well-definedness of the composite's *guar* with respect to its *view* is guaranteed regardless of what other components are included in the composition.

  The disadvantage of our approach is that if there is a desire to change the manner in which a component interfaces with other components, it is necessary to change the *hidd* portion of the specifications of the components. In the Abadi-Lamport approach, the specifications of the components do not need to be changed since *hidd* is specified separately. So far, we have not found this to be a serious disadvantage. A change in the manner in which components interact is a fairly significant design change for which it is not unreasonable to expect the specifications of components to change.

  In earlier versions of the framework, we included a field called *priv* in the definition of a component that indicated which data is private to a component and therefore not modifiable by other components. We used *priv* in place of *hidd* in the definition of composition. However, this approach has a weakness when more than two components must be composed. This weakness is the need for component-level (or finer) granularity in specifying what state information of each component is protected from other components. The granularity on *priv* was such that it could only distinguish between data private to the component and data potentially shared with other components. However, the collection of shared data elements can differ for each pair of components drawn from the set of components being composed. At a minimum we need the ability to specify data privacy on a component-pair basis. This allows us to automatically limit each component to its own data as we compose. We call this the *priv problem*. The solution adopted in the framework goes further by allowing privacy to be specified on a per-agent basis in the

$hidd$ field of each component specification. The granularity of $hidd$ allows a specification of exactly which agents have access to each data structure.

Our use of $hidd$ constitutes a generalization of Shankar's definition since we can take $hidd_i$ to be the set of all $(st_1, st_2, ag)$ triples such that $(st_1, st_2)$ is in $Shankar\_rely_i$.[6] There are two options to consider for the selection of a $Shankar\_rely$ relation:

**weak rely** — $Shankar\_rely$ requires that the component's private data does not change, but places no restrictions on changes to the interface data of the component. This definition leads to the situation described above where component $A$ can make arbitrary modifications to data it is unable to see. This can be dealt with in defining $guar$, but only at the expense of decreased maintainability of the specifications.

**strong rely** — $Shankar\_rely$ requires that the component's private data does not change and that any changes to the component's interface data follow the assumed protocol. In this case, the changes to the interface data are at least not arbitrary. However, if we are interested in restricting the components that can communicate with the component being defined, then we must do so in the $guar$ of the other components. This again leads to maintainability problems.

The basic weakness in both of these definitions of $hidd$ is that they are not sensitive to agents. From the standpoint of system functionality, this may not be a crucial concern, but from the standpoint of security it is very important to know not just what happened but also who caused it to happen.

In earlier versions of the framework, we explored an option like the Abadi-Lamport approach in which $hidd$ was defined separate from a component. We called the $hidd$ sets "respect relations". As discussed previously, we have now rejected this approach due to the issue with well-definedness of the composite's *guar* with respect to its *view*.

To ensure the composition of two components is consistent, it is necessary to check that each component satisfies the assumptions the other component makes about its environment. The expression $guar_a \cap rely_b$[7] denotes the set of transitions that agents of component $cmp_a$ can make that also satisfy the environmental assumptions of component $cmp_b$. Shankar's approach of using $guar_a \cap rely_b$ as the basis for the definition of composition ensures the result is consistent by eliminating transitions that violate environmental assumptions of the other component. Our approach retains the proof obligation to demonstrate that the components are consistent. This means proving $guar_a \cap hidd_b$ is a subset of $guar_b \cup rely_b$.

- While the Shankar approach and our previous frameworks have defined composition pairwise, we now allow an arbitrary number of sets to be composed at once. For example, we now use $compose(\{\, cmp_1, cmp_2, cmp_3 \,\})$ to denote the composition of components $cmp_1$, $cmp_2$, and $cmp_3$ where we previously used $compose(compose(cmp_1, cmp_2), cmp_3)$. As the number of components increases, the former approach is much nicer than the latter approach from a notational standpoint. Also, the more general definition of composition was found to greatly simplify the proof of a more general version of the composition theorem.

  Note that the discussion above only addresses the case of the composition of two components but can be generalized to the composition of an arbitrary number of components in a straightforward manner.

[6] Our $rely$ includes agents while Shankar's does not. We will henceforth use $rely$ and $Shankar\_rely$ to avoid confusion.
[7] Note that we use "$\cap$" and "$\wedge$" interchangeably. Similarly, we use "$\cup$" and "$\vee$" interchangeably.

## 9.2 Definition of Composition

We define the expression *compose*(*cset*) to denote the composition of each of the components in the set *cset*. Note that in prior versions of our framework:

- the parameters to *compose* contained a pair of components rather than a set of components, and
- the parameters to *compose* also contained state and agent translator functions that were used to address differences in data types used in the specification of the various components.

As noted earlier, we now have generalized the definition of *compose* to sets of components. In doing so, our use of PVS forced us to assume that the same data types were used in the definition of each component since each element of a PVS set must be of the same data type. We could have addressed this problem by using some data structure other than a set. For example, if we made *cset* be a tuple rather than a set, then each of the elements of the tuple could be of a different data type. However, then we would have to specify in advance how many elements are in the tuple. This is essentially the situation we were in previously where we could compose components of different data types but could only compose two at a time. Since we know of no way in PVS to specify an arbitrary sized collection of elements of different data types, we now require each of the elements to be of the same data type. Consequently, it is no longer necessary to provide state and agent translator functions as parameters to *compose* because no type translation is required.

However, we do still include the notion of state and agent translators in our framework (see Section 14). The intent is that the analyst would use state and agent translators as necessary to convert all of the components to the same data type. Then, the *compose* function could be used to compose all of the components together.

In defining *compose*, it is convenient to have generalized notions of union and intersection. For example, *cags* for the *compose*(*cset*) is defined as the union of *cags* for each element of *cset*. To formalize this in PVS, we define *gen_union*(*ss*) to be the union of each of the sets in the set of sets *ss*. For example, $gen\_union(\{\{a, b\}, \{c\}, \{b, d\}\})$ is $\{a, b, c, d\}$. Similarly, we define *gen_intersection* as a generalized version of intersection.

## THEORY *gen_set*

```
gen_set[X: TYPE]: THEORY
  BEGIN

  s, s1, s2: VAR setof[X]

  ss, ss1, ss2: VAR setof[setof[X]]

  x, x1: VAR X

  nonempty_th: THEOREM s /= emptyset IFF (EXISTS x: member(x, s))

  gen_union(ss): setof[X] =
    (LAMBDA x: (EXISTS s: member(s, ss) AND member(x, s)))

  gen_intersection(ss): setof[X] =
    (LAMBDA x: (FORALL s: member(s, ss) IMPLIES member(x, s)))
```

```
gen_union_zero: THEOREM gen_union(emptyset[setof[X]]) = emptyset

gen_intersection_zero: THEOREM gen_intersection(emptyset[setof[X]]) = fullset

gen_union_two: THEOREM gen_union({s | s = s1 OR s = s2}) = union(s1, s2)

gen_intersection_two: THEOREM
        gen_intersection({s | s = s1 OR s = s2}) = intersection(s1, s2)

gen_union_one: THEOREM gen_union(singleton(s)) = s

gen_intersection_one: THEOREM gen_intersection(singleton(s)) = s

gen_intersection_bigger: THEOREM
        subset?(ss1, ss2)
            IMPLIES subset?(gen_intersection(ss2), gen_intersection(ss1))

gen_union_smaller: THEOREM
        subset?(ss1, ss2) IMPLIES subset?(gen_union(ss1), gen_union(ss2))

contains_at_most_one(s): bool =
   (FORALL x, x1: member(x, s) AND member(x1, s) IMPLIES x = x1)

contains_one(s): bool = s /= emptyset AND contains_at_most_one(s)

contains_one_def: THEOREM contains_one(s) IFF (EXISTS x: s = singleton(x))

END gen_set
```

We restrict the domain of *compose* as follows:

- *cset* must be nonempty,
- each of the elements of *cset* must be a component as defined in Section 5,
- and there must be some state that is in *init* for each of the components in *cset*.

  Intuitively, this requires that there is some state that is an acceptable start state for every component in *cset*.

The function $composable(cset)$ tests whether *cset* satisfies these conditions. The function $compose(cset)$ is defined whenever $composable(cset)$ holds. Then, the result of the composition is defined to be a component for which:

- The set of allowable initial states for the composite is the intersection of the *init* sets for the individual components in *cset*
- The set of transitions that the composite can make consists of the transitions that belong to *guar* for at least one of the components and belong to either *guar* or *hidd* for the remaining components.

  The motivation for this definition is:

  – The composite should be able to perform only transitions that could be performed by at least one of the components. This means that each element of the composite's *guar* must be an element of *guar* for some component.

– The composite should not be able to perform transitions that violate the interface requirements of any of the components. This means that any transition of the composite that is not in *guar* for one of the components must be in *hidd* for that component (meaning that it respects the interface requirements of the component).

Suppose we let $G$ denote the union of the *guar*'s for each of the components in *cset* and let $GH$ denote the intersection of $guar \cup hidd$ for each of the components. Then, $G$ denotes the set of transitions that are in *guar* for at least one of the components while $GH$ denotes the set of transitions that are in either *guar* or *hidd* for each component. The transitions in *guar* for the composite are transitions belonging to both $G$ and $GH$. So, the definition of *guar* for the composite can be given as $G \cap GH$.

- The environment transitions allowed by the composite consist of transitions that each component allows of its environment. In other words, *rely* for the composite is the intersection of the *rely*'s for each component in *cset*.
- The agents for the composite consists of the union of the agents for the individual components in *cset*.
- Two states appear the same to the composite only if they appear the same to each component. In other words, *view* for the composite is the intersection of the *view*'s for each component in *cset*.
- Since *hidd* is similar to *rely*, *hidd* for the composite is defined to be the intersection of the *hidd*'s for each component in *cset*.
- The fairness assumptions are cumulative. In other words, if a component makes a weak fairness assumption for some transition class, then the composite must make a weak fairness assumption for that transition class, too. Consequently, the set of transition classes for which the composite assumes weak fairness is the union of the *wfar*'s for each of the components in *cset*.
- Similarly, *sfar* for the composite is the union of the *sfar*'s for each of the components in *cset*.

We prove that the result of the composition is itself a component in the sense defined in Section 5. Some of the requirements of a component are only provable when $composable(cset)$ holds. In fact, this is precisely why *composable* is defined as it is. It is the weakest definition for which $composable(cset)$ ensures $compose(cset)$ satisfies the requirements on components. Examples of requirements of components that depend on *composable* are as follows:

- $gen\_union(\emptyset)$ is equal to $\emptyset$. Consequently, if *cset* is empty, *cags* for the composite is empty and violates the requirement that a component have a nonempty *cags* set.
- If there was not some state that belonged to the *init* set for each element of *cset*, then the intersection of the *init*'s for components in *cset* would be empty. This would violate the requirement that a component have a nonempty *init* set.
- For most of the requirements on a component, it is necessary that the requirement hold for each element of *cset* in order for it to hold for the composite. For example, suppose *cags* were empty for each element of *cset*. Then, *cags* for the composite would be empty, too. Thus, each element of *cset* is required to satisfy the requirements on components.[8]

[8] Technically, this is not precisely necessary for the composite to be a component. For example, even if some of the elements of *cset* had an empty *cags* set, as long as one element has a nonempty *cags* set, the composite is

# THEORY *compose*

```
compose[ST: NONEMPTY_TYPE, AG: NONEMPTY_TYPE]: THEORY
  BEGIN

  IMPORTING gen_set

  IMPORTING component[ST, AG]

  cset: VAR setof[(comp_t)]

  cmp, cmp1, cmp2: VAR (comp_t)

  st, st1, st2, st3, st4: VAR ST

  ag: VAR AG

  agreeable_start(cset): bool =
    (EXISTS st: (FORALL cmp: member(cmp, cset) IMPLIES member(st, init(cmp))))

  composable(cset): bool = cset /= emptyset AND agreeable_start(cset)

  st_set: VAR setof[ST]

  inits_for(cset): setof[setof[ST]] =
    (LAMBDA st_set: (EXISTS cmp: member(cmp, cset) AND st_set = init(cmp)))

  compose_init(cset): setof[ST] = gen_intersection(inits_for(cset))

  tranc: VAR TRANSITION_CLASS

  guars_for(cset): setof[TRANSITION_CLASS] =
    (LAMBDA tranc: (EXISTS cmp: member(cmp, cset) AND tranc = guar(cmp)))

  guar_or_hidds_for(cset): setof[TRANSITION_CLASS] =
    (LAMBDA tranc:
       (EXISTS cmp: member(cmp, cset) AND tranc = union(guar(cmp), hidd(cmp))))

  relys_for(cset): setof[TRANSITION_CLASS] =
    (LAMBDA tranc: (EXISTS cmp: member(cmp, cset) AND tranc = rely(cmp)))

  hidds_for(cset): setof[TRANSITION_CLASS] =
    (LAMBDA tranc: (EXISTS cmp: member(cmp, cset) AND tranc = hidd(cmp)))

  v: VAR (VIEWS)

  views_for(cset): setof[(VIEWS)] =
    (LAMBDA v: (EXISTS cmp: member(cmp, cset) AND v = view(cmp)))

  ag_set: VAR setof[AG]

  cagss_for(cset): setof[setof[AG]] =
    (LAMBDA ag_set: (EXISTS cmp: member(cmp, cset) AND ag_set = cags(cmp)))

  tc_set: VAR setof[TRANSITION_CLASS]

  sfars_for(cset): setof[setof[TRANSITION_CLASS]] =
    (LAMBDA tc_set: (EXISTS cmp: member(cmp, cset) AND tc_set = sfar(cmp)))
```

guaranteed to have a nonempty *cags* set. Since stating the precise necessary conditions would be more difficult and it seems reasonable to require each element of *cset* satisfy the requirements on components, we have chosen to define *composable* to require each element of *cset* be a component.

```
wfars_for(cset): setof[setof[TRANSITION_CLASS]] =
  (LAMBDA tc_set: (EXISTS cmp: member(cmp, cset) AND tc_set = wfar(cmp)))

compose_guar(cset): setof[transition] =
  intersection(gen_intersection(guar_or_hidds_for(cset)),
               gen_union(guars_for(cset)))

compose_rely(cset): setof[transition] = gen_intersection(relys_for(cset))

compose_hidd(cset): setof[transition] = gen_intersection(hidds_for(cset))

compose_cags(cset): setof[AG] = gen_union(cagss_for(cset))

compose_view_base(cset): setof[[ST, ST]] =
  gen_intersection(extend[setof[[ST, ST]],
                          ((VIEWS)), bool,
                          FALSE](views_for(cset)))

compose_view_tc: THEOREM VIEWS(compose_view_base(cset))

compose_view(cset): (VIEWS[ST]) =
  gen_intersection(extend[setof[[ST, ST]],
                          ((VIEWS)), bool,
                          FALSE](views_for(cset)))

compose_wfar(cset): setof[TRANSITION_CLASS] = gen_union(wfars_for(cset))

compose_sfar(cset): setof[TRANSITION_CLASS] = gen_union(sfars_for(cset))

compose_base(cset): base_comp_t[ST, AG] =
  (# init := compose_init(cset),
     guar := compose_guar(cset),
     rely := compose_rely(cset),
     hidd := compose_hidd(cset),
     cags := compose_cags(cset),
     view := compose_view(cset),
     wfar := compose_wfar(cset),
     sfar := compose_sfar(cset) #)

compose_base_init: THEOREM
      cset /= emptyset AND agreeable_start(cset)
          IMPLIES init_restriction(compose_base(cset))

compose_base_guar: THEOREM guar_restriction(compose_base(cset))

compose_base_rely_hidd: THEOREM rely_hidd_restriction(compose_base(cset))

compose_base_hidd: THEOREM hidd_restriction(compose_base(cset))

compose_base_cags: THEOREM
      cset /= emptyset IMPLIES cags_restriction(compose_base(cset))

compose_base_view_rely: THEOREM view_rely_restriction(compose_base(cset))

compose_base_view_hidd: THEOREM view_hidd_restriction(compose_base(cset))

compose_base_view_guar: THEOREM view_guar_restriction(compose_base(cset))

compose_base_view_init: THEOREM view_init_restriction(compose_base(cset))

compose_base_view_sfar: THEOREM view_sfar_restriction(compose_base(cset))

compose_base_view_wfar: THEOREM view_wfar_restriction(compose_base(cset))
```

```
compose_base_guar_stuttering: THEOREM
        guar_stuttering_restriction(compose_base(cset))

compose_base_rely_stuttering: THEOREM
        rely_stuttering_restriction(compose_base(cset))

cmset: VAR (composable)

compose_base_tc: THEOREM comp_t(compose_base(cmset))

compose(cmset): (comp_t) =
  (# init := compose_init(cmset),
     guar := compose_guar(cmset),
     rely := compose_rely(cmset),
     hidd := compose_hidd(cmset),
     cags := compose_cags(cmset),
     view := compose_view(cmset),
     wfar := compose_wfar(cmset),
     sfar := compose_sfar(cmset) #)

END compose
```

In prior versions of the framework, we have proven that composition is:

- idempotent — this means composing $cmp$ with $cmp$ results in $cmp$
- commutative — this means composing $cmp_1$ with $cmp_2$ is the same as composing $cmp_2$ with $cmp_1$
- associative — this means $compose2(cmp_1, compose2(cmp_2, cmp_3))$ is the same as $compose2(compose2(cmp_1, cmp_2), cmp_3)$

  Here, $compose2$ denotes our old definition of a pairwise composition operator rather than the definition of $compose$ for sets given here.

It is common when defining binary operators to consider whether these properties hold. Since our previous definition of $compose$ was pairwise, it was a binary operator and we considered these properties. The analogue of idempotency for the definition of composition of sets of components is:

$composable(\{ cmp \})$ holds for any $cmp$, and

$compose(\{ cmp \}) = cmp$ holds for any $cmp$.

The commutativity requirement is not of interest because sets are unordered. For example, it would require showing that the composition of a set $\{ cmp_1, cmp_2, cmp_3 \}$ is the same as the composition of the set $\{ cmp_2, cmp_3, cmp_1 \}$. Since the order of elements of a set is irrelevant, the two sets are equivalent and the results of the composition must be the same.

Associativity deals with the way in which components are grouped into composites. For example, we would like to know that:

$compose(\{ cmp_1, compose(\{ cmp_2, cmp_3 \}) \}) =$

$$compose(\{\ compose(\{\ cmp_1, cmp_2\ \}), cmp_3\ \}) =$$

$$compose(\{\ cmp_1, cmp_2, cmp_3\ \})$$

In other words, we would like to know that the composition of a collection of components is the same regardless of whether and how the components are grouped into intermediate composite systems.

Our generalization of this requirement is as follows:

*Suppose $S_1$, ..., $S_n$ are sets of components and $S$ is the union of all of the $S_i$'s. Then,*

$$compose(\{\ compose(S_1), \ldots, compose(S_n)\ \}) = compose(S)$$

In other words, the requirement is that composing a collection of composites is equivalent to performing a single composition of all of the individual components. A corollary of this requirement is that the result of composing a collection of composites is independent of the way in which the components are grouped into composite systems.

The composition operator we defined previously satisfies our generalizations of idempotency, commutativity (trivially), and associativity to sets of components.

## THEORY *compose_idempotent*

```
compose_idempotent[ST: NONEMPTY_TYPE, AG: NONEMPTY_TYPE]: THEORY
  BEGIN

  IMPORTING compose[ST, AG]

  cmp: VAR (comp_t)

  ci_init: THEOREM compose_init(singleton(cmp)) = init(cmp)

  ci_cags: THEOREM compose_cags(singleton(cmp)) = cags(cmp)

  ci_guar: THEOREM compose_guar(singleton(cmp)) = guar(cmp)

  ci_rely: THEOREM compose_rely(singleton(cmp)) = rely(cmp)

  ci_hidd: THEOREM compose_hidd(singleton(cmp)) = hidd(cmp)

  ci_view: THEOREM compose_view(singleton(cmp)) = view(cmp)

  ci_sfar: THEOREM compose_sfar(singleton(cmp)) = sfar(cmp)

  ci_wfar: THEOREM compose_wfar(singleton(cmp)) = wfar(cmp)

  ci_composable: THEOREM composable(singleton(cmp))

  ci_component: THEOREM compose(singleton(cmp)) = cmp

  END compose_idempotent
```

## THEORY *compose_associative*

```
compose_associative[ST: NONEMPTY_TYPE, AG: NONEMPTY_TYPE]: THEORY
  BEGIN

  IMPORTING compose[ST, AG]

  cset: VAR (composable)

  csets: VAR setof[(composable)]

  cmp: VAR (comp_t)

  ca_composable: THEOREM
        composable(gen_union(extend[setof[((comp_t[ST, AG]))],
                                        ((composable)),
                                        bool,
                                        FALSE](csets)))
            IFF
          composable({cmp |
                        (EXISTS cset:
                            member(cset, csets) AND cmp = compose(cset))})

  ca_init: THEOREM
        composable(gen_union(extend[setof[((comp_t[ST, AG]))],
                                        ((composable)),
                                        bool,
                                        FALSE](csets)))
            IMPLIES
          init(compose(gen_union(extend[setof[((comp_t[ST, AG]))],
                                            ((composable)),
                                            bool,
                                            FALSE](csets))))
              =
            init(compose({cmp |
                            (EXISTS cset:
                                member(cset, csets)
                                    AND cmp = compose(cset))}))

  ca_cags: THEOREM
        composable(gen_union(extend[setof[((comp_t[ST, AG]))],
                                        ((composable)),
                                        bool,
                                        FALSE](csets)))
            IMPLIES
          cags(compose(gen_union(extend[setof[((comp_t[ST, AG]))],
                                            ((composable)),
                                            bool,
                                            FALSE](csets))))
              =
            cags(compose({cmp |
                            (EXISTS cset:
                                member(cset, csets)
                                    AND cmp = compose(cset))}))

  tran: VAR [ST, ST, AG]

  ca_guar1: THEOREM
        composable(gen_union(extend[setof[((comp_t[ST, AG]))],
                                        ((composable)),
                                        bool,
                                        FALSE](csets)))
```

```
     AND
   guar(compose(gen_union(extend[setof[((comp_t[ST, AG]))],
                                 ((composable)),
                                 bool,
                                 FALSE](csets))))
     (tran)
     IMPLIES
   gen_union(guars_for({cmp |
                  (EXISTS cset:
                     member(cset, csets)
                        AND cmp
                          = compose(cset))}))(tran)

ca_guar2: THEOREM
  composable(gen_union(extend[setof[((comp_t[ST, AG]))],
                              ((composable)),
                              bool,
                              FALSE](csets)))
     AND
   guar(compose(gen_union(extend[setof[((comp_t[ST, AG]))],
                                 ((composable)),
                                 bool,
                                 FALSE](csets))))
     (tran)
     IMPLIES
   gen_intersection(guar_or_hidds_for({cmp |
                                (EXISTS cset:
                                   member(cset, csets)
                                      AND cmp
                                        =
                                      compose(cset))}))
     (tran)

ca_guar3: THEOREM
  composable(gen_union(extend[setof[((comp_t[ST, AG]))],
                              ((composable)),
                              bool,
                              FALSE](csets)))
     AND
   guar(compose(gen_union(extend[setof[((comp_t[ST, AG]))],
                                 ((composable)),
                                 bool,
                                 FALSE](csets))))
     (tran)
     IMPLIES
   guar(compose({cmp |
              (EXISTS cset:
                 member(cset, csets)
                    AND cmp = compose(cset))}))(tran)

ca_guar4: THEOREM
  composable(gen_union(extend[setof[((comp_t[ST, AG]))],
                              ((composable)),
                              bool,
                              FALSE](csets)))
     AND
   guar(compose({cmp |
              (EXISTS cset:
                 member(cset, csets)
                    AND cmp = compose(cset))}))(tran)
     IMPLIES
   gen_union(guars_for(gen_union(extend[setof[((comp_t[ST, AG]))],
                                        ((composable)),
                                        bool,
```

```
                                                  FALSE](csets))))
         (tran)

  ca_guar5: THEOREM
       composable(gen_union(extend[setof[((comp_t[ST, AG]))],
                                       ((composable)),
                                       bool,
                                       FALSE](csets)))
         AND
        guar(compose({cmp |
                     (EXISTS cset:
                        member(cset, csets)
                           AND cmp = compose(cset))}))(tran)
         IMPLIES
        gen_intersection
         (guar_or_hidds_for(gen_union(extend[setof[((comp_t[ST,
                                                          AG]))],
                                              ((composable)),
                                              bool,
                                              FALSE](csets))))
         (tran)

  ca_guar6: THEOREM
       composable(gen_union(extend[setof[((comp_t[ST, AG]))],
                                       ((composable)),
                                       bool,
                                       FALSE](csets)))
         AND
        guar(compose({cmp |
                     (EXISTS cset:
                        member(cset, csets)
                           AND cmp = compose(cset))}))(tran)
         IMPLIES
        guar(compose(gen_union(extend[setof[((comp_t[ST, AG]))],
                                          ((composable)),
                                          bool,
                                          FALSE](csets))))
         (tran)

  ca_guar: THEOREM
       composable(gen_union(extend[setof[((comp_t[ST, AG]))],
                                       ((composable)),
                                       bool,
                                       FALSE](csets)))
         IMPLIES
        guar(compose(gen_union(extend[setof[((comp_t[ST, AG]))],
                                          ((composable)),
                                          bool,
                                          FALSE](csets))))
           =
         guar(compose({cmp |
                      (EXISTS cset:
                         member(cset, csets)
                            AND cmp = compose(cset))}))

  ca_rely: THEOREM
       composable(gen_union(extend[setof[((comp_t[ST, AG]))],
                                       ((composable)),
                                       bool,
                                       FALSE](csets)))
         IMPLIES
        rely(compose(gen_union(extend[setof[((comp_t[ST, AG]))],
                                          ((composable)),
                                          bool,
```

```
                                                FALSE](csets))))
        =
        rely(compose({cmp |
                        (EXISTS cset:
                            member(cset, csets)
                                AND cmp = compose(cset))}))

ca_hidd: THEOREM
    composable(gen_union(extend[setof[((comp_t[ST, AG]))],
                                            ((composable)),
                                            bool,
                                            FALSE](csets)))
        IMPLIES
      hidd(compose(gen_union(extend[setof[((comp_t[ST, AG]))],
                                                ((composable)),
                                                bool,
                                                FALSE](csets))))
        =
        hidd(compose({cmp |
                        (EXISTS cset:
                            member(cset, csets)
                                AND cmp = compose(cset))}))

ca_view: THEOREM
    composable(gen_union(extend[setof[((comp_t[ST, AG]))],
                                            ((composable)),
                                            bool,
                                            FALSE](csets)))
        IMPLIES
      view(compose(gen_union(extend[setof[((comp_t[ST, AG]))],
                                                ((composable)),
                                                bool,
                                                FALSE](csets))))
        =
        view(compose({cmp |
                        (EXISTS cset:
                            member(cset, csets)
                                AND cmp = compose(cset))}))

ca_sfar: THEOREM
    composable(gen_union(extend[setof[((comp_t[ST, AG]))],
                                            ((composable)),
                                            bool,
                                            FALSE](csets)))
        IMPLIES
      sfar(compose(gen_union(extend[setof[((comp_t[ST, AG]))],
                                                ((composable)),
                                                bool,
                                                FALSE](csets))))
        =
        sfar(compose({cmp |
                        (EXISTS cset:
                            member(cset, csets)
                                AND cmp = compose(cset))}))

ca_wfar: THEOREM
    composable(gen_union(extend[setof[((comp_t[ST, AG]))],
                                            ((composable)),
                                            bool,
                                            FALSE](csets)))
        IMPLIES
      wfar(compose(gen_union(extend[setof[((comp_t[ST, AG]))],
                                                ((composable)),
                                                bool,
```

```
                                            FALSE](csets))))
          =
          wfar(compose({cmp |
                        (EXISTS cset:
                            member(cset, csets)
                                AND cmp = compose(cset))}))

ca_component: THEOREM
      composable(gen_union(extend[setof[((comp_t[ST, AG]))],
                                       ((composable)),
                                       bool,
                                       FALSE](csets)))
          IMPLIES
        compose(gen_union(extend[setof[((comp_t[ST, AG]))],
                                      ((composable)),
                                      bool,
                                      FALSE](csets)))
          =
          compose({cmp |
                   (EXISTS cset:
                       member(cset, csets) AND cmp = compose(cset))})

END compose_associative
```

*Section* *10*
# Composition Theorem

The composition theorem is:

> *Suppose a collection of components $S_1$ satisfies a property $P$, and a "bigger" set of components $S_2$ is such that its actions are "tolerable" with respect to $S_1$'s environmental assumptions. Then, $P$ holds for $S_2$, also.*

A typical use of this theorem would be to choose $S_1$ to be a set of previously analyzed components and $S_2$ to be $S_1$ with some additional components added. Then, the theorem can be used to demonstrate that properties proved of $S_1$ hold for $S_2$, too, as long as the environmental assumptions tolerate the new components.

## THEORY *cmp_thm*

```
cmp_thm[ST: NONEMPTY_TYPE, AG: NONEMPTY_TYPE]: THEORY
  BEGIN

  IMPORTING cmp_thm_aux[ST, AG]

  cset1, cset2: VAR setof[(comp_t)]

  p: VAR prop_t

  cmp_thm_base: THEOREM
        contains(cset1, cset2)
            AND tolerates(cset1, cset2)
              AND composable(cset1) AND satisfies(compose(cset1), p)
            IMPLIES (composable(cset2) IMPLIES satisfies(compose(cset2), p))

  cmp_thm_base_disj: THEOREM
        contains(cset1, cset2)
            AND tolerates_disj(cset1, cset2)
              AND composable(cset1) AND satisfies(compose(cset1), p)
            IMPLIES (composable(cset2) IMPLIES satisfies(compose(cset2), p))

  cmp_thm: THEOREM
        subset?(cset1, cset2)
            AND tolerates(cset1, cset2)
              AND composable(cset2)
                AND cset1 /= emptyset AND satisfies(compose(cset1), p)
            IMPLIES satisfies(compose(cset2), p)

  cmp_thm_disj: THEOREM
        subset?(cset1, cset2)
            AND tolerates_disj(cset1, cset2)
              AND cset1 /= emptyset
                AND composable(cset2) AND satisfies(compose(cset1), p)
            IMPLIES satisfies(compose(cset2), p)

  END cmp_thm
```

In the above, it is implicitly assumed that both $S_1$ and $S_2$ are composable. By a collection of components satisfying a property, we simply mean that the property holds for the composite of the collection of components. In the simplest case, "bigger" simply refers to a subset relation between the sets. However, we are actually able to prove a more general version of the theorem in which $S_2$ being bigger than $S_1$ means that for each element, $cmp_1$ of $S_1$, there exists an element $cmp_2$ of $S_2$ such that:

$$cmp\_contains(cmp_2, cmp_1)$$

where $cmp\_contains$ is as defined at the end of Section 5. Since $cmp\_contains(cmp, cmp)$ holds for any component, $cmp$, whenever $S_1$ is a subset of $S_2$, the above notion of "bigger" is satisfied by choosing $cmp_2$ to be $cmp_1$ for each $cmp_1$ in $S_1$.

## THEORY *contains*

```
contains[ST: NONEMPTY_TYPE, AG: NONEMPTY_TYPE]: THEORY
  BEGIN

  IMPORTING cmp_contains[ST, AG]

  cmp, cmp1, cmp2: VAR (comp_t)

  cset1, cset2: VAR setof[(comp_t)]

  contains(cset1, cset2): bool =
    (FORALL cmp1:
       member(cmp1, cset1)
           IMPLIES
         (EXISTS cmp2: member(cmp2, cset2) AND cmp_contains(cmp2, cmp1)))

  END contains
```

The last detail of the composition theorem that needs to be addressed is the notion of the actions of a collection of components being tolerable with respect to the environmental assumptions of another collection of components.

First, we define $tolerates\_cmp(S_1, cmp_2)$ to hold exactly when for each transition in $cmp_2$'s *guar* either:

- there exists a component in $S_1$ whose *guar* contains the transition, or
- for each component in $S_1$, either the transition violates *hidd* for the component or the transition is an element of *rely* for the component.

In the first case, a step of $cmp_2$ is acceptable to $S_1$ because some component in $S_1$ could perform the step itself. In the second case, a step of $cmp_2$ is acceptable to $S_1$ because it does not violate any of the environmental assumptions of elements of $S_1$ (in the sense that whenever the step is consistent with the interface specified by *hidd*, then the step is consistent with the assumption captured by *rely*).

Given the definition of $tolerates\_cmp(S_1, cmp_2)$ we define $tolerates(S_1, S_2)$ to hold whenever $tolerates\_cmp(S_1, cmp_2)$ holds for each element $cmp_2$ of $S_2$. In many cases, a stronger relation can be shown to hold between two sets of components. In particular, it is not uncommon for the agent sets of the components in $S_1$ to be non-overlapping with the agent sets of the components in $S_2$. Then, it is not possible for an element of the $guar$ for a component in $S_2$ to also be an element of the $guar$ for a component in $S_1$. Consequently, the demonstration that $S_1$ tolerates $S_2$ requires showing the second case in the definition of $tolerates\_cmp$. We define $tolerates\_disj(S_1, S_2)$ to hold when this stronger notion of tolerance holds. Two even stronger notions are also defined:

- $tolerates\_stutter(S_1, S_2)$ — the $guar$ transitions of components in $S_2$, when restricted by the $hidd$ of components in $S_1$ are stuttering steps with respect to the $view$ of $S_1$ components.
- $tolerates\_cags(S_1, S_2)$ — the $hidd$ relations of components in $S_1$ allow only stuttering steps for agents of components in $S_2$.

The latter is particularly useful when applicable since no $guar$ transitions need be considered. This concept and the associated theorem $tolerates\_cags\_stronger$ play a central role in the example worked later in this report.

## THEORY *tolerates*

```
tolerates[ST: NONEMPTY_TYPE, AG: NONEMPTY_TYPE]: THEORY
  BEGIN

  IMPORTING component[ST, AG]

  cset, cset1, cset2, cset3: VAR setof[(comp_t)]

  cmp, cmp1, cmp2: VAR (comp_t)

  st, st1, st2: VAR ST

  ag: VAR AG

  ags: VAR setof[AG]

  tran: VAR transition

  tolerates_cmp(cset1, cmp2): bool =
    (FORALL tran:
        member(tran, guar(cmp2))
            IMPLIES
          ((EXISTS cmp1: member(cmp1, cset1) AND member(tran, guar(cmp1)))
              OR
            (FORALL cmp1:
                member(cmp1, cset1) AND member(tran, hidd(cmp1))
                    IMPLIES member(tran, rely(cmp1)))))

  tolerates_cmp_disj(cset1, cmp2): bool =
    (FORALL tran:
        member(tran, guar(cmp2))
            IMPLIES
          ((FORALL cmp1:
                member(cmp1, cset1) AND member(tran, hidd(cmp1))
                    IMPLIES member(tran, rely(cmp1)))))
```

```
tolerates_cmp_stutter(cset1, cmp2): bool =
  (FORALL st1, st2, ag, cmp1:
        member(cmp1, cset1) AND member((st1, st2, ag), guar(cmp2))
                 AND member((st1, st2, ag), hidd(cmp1))
        IMPLIES member((st1, st2), view(cmp1)))

tolerates_cmp_cags(cset1, cmp2): bool =
  (FORALL st1, st2, ag, cmp1:
        member(cmp1, cset1) AND member(ag, cags(cmp2))
                 AND member((st1, st2, ag), hidd(cmp1))
        IMPLIES member((st1, st2), view(cmp1)))

tolerates_cmp_disj_stronger: THEOREM
        tolerates_cmp_disj(cset1, cmp2) IMPLIES tolerates_cmp(cset1, cmp2)

tolerates_cmp_stutter_stronger: THEOREM
        tolerates_cmp_stutter(cset1, cmp2) IMPLIES tolerates_cmp_disj(cset1, cmp2)

tolerates_cmp_cags_stronger: THEOREM
        tolerates_cmp_cags(cset1, cmp2) IMPLIES tolerates_cmp_stutter(cset1, cmp2)

tolerates_cmp_cags_stronger2: THEOREM
        tolerates_cmp_cags(cset1, cmp2) IMPLIES tolerates_cmp_disj(cset1, cmp2)

tolerates(cset1, cset2): bool =
  (FORALL cmp2: member(cmp2, cset2) IMPLIES tolerates_cmp(cset1, cmp2))

tolerates_prop: THEOREM
        tolerates(cset1, cset2) AND subset?(cset, cset2)
            IMPLIES tolerates(cset1, cset)

tolerates_union: THEOREM
  tolerates(cset1, cset2)
        AND tolerates(cset1, cset3)
        AND cset = union(cset2, cset3)
  => tolerates(cset1, cset)

tolerates_disj(cset1, cset2): bool =
  (FORALL cmp2: member(cmp2, cset2) IMPLIES tolerates_cmp_disj(cset1, cmp2))

tolerates_stutter(cset1, cset2): bool =
  (FORALL cmp2: member(cmp2, cset2) IMPLIES tolerates_cmp_stutter(cset1, cmp2))

tolerates_cags(cset1, cset2): bool =
  (FORALL cmp2: member(cmp2, cset2) IMPLIES tolerates_cmp_cags(cset1, cmp2))

tolerates_cags_help: THEOREM
  (FORALL cmp1, cmp2, st1, st2, ag :
        (cset1(cmp1) AND hidd(cmp1)(st1, st2, ag)
            => ags(ag) OR view(cmp1)(st1, st2))
     AND (cset2(cmp2) AND cags(cmp2)(ag) => NOT ags(ag)))
  IMPLIES
     tolerates_cags(cset1, cset2)

tolerates_disj_stronger: THEOREM
        tolerates_disj(cset1, cset2) IMPLIES tolerates(cset1, cset2)

tolerates_stutter_stronger: THEOREM
        tolerates_stutter(cset1, cset2) IMPLIES tolerates(cset1, cset2)

tolerates_cags_stronger: THEOREM
        tolerates_cags(cset1, cset2) IMPLIES tolerates(cset1, cset2)

tolerates_disj_prop2: THEOREM
```

```
tolerates_disj(cset1, cset2) AND subset?(cset, cset2)
    IMPLIES tolerates_disj(cset1, cset)

END tolerates
```

The key to the proof of the composition theorem is to show:

$$cmp\_contains(compose(S_2), compose(S_1))$$

Then, theorem $satisfies\_contains\_prop$ from Section 7 ensures that any property satisfied by $compose(S_1)$ is also satisfied by $compose(S_2)$. Demonstrating the $cmp\_contains$ relation holds requires showing that:[9]

- *init*, *rely*, *hidd*, and *view* for $S_2$ are smaller than their counterparts for $S_1$.

  Since these fields of a component are intersected when components are composed, this requirement holds whenever $S_2$ is bigger than $S_1$. Intersection can only make sets smaller, so the more sets that are intersected together, the smaller the result.

- *cags*, *wfar*, and *sfar* for $S_1$ are smaller than their counterparts for $S_2$.

  These fields of a component are unioned when components are composed. Thus, this requirement, too, holds whenever $S_2$ is bigger than $S_1$. Union can only make sets bigger, so the more sets that are unioned together, the bigger the result.

- *guar* for $S_2$ is smaller than *steps* for $S_1$.

  This is the hard part of the proof. We assume that $tran$ is an element of $S_2$'s *guar* and give a chain of reasoning that demonstrates $tran$ is an element of either *guar* or *rely* for $S_1$ (which means $tran$ is an element of *steps* for $S_1$).

  - For each $cmp_1$ in $S_1$, $tran$ is an element of either *guar* or *hidd* for $cmp_1$.

    Let $cmp_1$ be an arbitrary element of $S_1$. By definition, $contains(S_1, S_2)$ requires that there exists a $cmp_2$ in $S_2$ such that $cmp\_contains(cmp_2, cmp_1)$. Then, the definition of $cmp\_contains$ implies:

    * $guar(cmp_2) \subseteq steps(cmp_1) = guar(cmp_1) \cup rely(cmp_1)$, and
    * $hidd(cmp_2) \subseteq hidd(cmp_1)$

    Consequently, $guar(cmp_2) \cup hidd(cmp_2) \subseteq guar(cmp_1) \cup rely(cmp_1) \cup hidd(cmp_1)$. Since a component's *hidd* always contains its *rely*, the union of terms for $cmp_1$ reduces to simply $guar(cmp_1) \cup hidd(cmp_1)$. In summary, for each $cmp_1$, there exists a $cmp_2$ such that $guar(cmp_2) \cup hidd(cmp_2) \subseteq guar(cmp_1) \cup hidd(cmp_1)$.

    To complete the proof of this step, it suffices to show that $tran$ is an element of $guar(cmp_2) \cup hidd(cmp_2)$. This follows immediately from the definition of the *guar* field of the composition.

  - If $tran$ is an element of *guar* for some component in $S_1$, then $tran$ is an element of *guar* for $S_1$. This completes the proof for this case since any element of *guar* is also an element of *steps*.

[9] The proof given here follows a hierarchical structure. The top-level bullets provide a sketch of the proof of the containment relationship. Text under a bullet provides a proof of the assertion in the bullet. Lower level bullets indicate more detailed steps of the proof.

This step follows immediately from the definition of *guar* for the composite; $tran$ is assumed to be an element of *guar* for a component in $S_1$ and the previous step of the proof showed that $tran$ is an element of either *guar* or *hidd* for each component in $S_1$.

- Otherwise, $tran$ is not an element of the *guar* for any component in $S_1$. Then $tran$ is an element of *hidd* for every component in $S_1$.

  The first step of the proof showed that $tran$ is an element of either *guar* or *hidd* of each component in $S_1$. So, if $tran$ is not an element of *guar* for any component in $S_1$, it must be an element of *hidd* for every component in $S_1$.

- $tran$ is an element of *rely* for $S_1$. This completes the proof since any element of *rely* is an element of *steps*.

  From the previous steps of the proof, we can assume that $tran$ is not an element of *guar* for any component in $S_1$ but is an element of *hidd* for every component in $S_1$. Then, the definition of $tolerates$ implies the desired result.

## THEORY *cmp_thm_aux*

```
cmp_thm_aux[ST: NONEMPTY_TYPE, AG: NONEMPTY_TYPE]: THEORY
  BEGIN

  IMPORTING compose[ST, AG]

  IMPORTING cprops[ST, AG]

  IMPORTING contains[ST, AG]

  IMPORTING tolerates[ST, AG]

  cset, cset1, cset2: VAR setof[(comp_t)]

  cmp1, cmp2: VAR (comp_t)

  tran: VAR [ST, ST, AG]

  st, st1, st2: VAR ST

  ag: VAR AG

  key_composable: THEOREM
        subset?(cset1, cset2) AND cset1 /= emptyset AND composable(cset2)
            IMPLIES composable(cset1)

  key_init: THEOREM
        contains(cset1, cset2)
            AND composable(cset2) AND member(st, init(compose(cset2)))
            IMPLIES
          (composable(cset1) IMPLIES member(st, init(compose(cset1))))

  key_guar1: THEOREM
        contains(cset1, cset2)
            AND composable(cset2) AND member(tran, guar(compose(cset2)))
            IMPLIES member(tran, gen_intersection(guar_or_hidds_for(cset1)))

  key_guar2: THEOREM
        composable(cset2)
            AND member(tran, guar(compose(cset2))) AND tolerates(cset1, cset2)
            IMPLIES
```

```
    (member(tran, gen_union(guars_for(cset1)))
         OR
       (FORALL cmp1:
          member(cmp1, cset1) AND member(tran, hidd(cmp1))
              IMPLIES member(tran, rely(cmp1))))

key_guar3: THEOREM
  NOT member(tran, gen_union(guars_for(cset1)))
      AND member(tran, gen_intersection(guar_or_hidds_for(cset1)))
      IMPLIES member(tran, gen_intersection(hidds_for(cset1)))

key_guar4: THEOREM
  member(tran, gen_intersection(hidds_for(cset1)))
      AND
    (FORALL cmp1:
       member(cmp1, cset1) AND member(tran, hidd(cmp1))
           IMPLIES member(tran, rely(cmp1)))
      IMPLIES member(tran, gen_intersection(relys_for(cset1)))

key_guar: THEOREM
  contains(cset1, cset2)
      AND tolerates(cset1, cset2)
        AND composable(cset2) AND member(tran, guar(compose(cset2)))
      IMPLIES
    (composable(cset1) IMPLIES member(tran, steps(compose(cset1))))

key_rely: THEOREM
  contains(cset1, cset2)
      AND composable(cset2) AND member(tran, rely(compose(cset2)))
      IMPLIES
    (composable(cset1) IMPLIES member(tran, rely(compose(cset1))))

key_hidd: THEOREM
  contains(cset1, cset2)
      AND composable(cset2) AND member(tran, hidd(compose(cset2)))
      IMPLIES
    (composable(cset1) IMPLIES member(tran, hidd(compose(cset1))))

key_view: THEOREM
  contains(cset1, cset2)
      AND composable(cset2) AND member((st1, st2), view(compose(cset2)))
      IMPLIES
    (composable(cset1) IMPLIES member((st1, st2), view(compose(cset1))))

tranc: VAR TRANSITION_CLASS

key_wfar: THEOREM
  contains(cset1, cset2)
      AND composable(cset1) AND member(tranc, wfar(compose(cset1)))
      IMPLIES
    (composable(cset2) IMPLIES member(tranc, wfar(compose(cset2))))

key_sfar: THEOREM
  contains(cset1, cset2)
      AND composable(cset1) AND member(tranc, sfar(compose(cset1)))
      IMPLIES
    (composable(cset2) IMPLIES member(tranc, sfar(compose(cset2))))

key_cags: THEOREM
  contains(cset1, cset2)
      AND composable(cset1) AND member(ag, cags(compose(cset1)))
      IMPLIES
    (composable(cset2) IMPLIES member(ag, cags(compose(cset2))))
```

```
key: THEOREM
      contains(cset1, cset2)
          AND tolerates(cset1, cset2)
            AND composable(cset2)
              AND cmp2 = compose(cset2)
                AND composable(cset1) AND cmp1 = compose(cset1)
          IMPLIES cmp_contains(cmp2, cmp1)

END cmp_thm_aux
```

# Section 11 Distinction between *hidd* and *rely*

In this section we provide an example illustrating the importance of including both *hidd* and *rely* rather than simply *rely* in the framework. For our example, we suppose that we have two concurrently executing processes that both increment a shared variable named *value* to indicate when they have completed some task. For purposes of the example we are interested in only the correct setting of *value* and do not model the processing associated with the task performed by each process. We assume that the processes cannot atomically increment *value*. For example, while *value* might be a remote variable that each process can atomically read or write, incrementing *value* would require atomically reading, adding 1, and writing the new value. Doing so correctly in the face of concurrency requires some type of mutual exclusion protocol to be used. For purposes of the example, we assume a simple locking protocol. The variable *locked?* is checked by each process before accessing *value*. If *locked?* is not set, the process sets *locked?* and reads the contents of *value* into a local variable. Next, the process adds one to its local variable, writes the result to *value*, and clears *locked?*. After this point, no further processing is performed by the process.

## 11.1 State

We name the processes *ONE* and *TWO*. In addition to *value* and *locked?*, the system state for our example also contains the following variables:

- *locker* — set to *ONE* or *TWO* depending on which process last set *locked?*; the value of this variable is only meaningful when *locked?* is set
- *v* — an array indexed by processes that denotes each process's local variable; for example, *v(ONE)* denotes the local variable for process *ONE*
- *pc* — an array of "program counters" indexed by processes that denotes where each process is in its processing; for example, *pc(ONE)* indicates where process *ONE* is in its processing

  Possible values for the program counter are:

  - *READ* — the initial value indicating that the process has yet to read *value*
  - *WRITE* — the second value indicating that the process has read but not yet written *value*
  - *DONE* — the third and final value indicating the process is finished

**THEORY** *state*

```
state: THEORY
  BEGIN

  AGENT: TYPE = {ONE, TWO}
```

```
STEP: TYPE = {READ, WRITE, DONE}

STATE:
    TYPE =
      [# locked?: bool,
         locker: AGENT,
         value: nat,
         v: [AGENT -> nat],
         pc: [AGENT -> STEP] #]

END state
```

## 11.2 Component Specification

The set of initial states for a process *p* are those states in which:

- *locked?* is cleared,
- *value* is set to 0, and
- *pc(p)* is *READ*

The set of agents for each component is the singleton set containing only the process.

The *locked?*, *locker*, and *value* variables are included in the view for each process. Only a process's own elements of *v* and *pc* are included in that process's view. For example, the view of process *ONE* contains *v(ONE)* and *pc(ONE)* as well as *locked?*, *locker*, and *value*.

The assumptions made by each process, captured by the *rely* for each process, are:

1. The other process does not modify the first process's elements of *v* and *pc*.
2. If the first process has *locked?* set, then the second process cannot change *locked?*, *locker*, or *value*.

We consider two definitions for the definition of *hidd* for a process. The first definition for each is the "ideal" definition in that:

- *hidd* captures the requirement that each process cannot modify the other process's elements of *v* and *pc*. This is assumption 1 of the definition of *rely*.

  If this requirement is not included in *hidd*, then ensuring a process's private variables are private can only be accomplished by constraining the second process's *guar* to not modify those variables. In this particular example, that is not possible without expanding the view of each process to be the entire state. For example, the *guar* for process *TWO* cannot specify that *v(ONE)* is not modified unless process *TWO*'s view is expanded to include *v(ONE)*. From a maintainability standpoint, including private variables of one component in the view for a second component is undesirable. First, as new components are specified for the system, the specifications of other components must be updated to recognize those variables as private. Second, replacing one component with a second component that is equivalent from the standpoint of its interface could still require changes in the other components since the two versions of the component could include different private variables.

- Aside from indicating which variables can and cannot be altered by the environment, *hidd* places no restrictions on how variables are changed.

  Using such a minimal definition of *hidd* results in a proof obligation being generated for showing that composition is "correct". This proof obligation requires showing that each component's *guar* when restricted by *hidd* of the second component contains only transitions that are included in *rely* of the second component. If a more restrictive definition is used for *hidd*, the proof obligation for "correct" composition can become trivial at the expense of not being forced to consider potential inconsistencies between the components. For example, if *hidd* is chosen to be the same as *rely*, then the proof obligation is trivially true regardless of how *guar* is defined. In summary, if *hidd* is defined too strongly, errors in the *guar* for one component that cause it to be inconsistent with *rely* for another component cannot be brought to light by the proof obligations.

## THEORY *common*

```
common: THEORY
  BEGIN

  IMPORTING state

  st, st1, st2: VAR STATE

  ag, ag1: VAR AGENT

  pinit(ag): setof[STATE] =
    {st | locked?(st) = FALSE AND value(st) = 0 AND pc(st)(ag) = READ}

  pcags(ag): setof[AGENT] = {ag1 | ag = ag1}

  pview_base(ag): setof[[STATE, STATE]] =
    {st1, st2 |
    locked?(st1) = locked?(st2)
          AND locker(st1) = locker(st2)
            AND value(st1) = value(st2)
              AND v(st1)(ag) = v(st2)(ag) AND pc(st1)(ag) = pc(st2)(ag)}

  pview_base_ref: THEOREM reflexive?(pview_base(ag))

  pview_base_sym: THEOREM symmetric?(pview_base(ag))

  pview_base_tran: THEOREM transitive?(pview_base(ag))

  pview_base_equiv: THEOREM equivalence?(pview_base(ag))

  pview(ag): (equivalence?[STATE]) =
    {st1, st2 |
    locked?(st1) = locked?(st2)
          AND locker(st1) = locker(st2)
            AND value(st1) = value(st2)
              AND v(st1)(ag) = v(st2)(ag) AND pc(st1)(ag) = pc(st2)(ag)}

  phidd(ag): setof[[STATE, STATE, AGENT]] =
    {st1, st2, ag1 |
    ag1 /= ag AND v(st1)(ag) = v(st2)(ag) AND pc(st1)(ag) = pc(st2)(ag)}

  prely(ag): setof[[STATE, STATE, AGENT]] =
    {st1, st2, ag1 |
```

```
  ag1 /= ag
      AND phidd(ag)(st1, st2, ag1)
        AND
      (locked?(st1) AND locker(st1) = ag
           IMPLIES locked?(st2) = locked?(st1)
             AND locker(st2) = locker(st1) AND value(st2) = value(st1))}

ST_WITNESS: STATE =
  (# locked? := FALSE,
     locker := ONE,
     value := 0,
     v := (LAMBDA ag: 0),
     pc := (LAMBDA ag: READ) #)

pinit_thm: THEOREM pinit(ag) /= emptyset

END common
```

Our first definition for *hidd* is simply that a process assumes that its private variables are not modified by the other process. Our second definition for *hidd* is as being equivalent to *rely* defined above. Below, we will show that the second definition of *hidd* allows correct operation of the composite to be shown even if a crucial precondition of *guar* is omitted.

The correct definition of *guar* for a process *p* should be that it allows the following transitions:

- stuttering transitions that do not change the view for the process,
- transitions in which either:
  - initially *pc(p)* = *READ* and *locked?* is clear, and
  - in the new state *locked?* is set, *locker* = *p*, *value* is unchanged, *v(p)* = *value*, and *pc(p)* = *WRITE*

  or

  - initially *pc(p)* = *WRITE*, *locked?* is set, and *locker* = *p* and
  - in the new state *locked?* is clear, *locker* = *p*, *value* = *v(p)+1*, and *pc(p)* = *DONE*

Note that once *pc(p)* = *DONE* only stuttering transitions are permitted.

## THEORY *ex*

```
ex: THEORY
  BEGIN

  IMPORTING state

  IMPORTING common

  comp: LIBRARY = "/home/cmt/rev/dtos/docs/compose/"

  IMPORTING comp@cmp_thm[STATE, AGENT]

  IMPORTING comp@compose_idempotent[STATE, AGENT]
```

```
IMPORTING comp@preds[STATE, AGENT]

ag, ag1: VAR AGENT

st, st1, st2: VAR STATE

pguar(ag): setof[[STATE, STATE, AGENT]] =
  {st1, st2, ag1 |
   ag1 = ag
     AND
    (pview(ag)(st1, st2)
         OR
       (pc(st1)(ag) = READ
           AND NOT locked?(st1)
             AND locked?(st2)
               AND locker(st2) = ag
                 AND value(st2) = value(st1)
                   AND v(st2)(ag) = value(st1) AND pc(st2)(ag) = WRITE)
          OR
         (pc(st1)(ag) = WRITE
             AND locked?(st1)
               AND locker(st1) = ag
                 AND NOT locked?(st2)
                   AND locker(st2) = ag
                     AND value(st2) = v(st1)(ag) + 1
                       AND pc(st2)(ag) = DONE))}

mk_cmp(ag): (comp_t) =
  (# init := pinit(ag),
     cags := pcags(ag),
     view := pview(ag),
     hidd := phidd(ag),
     rely := prely(ag),
     guar := pguar(ag),
     wfar := emptyset,
     sfar := emptyset #)

END ex
```

---

Our second definition of *guar* for a process *p* differs from that above only in that *locked?* is not required to be clear for a transition to be made from *pc(p) = READ* to *pc(p) = WRITE*. Intuitively, this should result in the composite operating incorrectly since even if one process currently has *locked?* set, the other process can set *locked?* and concurrently access *value*. However, as we will discuss next, the second definition of *hidd* is strong enough to allow the composite to be proved correct even with the error in the second definition of *guar*. This is our illustration that errors in the component of the *guar* of a component can be masked if *hidd* for another component is specified too strongly.

## THEORY *ex1*

---

```
ex: THEORY
  BEGIN

  IMPORTING state

  IMPORTING common
```

```
comp: LIBRARY = "/home/cmt/rev/dtos/docs/compose/"

IMPORTING comp@cmp_thm[STATE, AGENT]

IMPORTING comp@compose_idempotent[STATE, AGENT]

IMPORTING comp@preds[STATE, AGENT]

ag, ag1: VAR AGENT

st, st1, st2: VAR STATE

pguar(ag): setof[[STATE, STATE, AGENT]] =
  {st1, st2, ag1 |
  ag1 = ag
       AND
    (pview(ag)(st1, st2)
         OR
       (pc(st1)(ag) = READ
            AND locked?(st2)
              AND locker(st2) = ag
                AND value(st2) = value(st1)
                  AND v(st2)(ag) = value(st1) AND pc(st2)(ag) = WRITE)
           OR
         (pc(st1)(ag) = WRITE
              AND locked?(st1)
                AND locker(st1) = ag
                  AND NOT locked?(st2)
                    AND locker(st2) = ag
                      AND value(st2) = v(st1)(ag) + 1
                        AND pc(st2)(ag) = DONE))}

mk_cmp(ag): (comp_t) =
  (# init := pinit(ag),
     cags := pcags(ag),
     view := pview(ag),
     hidd := prely(ag),
     rely := prely(ag),
     guar := pguar(ag),
     wfar := emptyset,
     sfar := emptyset #)

END ex
```

## 11.3 Correctness

Our goal is to show that the composite is such that:

> *In any state in which pc(ONE) and pc(TWO) are both DONE, value is 2.*

Since each process individually executes code that increments *value* by 1 and *value* is initially 0, this requires that each process is guaranteed exclusive access to *value* while it executes.

We consider two compositions; the first has the components defined using the first definitions for *hidd* and *guar* while the second composition uses the second set of definitions. The first composition can be seen to satisfy the above property by noting that each process individually increments *value* and the use of *locked?* prevents the scenario in which both processes concurrently read a value of 0 and then set the new value to 1. Since both the read transition and the write transition can update one of *locked?*, *locker*, or *value*, the definition of *rely* causes a proof obligation to be generated that neither transition can occur when *locked?* is set by the other process. This proof obligation is trivial to discharge since both transitions explicitly check *locked?*.

Now, consider the second set of definitions. Since the specification of the read transition does not check *locked?*, it seems that a process can successfully execute the read transition even if the other process has *locked?* set. Although this scenario would allow *value* to incorrectly end up being set to 1 rather than 2, it is possible to show this scenario cannot occur in the second composition since the *guar* for the composite is obtained by intersecting one component's *guar* with the other component's *hidd*. With the second set of definitions, *hidd* is the same as *rely*. Suppose that process *ONE* currently has *locked?* set and consider whether process *TWO* can execute the read transition. Since *guar* in the second set of definitions does not require the read transition to check *locked?*, the read transition from the given state is in *guar* for process *TWO*. However, the read transition would change *locker* to *TWO* violating the requirement in *rely* for process *ONE* that process *TWO* not be able to change *locker* when process *ONE* has *locked?* set. In summary, the definition of *guar* for the composite is such that the composite is prevented from reaching the bad state in which both processes have simultaneously set *locked?* even if *guar* is defined incorrectly as long as *hidd* is defined restrictively enough. In other words, the constraints in *hidd* represent scoping assumptions about variables and the stronger these assumptions are the more likely it is that the analysis will fail to detect an inconsistency between the *guar* for one component and the *rely* for another component.

## THEORY *thms*

```
thms: THEORY
  BEGIN

  IMPORTING ex

  IMPORTING common2

  st, st1, st2: VAR STATE

  ag: VAR AGENT

  cmp12_rr(st): bool =
    NOT locked?(st)
        AND value(st) = 0 AND pc(st)(ONE) = READ AND pc(st)(TWO) = READ

  cmp12_rw(st): bool =
    locked?(st)
        AND locker(st) = TWO
          AND value(st) = 0
            AND v(st)(TWO) = 0 AND pc(st)(ONE) = READ AND pc(st)(TWO) = WRITE

  cmp12_rd(st): bool =
    NOT locked?(st)
        AND value(st) = 1 AND pc(st)(ONE) = READ AND pc(st)(TWO) = DONE

  cmp12_wr(st): bool =
    locked?(st)
        AND locker(st) = ONE
          AND value(st) = 0
```

```
            AND v(st)(ONE) = 0 AND pc(st)(ONE) = WRITE AND pc(st)(TWO) = READ

cmp12_wd(st): bool =
  locked?(st)
      AND locker(st) = ONE
        AND value(st) = 1
          AND v(st)(ONE) = 1 AND pc(st)(ONE) = WRITE AND pc(st)(TWO) = DONE

cmp12_dr(st): bool =
  NOT locked?(st)
      AND value(st) = 1 AND pc(st)(ONE) = DONE AND pc(st)(TWO) = READ

cmp12_dw(st): bool =
  locked?(st)
      AND locker(st) = TWO
        AND value(st) = 1
          AND v(st)(TWO) = 1 AND pc(st)(ONE) = DONE AND pc(st)(TWO) = WRITE

cmp12_dd(st): bool =
  NOT locked?(st)
      AND value(st) = 2 AND pc(st)(ONE) = DONE AND pc(st)(TWO) = DONE

cmp12_inv: STATE_PRED =
  (LAMBDA st:
     cmp12_rr(st)
         OR cmp12_rw(st)
           OR cmp12_rd(st)
             OR cmp12_wr(st)
               OR cmp12_wd(st)
                 OR cmp12_dr(st) OR cmp12_dw(st) OR cmp12_dd(st))

steps_thm: THEOREM
      member((st1, st2, ag), steps(cmp12))
          =
        ((member((st1, st2, ag), guar(cmp1))
              AND member((st1, st2, ag), hidd(cmp2)))
             OR
           (member((st1, st2, ag), guar(cmp2))
                AND member((st1, st2, ag), hidd(cmp1)))
               OR
             (member((st1, st2, ag), rely(cmp1))
                  AND member((st1, st2, ag), rely(cmp2))))

rr_step: THEOREM
      member((st1, st2, ag), steps(cmp12)) AND cmp12_rr(st1)
          IMPLIES (cmp12_rw(st2) OR cmp12_wr(st2) OR cmp12_rr(st2))

rw_step: THEOREM
      member((st1, st2, ag), steps(cmp12)) AND cmp12_rw(st1)
          IMPLIES (cmp12_rd(st2) OR cmp12_rw(st2))

rd_step: THEOREM
      member((st1, st2, ag), steps(cmp12)) AND cmp12_rd(st1)
          IMPLIES (cmp12_wd(st2) OR cmp12_rd(st2))

wr_step: THEOREM
      member((st1, st2, ag), steps(cmp12)) AND cmp12_wr(st1)
          IMPLIES (cmp12_dr(st2) OR cmp12_wr(st2))

wd_step: THEOREM
      member((st1, st2, ag), steps(cmp12)) AND cmp12_wd(st1)
          IMPLIES (cmp12_dd(st2) OR cmp12_wd(st2))

dr_step: THEOREM
```

```
    member((st1, st2, ag), steps(cmp12)) AND cmp12_dr(st1)
        IMPLIES (cmp12_dw(st2) OR cmp12_dr(st2))

dw_step: THEOREM
    member((st1, st2, ag), steps(cmp12)) AND cmp12_dw(st1)
        IMPLIES (cmp12_dd(st2) OR cmp12_dw(st2))

dd_step: THEOREM
    member((st1, st2, ag), steps(cmp12)) AND cmp12_dd(st1)
        IMPLIES cmp12_dd(st2)

cmp12_init: THEOREM init_satisfies(cmp12, cmp12_inv)

cmp12_steps: THEOREM steps_satisfy(cmp12, stable(cmp12_inv))

cmp12_thm: THEOREM satisfies(cmp12, alwayss(cmp12_inv))

final: STATE_PRED =
  (LAMBDA st:
     pc(st)(ONE) = DONE AND pc(st)(TWO) = DONE IMPLIES value(st) = 2)

final_thm: THEOREM satisfies(cmp12, alwayss(final))

END thms
```

## 11.4 Summary

The fact that inconsistencies might not be detected in a specification could be viewed as suggesting that the framework is flawed. Instead, we view it as simply being a trade-off between the power of the framework and soundness. If an analyst chooses to do so, he can define *hidd* to make no assumptions at all. Then, each component's *guar* must be defined to respect the *rely* of other components for the proof obligations to be discharged. At the other extreme, if an analyst does not want to be bothered with proof obligations, he can choose *hidd* to be the same as *rely*. By doing so, the analyst is taking the risk of inconsistencies between *guar* and *rely*. As a compromise, the analyst can choose to pick *hidd* strongly enough to capture which variables are accessible by each component but weakly enough to cause proof obligations to be generated about how those variables are modified. We feel this provides a good compromise by allowing maintainable specifications to be written that still cause meaningful proof obligations to be generated. Although it is possible that inconsistencies could remain between *guar* and *rely* if one component's *guar* modifies a variable specified as being inaccessible by another component's *hidd*, we feel it is relatively simple to avoid such inconsistencies. For example, it is a relatively simple matter to note that process *TWO* does not reference a private variable of process *ONE* (such as *v(ONE)*). In contrast, it is more difficult to notice a "semantic" error such as the omission of the check of *locked?* in the read transition.

# Section 12 Correctness of Definition

It is interesting to note that the composition theorem would hold for many other definitions of composition. In particular, the theorem holds for any definition of composition such that the set of behaviors for the composite is a subset of the intersection of the behaviors for the components. Consequently, the composition theorem by itself is somewhat meaningless. To be of use, the definition of composition must satisfy an intuitive notion of composition as well as satisfy the composition theorem.

We propose the following as an intuitive requirement on composition:

> *The composition of a collection of components is meaningful if the behaviors of the composite are exactly those behaviors that are acceptable to each of the components.*

Another way of stating this requirement is that composition is essentially "conjunction" of components in that the behaviors of the composite are those that are acceptable to the first component, and the second component, ..., and the last component. In the Abadi-Lamport work, composition is actually defined simply as conjunction. Our approach here is slightly different in that we define composition in terms of structure we have imposed on components and then "test" whether a given composition of components is meaningful by checking whether the result is simply conjunction. Rather than directly testing whether a given composition is equivalent to conjunction, we use the following theorem:

> *If each component in a collection of components satisfies the environmental assumptions of each of the other components, then the composition of the components is meaningful.*

Here, the meaning of "satisfies the environmental assumptions" is as it was in Section 10. The proof consists of the following steps:

- Each behavior acceptable to the composite is acceptable to each individual component.

  This follows from the composition theorem using the set containing only the individual component for $S_1$ and the entire collection of components as $S_2$. Then, the composition theorem ensures that any property of the component is a property of the composite. From the definition of a component satisfying a property, we can conclude that the behaviors accepted by the composite are a subset of those accepted by the component.

- Each behavior acceptable to each individual component is acceptable to the composite.

  To prove this step, let $b$ be a behavior that is acceptable to each individual component and consider an arbitrary step, $tran$, in $b$. By definition, $tran$ is an element of either *guar* or *rely* for each component.

  - If $tran$ is an element of *rely* for each component, then $tran$ is an element of *rely* for the composite and consequently in *steps* for the composite.
  - Otherwise, $tran$ is an element of *guar* for a non-zero number of components and an element of *rely* for the remaining components. Since any component's *rely* is a subset

of its $hidd$, $tran$ is an element of $guar$ for at least one of the components and is an element of either $guar$ or $hidd$ for every component. Consequently, $tran$ is an element of $guar$ for the composite by definition.

- Since $b$ is accepted by each component, $sts(b)(0)$ is in each component's $init$. By the definition of composition, $sts(b)(0)$ is in the composite's $init$.
- By definition, $b$ is accepted by the composite since it starts with a state in the composite's $init$ and contains only transitions in the composite's $steps$.

## THEORY *compose_right*

```
compose_right[ST: NONEMPTY_TYPE, AG: NONEMPTY_TYPE]: THEORY
  BEGIN

  IMPORTING cmp_thm[ST,AG]

  IMPORTING compose_idempotent[ST,AG]

  cset: VAR (composable[ST,AG])

  b: VAR trace_t[ST,AG]

  cmp: VAR (comp_t)

  n: VAR nat

  cr_init: THEOREM (forall cmp: member(cmp,cset) implies
    member(b,prop_for(cmp))) implies initial_okay(compose(cset),b)

  cr_rely: THEOREM (forall cmp: member(cmp,cset) implies
    member(b,prop_for(cmp)) and
      member((sts(b)(n),sts(b)(n+1),ags(b)(n)),rely(cmp))) implies
          member((sts(b)(n),sts(b)(n+1),ags(b)(n)),steps(compose(cset)))

  cr_guar: THEOREM (forall cmp: member(cmp,cset) implies
    member(b,prop_for(cmp))) and
       (exists cmp: member(cmp,cset) and
    not member((sts(b)(n),sts(b)(n+1),ags(b)(n)),rely(cmp))) implies
          member((sts(b)(n),sts(b)(n+1),ags(b)(n)),steps(compose(cset)))

  cr_steps: THEOREM (forall cmp: member(cmp,cset) implies
    member(b,prop_for(cmp))) implies
          steps_okay(compose(cset),b)

  cr_wfar: THEOREM (forall cmp: member(cmp,cset) implies
             member(b,prop_for(cmp))) implies
                 is_wfar(compose(cset),b)


  cr_sfar: THEOREM (forall cmp: member(cmp,cset) implies
             member(b,prop_for(cmp))) implies
                 is_sfar(compose(cset),b)

  cr_aux: THEOREM (forall cmp: member(cmp,cset) implies
                member(b,prop_for(cmp))) implies
                 member(b,prop_for(compose(cset)))


  compose_right: THEOREM (forall cmp: member(cmp,cset) implies
```

```
    tolerates(singleton(cmp),cset)) implies
      ( (forall cmp: member(cmp,cset) implies
          member(b,prop_for(cmp))) iff
            member(b,prop_for(compose(cset))))

END compose_right
```

Section 13
# Proving Liveness

As discussed in Section 5, our definition of a component includes the notion of fairness assumptions for use in proving liveness properties. While the inclusion of these assumptions makes it possible to prove liveness properties, in practice it is useful to have higher level proof rules for liveness rather than doing proofs explicitly in terms of the fairness assumptions.

One starting point for developing such proof rules is the Abadi-Lamport work. However, we chose to base our work on Chandy and Misra's UNITY work [2] instead. We were aware of UNITY from work we had previously done outside the context of DTOS and did some experimentation with incorporating the UNITY work into the DTOS work on this other project. Given that we had already done this work, the easiest approach for us to use on DTOS was to simply finish integrating the UNITY work into the DTOS framework. Originally, we also believed that the UNITY proof rules were somewhat simpler than the Abadi-Lamport rules. However, further study of the Abadi-Lamport rules shows that the apparent additional complexity is due to:

- The UNITY proof rules are presented better. In particular, additional concepts are defined that simplify the final statement of the proof rules.
- The Abadi-Lamport proof rules address proving liveness properties of refinements. The UNITY proof rules we incorporated into the DTOS framework do not address refinement at all.

In summary, the UNITY work described here is almost identical to a subset of the Abadi-Lamport work. This means that it is consistent with the Abadi-Lamport work, but not yet complete. The remaining work to be done is in the area of refinement of systems which we have generally ignored throughout this report.

In general, a liveness property asserts that something eventually happens. Here, we consider only state predicates, so the liveness properties assert that certain classes of states are eventually reached. For example, we might assert that once a state is reached in which a kernel service has been requested, then eventually a state is reached in which the service has been provided.

The fairness assumptions ensure that as long as certain transition classes are enabled sufficiently often, then a transition from one of those classes will eventually occur. Suppose that $q$ is a property that is desired to eventually hold and $tranc$ and $p$ are such that, whenever $p$ holds in a state, a transition from transition class $tranc$ will result in $q$ holding. Then, $q$ can be shown to eventually hold as long as a point is reached where $p$ is repeatedly true for successive states, $tranc$ is enabled sufficiently often, and a fairness assumption is made about $tranc$. The key is that once $p$ becomes true, it remains true until $tranc$ occurs and causes $q$ to become true.

Given state predicates $sp_1$ and $sp_2$, $unless\_pred(sp_1, sp_2)$ is defined to be the action predicate denoting that $sp_1$ or $sp_2$ holds in the final state whenever $sp_1$ holds in the initial state and $sp_2$ does not hold in the initial state. Thus, $unless\_pred$ "recognizes" all transitions that either cause $sp_1$ to lead to $sp_2$ or preserve $sp_1$'s holding. In other words, the recognized transitions are those that keep $sp_1$ "stuck" on until $sp_2$ becomes true.

We define $unless(cmp, sp_1, sp_2)$ to denote that every transition in *steps* for $cmp$ satisfies $unless\_pred(sp_1, sp_2)$. This function returns true exactly when $cmp$'s allowable steps keep $sp_1$ "stuck" on until $sp_2$ becomes true.

UNITY defines $ensures(cmp, sp_1, sp_2)$ to denote that $unless(cmp, sp_1, sp_2)$ holds and there exists a transition class, $tranc$, that causes $sp_2$ to be true in the final state whenever $sp_1$ is true and $sp_2$ is false in the initial state. Our definition of $ensures$ must be slightly different because our fairness assumptions are of a different nature than UNITY's. In UNITY, transition classes are considered to always be enabled and the only fairness assumption is that every transition happens an infinite number of times. Transitions that are intended to occur only a finite number of times are modeled as being no-ops whenever their enabling conditions do not hold. Even though they "occur" infinitely many times more than intended, the extra occurrences are no-ops that are not of concern. Our framework provides a notion of fairness for only those transition classes identified in $sfar$ and $wfar$. Consequently, our definition of $ensures$ must require $tranc$ to be an element of either $wfar$ or $sfar$. We actually define two versions of $ensures$:

- $wensuresb(cmp, sp_1, sp_2)$ corresponds to the UNITY notion of $ensures$ with $tranc$ being an element of $wfar(cmp)$
- $ensuresb(cmp, sp_1, sp_2)$ corresponds to the UNITY notion of $ensures$ with $tranc$ being an element of $sfar(cmp)$[10]

We define $leads\_to(cmp, sp_1, sp_2)$ to denote that $cmp$ is such that whenever $sp_1$ holds in some state, then $sp_2$ holds in a later state. In other words, the function returns true whenever $cmp$ satisfies the property "whenever $sp_1$ holds $sp_2$ will eventually hold."[11] The following properties of $leads\_to$ are straightforward to prove:

- $(leads\_to1w)$ $wensuresb(cmp, sp_1 \wedge enabled\_sp(tranc), sp_2, tranc) \Rightarrow$ $leads\_to(cmp, sp_1 \wedge enabled\_sp(tranc), sp_2)$

  Here, $enabled\_sp(tranc)$ is the state predicate indicating whether $tranc$ is enabled in a given state. This rule allows liveness properties to be derived from weak fairness assumptions by showing that $wensuresb$ holds. Since $wensuresb$ requires consideration of only transitions in isolation, this allows the proof of a temporal property ($leads\_to$) to be reduced to analysis of individual transitions. The proof of the rule is as follows:

  - $wensuresb$ requires that once $sp_1$ holds and $tranc$ is enabled, $sp_1$ and $tranc$ being enabled continues to hold until $sp_2$ holds.
  - So, $sp_2$ eventually holds unless either $sp_1$ and $tranc$ being enabled never hold simultaneously or a point is reached from which they hold continuously.
  - The former case can be ignored since the goal is to show that if $sp_1$ and $tranc$ being enabled both hold at some point, then eventually $sp_2$ holds.
  - In the latter case, the weak fairness assumption on $tranc$ ensures $tranc$ eventually occurs. Then, $wensuresb$ requires that it cause $sp_2$ to become true.

- $(leads\_to1)$ $ensuresb(cmp, sp_1, sp_2, tranc) \wedge leads\_to(cmp, sp_1, enabled\_sp(tranc)) \Rightarrow$ $leads\_to(cmp, sp_1, sp_2)$.

  This is the proof rule for proving liveness properties from strong fairness assumptions. Here it is necessary to show that $sp_1$ causes $tranc$ to eventually be enabled in addition to proving $ensuresb$ holds. The proof of the rule is as follows:

[10] A better name for this function would be $sensuresb$.

[11] This would be formalized as $\Box(sp_1 \Rightarrow \Diamond sp_2)$ in temporal logic.

- Suppose $sp_1$ is true at some point. Then we must show $sp_2$ eventually holds.
- Assume $sp_2$ never holds at or later than the point $sp_1$ holds.
- $ensuresb$ requires $sp_1$ hold from that point on.
- The $leads\_to$ assumption means $tranc$ cannot be stuck disabled because for each of the infinite number of states in the tail at which $sp_1$ holds, $tranc$ must be enabled at that point or later.
- So, strong fairness implies $tranc$ occurs infinitely often.
- $ensuresb$ implies $sp_2$ holds infinitely often.
- So, $sp_2$ eventually holds after $sp_1$.

- ($leads\_to2$) $(sp_1 \Rightarrow sp_2) \wedge leads\_to(cmp, sp_2, sp) \Rightarrow leads\_to(cmp, sp_1, sp)$

  Proof: If $sp_1$ holds at any point, then $sp_2$ is assumed to hold then, too. It is also assumed that whenever $sp_2$ holds, then $sp$ eventually holds.

- ($leads\_to3$) $(sp_1 \Rightarrow sp_2) \wedge leads\_to(cmp, sp, sp_1) \Rightarrow leads\_to(cmp, sp, sp_2)$

  Proof: Similar to that for $leads\_to2$.

- ($leads\_to\_or$) $leads\_to(cmp, sp_1, sp) \wedge leads\_to(cmp, sp_2, sp) \Rightarrow leads\_to(cmp, sp_1 \vee sp_2, sp)$

  Proof: If $sp_1$ and $sp_2$ both individually ensure that $sp$ eventually holds, then $sp$ is guaranteed to eventually hold if one of $sp_1$ or $sp_2$ holds.

- ($leads\_to\_tran$) $leads\_to(cmp, sp, sp_1) \wedge leads\_to(cmp, sp_1, sp_2) \Rightarrow leads\_to(cmp, sp, sp_2)$

  Proof: $sp$ guarantees that $sp_1$ eventually occurs which itself guarantees that $sp_2$ eventually occurs.

- ($leads\_to\_true$) If $true$ leads to $sp$ holding, then $sp$ holds infinitely often.

  Proof: Since $true$ always holds, the assumption is that it is always true that $sp$ holds at a later time. Thus, $sp$ must hold an infinite number of times.

- ($leads\_to\_stable$) If $true$ leads to $sp$ holding and every step allowed by $cmp$ holds $sp$ stable, then eventually $sp$ holds continuously.

  Proof: Since $true$ always holds, $sp$ must hold eventually. Once it holds, it must continue to hold forever since the steps allowed by $cmp$ hold it stable.

- ($leads\_to\_invariant$) If $sp_2$ always holds and $sp_1$ leads to $sp$, then $sp_1 \wedge sp_2$ leads to $sp$, $sp_1$ leads to $sp \wedge sp_2$, and $sp_1 \wedge sp_2$ leads to $sp \wedge sp_2$.

  Proof: Consider the first claim. Suppose $sp_1 \wedge sp_2$ holds at some point. The goal is to show that $sp$ eventually holds. Whenever $sp_1 \wedge sp_2$ holds, then $sp_1$ obviously holds. Since $sp_1$ leads to $sp$, $sp$ must eventually hold. As a more complicated example, consider the second claim. Suppose $sp_1$ holds at some point. Then, the goal is to show that $sp \wedge sp_2$ holds at a later point. By assumption, $sp_1$ leads to $sp$. So, $sp$ holds at a later point. Since $sp_2$ is assumed to always hold, $sp \wedge sp_2$ holds at the later point. The proof of the third claim is similar.

- ($leads\_to\_invariant1$) If $sp_2$ always holds, and $sp_1$ leads to $sp$, and $sp_3$ and $sp_2$ together imply $sp_1$, then $sp_3$ leads to $sp$.

  Proof: Suppose $sp_3$ holds at some point. The goal is to show that $sp$ holds at a later point. Since $sp_2$ is assumed to always hold, it holds at the point $sp_3$ holds. Then, the assumption is that $sp_1$ holds, too. Finally, the assumption that $sp_1$ leads to $sp$ guarantees that $sp$ becomes true at a later point.

The above rules greatly simplify the proofs of liveness properties. Note, however, that liveness properties are still fairly difficult to prove because:

- To get from the fairness assumptions to a leads to relation, it is necessary to prove an *ensures* relation. This in turn requires proving an *unless* relation which requires consideration of every class of transitions. Thus, each transformation from fairness assumption to leads to requires consideration of all of the different types of operations in the system. While the proofs are straightforward, they are tedious and time consuming.

  This issue seems to be an inherent difficulty. Often when a system fails to satisfy a desired liveness property, investigation reveals that concurrently operating processes did not follow the proper protocol. As a specific example, an implementation error might result in a lock request made by a process being discarded. If the system requires a resource to be locked before providing service, then the loss of the lock request could result in service not being provided. If $p$ is thought of as "the resource is locked" and $q$ is thought of as "service is provided", then the intended design of the system is that $unless\_pred(p, q)$ hold. However, the transition that causes the lock request to be lost violates this predicate. Ensuring that this does not occur requires considering all of the transitions to see that they keep $p$ true until $q$ becomes true.

- In many cases, the desired liveness property cannot be jumped to immediately. Instead, a divide-and-conquer approach must be used in which it is shown that $sp_1$ leads to $sp_2$ which leads to $sp_3$ ...Since each of the intermediate steps involves proving a leads to relation (which was argued to be non-trivial in the previous bullet), the proof of the overall liveness property desired is non-trivial, too.

## THEORY *unity*

```
unity[ST: NONEMPTY_TYPE, AG: NONEMPTY_TYPE]: THEORY
  BEGIN

  IMPORTING preds[ST, AG]

  sp, sp1, sp2, sp3: VAR STATE_PRED

  cmp: VAR (comp_t)

  st, st1, st2: VAR ST

  ag: VAR AG

  tranc: VAR TRANSITION_CLASS

  t: VAR trace_t

  p1, p2: VAR prop_t

  pimplies(p1, p2): prop_t = (LAMBDA t: member(t, p1) IMPLIES member(t, p2))

  por(p1, p2): prop_t = (LAMBDA t: member(t, p1) OR member(t, p2))

  negate_sp(sp): STATE_PRED = (LAMBDA st: NOT sp(st))

  unless_pred(sp1, sp2): ACTION_PRED =
    (LAMBDA st1, st2, ag:
```

```
      sand(sp1, negate_sp(sp2))(st1) IMPLIES sor(sp1, sp2)(st2))

unless(cmp, sp1, sp2): bool = satisfies(cmp, alwaysa(unless_pred(sp1, sp2)))

unless_help: THEOREM
      steps_satisfy(cmp, unless_pred(sp1, sp2)) IMPLIES unless(cmp, sp1, sp2)

i, j, k, l, m: VAR nat

ip: VAR [nat -> bool]

ip_help: THEOREM
      ip(m)
          AND
        (FORALL i: (FORALL j: m <= j AND j < i IMPLIES ip(j)) IMPLIES ip(i))
          IMPLIES (FORALL k: (FORALL l: m <= l AND l <= k IMPLIES ip(l)))

ip_help1: THEOREM
      ip(m)
          AND
        (FORALL i: (FORALL j: m <= j AND j < i IMPLIES ip(j)) IMPLIES ip(i))
          IMPLIES (FORALL k: m <= k IMPLIES ip(k))

unless_prop1: THEOREM
      unless(cmp, sp1, sp2)
          IMPLIES
        satisfies(cmp,
                  always(pimplies(stbp(sp1),
                                  por(alwayss(sp1),
                                      eventuallys(sp2)))))

unless_prop2: THEOREM
      unless(cmp, sp1, sp2) AND prop_for(cmp)(t) AND sp1(sts(t)(i))
          IMPLIES
        ((FORALL j: sp1(sts(t)(i + j)))
             OR
           (EXISTS k:
              sp2(sts(t)(i + k))
                  AND (FORALL l: l < k IMPLIES sp1(sts(t)(i + l)))))

ensuresb(cmp, sp1, sp2, tranc): bool =
  unless(cmp, sp1, sp2)
      AND member(tranc, sfar(cmp))
        AND
      (FORALL st1, st2, ag:
         (member((st1, st2, ag), tranc) AND sp1(st1) AND NOT sp2(st1))
             IMPLIES sp2(st2))

ensures(cmp, sp1, sp2): bool =
  unless(cmp, sp1, sp2) AND (EXISTS tranc: ensuresb(cmp, sp1, sp2, tranc))

wensuresb(cmp, sp1, sp2, tranc): bool =
  unless(cmp, sp1, sp2)
      AND member(tranc, wfar(cmp))
        AND
      (FORALL st1, st2, ag:
         (member((st1, st2, ag), tranc) AND sp1(st1) AND NOT sp2(st1))
             IMPLIES sp2(st2))

wensures(cmp, sp1, sp2): bool =
  unless(cmp, sp1, sp2) AND (EXISTS tranc: wensuresb(cmp, sp1, sp2, tranc))

enabled_sp(tranc): STATE_PRED = (LAMBDA st: enabled(tranc, st))
```

```
leads_to(cmp, sp1, sp2): bool =
  satisfies(cmp, always(pimplies(stbp(sp1), eventuallys(sp2))))

leads_to1: THEOREM
      ensuresb(cmp, sp1, sp2, tranc)
          AND leads_to(cmp, sp1, enabled_sp(tranc))
          IMPLIES leads_to(cmp, sp1, sp2)

leads_to1w: THEOREM
      wensuresb(cmp, sand(enabled_sp(tranc), sp1), sp2, tranc)
          IMPLIES leads_to(cmp, sand(enabled_sp(tranc), sp1), sp2)

leads_to_2: THEOREM
      (FORALL st: simplies(sp1, sp2)(st)) AND leads_to(cmp, sp2, sp)
          IMPLIES leads_to(cmp, sp1, sp)

leads_to_3: THEOREM
      (FORALL st: simplies(sp1, sp2)(st)) AND leads_to(cmp, sp, sp1)
          IMPLIES leads_to(cmp, sp, sp2)

leads_to_or: THEOREM
      leads_to(cmp, sp1, sp) AND leads_to(cmp, sp2, sp)
          IMPLIES leads_to(cmp, sor(sp1, sp2), sp)

leads_to_tran: THEOREM
      leads_to(cmp, sp, sp1) AND leads_to(cmp, sp1, sp2)
          IMPLIES leads_to(cmp, sp, sp2)

true_sp(st): bool = TRUE

leads_to_true: THEOREM
      leads_to(cmp, true_sp, sp)
          IMPLIES satisfies(cmp, always(eventuallys(sp)))

leads_to_stable: THEOREM
      leads_to(cmp, true_sp, sp) AND steps_satisfy(cmp, stable(sp))
          IMPLIES satisfies(cmp, eventually(alwayss(sp)))

leads_to_invariant: THEOREM
      leads_to(cmp, sp1, sp) AND satisfies(cmp, alwayss(sp2))
          IMPLIES leads_to(cmp, sand(sp1, sp2), sp)
            AND leads_to(cmp, sp1, sand(sp, sp2))
              AND leads_to(cmp, sand(sp1, sp2), sand(sp, sp2))

leads_to_invariant1: THEOREM
      leads_to(cmp, sp1, sp) AND satisfies(cmp, alwayss(sp2))
          IMPLIES
        ((FORALL st: simplies(sand(sp3, sp2), sp1)(st))
             IMPLIES leads_to(cmp, sp3, sp))

END unity
```

*Section* 14
# State and Agent Translation

It is typically the case that different components have different states and agents. This results in the properties defined for the components being type incompatible. We address this using translator functions that map elements of one type to another type. Such a function must map each source element to a non-empty set of target elements in such a way that no two sets of target elements overlap. We term any such function a *weak translator*. If in addition, a function maps some element of the source type to each element of the target type, then we term the function a *translator*.

Given a set $s$ and a translator (or weak translator) $t$, we use *tmap(t,s)* to denote the set of elements to which $t$ maps some element of $s$. In other words, *tmap* "maps" the translation $t$ across the set $s$. The function *tmap* distributes over set union and intersection.

We allow the translators to return a set of values rather than a single value to address different levels of abstraction. For example, a state might be mapped to a more detailed representation in which some parts of the state are unconstrained by the components of the more abstract state. Then, multiple more detailed states might correspond to each of the more abstract states. With regard to agents, what appears to be a single agent at a certain level of abstraction might be seen to be multiple agents at a lower level of abstraction. For example, the more abstract model might view agents as being processes while a more detailed model might view agents as being threads executing within the processes.

For convenience, we define:

- $trone(t, x)$ to be an arbitrary element in the set to which $t$ maps $x$. This function is defined for both translators and weak translators.
- $trinv(t, y)$ to be an $x$ (in fact, the unique $x$) that $t$ maps to $y$. This function is defined only for translators since in the case of weak translators there could be certain $y$ values with no corresponding $x$ values.

Theorem *inv_trans_prop* demonstrates that if an element *bt* of *base_translator_t* is the "inverse" of a projection function from $Y$ to $X$ that covers $X$, then *bt* is in *translator_t* (and hence *weak_translator_t*).

## THEORY *translators*

```
translators[X: NONEMPTY_TYPE, Y: NONEMPTY_TYPE]: THEORY
  BEGIN

  base_translator_t: TYPE = [X -> setof[Y]]

  inv_translator_t: TYPE = [Y -> X]

  bt: VAR base_translator_t

  it: VAR inv_translator_t
```

```
x, x1, x2: VAR X

y, y1, y2: VAR Y

weak_translator_t(bt): bool =
  (FORALL x: bt(x) /= emptyset)
      AND
    (FORALL x1, x2: x1 /= x2 IMPLIES intersection(bt(x1), bt(x2)) = emptyset)

t: VAR (weak_translator_t)

translator_t(t): bool = (FORALL y: (EXISTS x: member(y, t(x))))

t1: VAR (translator_t)

r, s: VAR setof[X]

tmap(bt, s): setof[Y] = (LAMBDA y: (EXISTS x: member(x, s) AND
      member(y, bt(x))))

s1: VAR setof[Y]

help1: THEOREM s1 /= emptyset IFF (EXISTS y: s1(y))

help2: THEOREM s /= emptyset IMPLIES (EXISTS x: s(x))

help3: THEOREM t(x1)(y) AND t(x2)(y) IMPLIES x1 = x2

help4: THEOREM (EXISTS y: t(x)(y))

help5: THEOREM (EXISTS x: t1(x)(y))

tmap_union: THEOREM tmap(t, union(r, s)) = union(tmap(t, r), tmap(t, s))

tmap_intersection: THEOREM
      tmap(t, intersection(r, s)) = intersection(tmap(t, r), tmap(t, s))

trone(t, x): Y = choose(t(x))

trone_def: THEOREM t(x)(trone(t, x))

trinv(t1, y): X = choose(LAMBDA x: member(y,t1(x)))

trinv_def: THEOREM t1(trinv(t1, y))(y)

inv_trans_prop: THEOREM
  (FORALL x:
      bt(x) = {y | it(y) = x}
      AND (EXISTS y: it(y) = x))
  => weak_translator_t(bt) AND translator_t(bt)

END translators
```

Theory *id_tran* defines the identity translator *idt*. The expression $idt(x)$ denotes $\{x\}$.

## THEORY *idtran*

```
idtran[X: NONEMPTY_TYPE]: THEORY
  BEGIN

  IMPORTING translators[X,X]

  x, y : VAR X

  idt: (translator_t) =
    (LAMBDA x: {y | y = x})

  END idtran
```

We use the expression $brmap(t, br)$ to denote the relation on the target elements to which $t$ maps a relation $br$. In other words, two elements $y_1$ and $y_2$ are related in the resulting relation exactly when there exist $x_1$ and $x_2$ such that:

- $t$ maps $x_1$ and $x_2$ to, respectively, $y_1$ and $y_2$, and
- $x_1$ and $x_2$ are related by $br$

If $v$ is an equivalence relation, the expression $vmap(t, v)$ denotes the equivalence relation on the target elements to which $t$ maps $v$. The function $vmap$ is simply a restriction of $brmap$ to equivalence relations. The $brmap$ function distributes over set union and intersection, and the $vmap$ function distributes over set intersection.

Note that we only define $brmap$ and $vmap$ for translators. If $t$ is a weak translator, then $vmap(t, v)$ is not necessarily an equivalence relation even if $v$ is. Requiring $t$ be a translator, however, is sufficient to ensure that $vmap(t, v)$ is an equivalence relation whenever $v$ is.

## THEORY *translator_views*

```
translator_views[X: NONEMPTY_TYPE, Y: NONEMPTY_TYPE]: THEORY
  BEGIN

  IMPORTING translators[X, Y]

  IMPORTING views[X]

  IMPORTING views[Y]

  t: VAR (translator_t)

  v, v1, v2: VAR (VIEWS[X])

  vy: VAR (VIEWS[Y])

  br, br1, br2: VAR BASE_RELATIONS[X]

  x, x1, x2: VAR X

  y, y1, y2: VAR Y

  vmap(t, v): (VIEWS[Y]) =
    (LAMBDA y1, y2:
       (EXISTS x1, x2:
          member((x1, x2), v) AND member(y1, t(x1)) AND member(y2, t(x2))))
```

```
  brmap(t, br): BASE_RELATIONS[Y] =
    (LAMBDA y1, y2:
       (EXISTS x1, x2:
          member((x1, x2), br) AND member(y1, t(x1)) AND member(y2, t(x2))))

  brmap_intersection: THEOREM
        brmap(t, intersection(br1, br2))
            = intersection(brmap(t, br1), brmap(t, br2))

  brmap_union: THEOREM
        brmap(t, union(br1, br2)) = union(brmap(t, br1), brmap(t, br2))

  vmap_brmap: THEOREM vmap(t, v) = brmap(t, v)

  vmap_intersection: THEOREM
        vmap(t, intersection(v1, v2)) = intersection(vmap(t, v1), vmap(t, v2))

  END translator_views
```

We use the expression $tr\_ac(ap, xt, yt)$ to denote the set of transitions to which $xt$ and $yt$ map a set of transitions $ap$. The result is a set of transitions with states and agents of the types mapped to by $xt$ and $yt$. More specifically, a transition $(x_1, x_2, y)$ is an element of the result exactly when there exists $a_1$, $a_2$, and $b$ such that:

- $xt$ maps $a_1$ and $a_2$ to, respectively, $x_1$ and $x_2$,
- $yt$ maps $b$ to $y$, and
- $(a_1, a_2, b)$ is a transition in $ap$

The $tr\_ac$ function distributes over both set union and intersection.

Note that we only define $tr\_ac$ for $xt$ that are translators. As noted previously in the discussion of $vmap$, the mapping for the state function generally needs to be a translator rather than a weak translator. In the definition of $tr\_ac$, we specify $yt$ as a weak translator. However, using a translator as $yt$ is acceptable, too, since any translator is also a weak translator.

## THEORY *ac_translators*

```
ac_translators[X1: NONEMPTY_TYPE, Y1: NONEMPTY_TYPE,
               X: NONEMPTY_TYPE, Y: NONEMPTY_TYPE]:
THEORY
  BEGIN

  IMPORTING translators[X1, X]

  IMPORTING translators[Y1, Y]

  ap, ap1, ap2: VAR setof[[X1, X1, Y1]]

  xt: VAR (translator_t[X1, X])

  yt: VAR (weak_translator_t[Y1, Y])

  x1, x2: VAR X
```

```
y: VAR Y

a1, a2: VAR X1

b: VAR Y1

tr_ac(ap, xt, yt): setof[[X, X, Y]] =
   (LAMBDA x1, x2, y:
      (EXISTS a1, a2, b:
         member((a1, a2, b), ap)
            AND member(x1, xt(a1))
              AND member(x2, xt(a2)) AND member(y, yt(b))))

tr_ac_intersection: THEOREM
      tr_ac(intersection(ap1, ap2), xt, yt)
         = intersection(tr_ac(ap1, xt, yt), tr_ac(ap2, xt, yt))

tr_ac_union: THEOREM
      tr_ac(union(ap1, ap2), xt, yt)
         = union(tr_ac(ap1, xt, yt), tr_ac(ap2, xt, yt))

END ac_translators
```

Similarly, we define $tr\_tcs(tcs, xt, yt)$ to translate a set of transition classes, $tcs$, using $xt$ and $yt$. The result is the set containing transition classes resulting from using $tr\_ac$ to translate the transition classes in $tcs$.

Note that we only define $tr\_tcs$ when $xt$ is a translator (rather than a weak translator). The rationale here is the same as that given previously under the discussion of the definition of $tr\_ac$.

## THEORY *tcs_translators*

```
tcs_translators[X1: NONEMPTY_TYPE, Y1: NONEMPTY_TYPE,
                X: NONEMPTY_TYPE, Y: NONEMPTY_TYPE]:
THEORY
  BEGIN

  IMPORTING ac_translators[X1, Y1, X, Y]

  tca : VAR setof[[X1,X1,Y1]]

  tcb : VAR setof[[X,X,Y]]

  tcsa, tcsa1, tcsa2 : VAR setof[setof[[X1,X1,Y1]]]

  xt: VAR (translator_t[X1, X])

  yt: VAR (weak_translator_t[Y1, Y])

  tr_tcs(tcsa, xt, yt): setof[setof[[X,X,Y]]] =
     (LAMBDA tcb: (exists tca: member(tca,tcsa) and tr_ac(tca,xt,yt) = tcb))

  tr_tcs_union: THEOREM
        tr_tcs(union(tcsa1, tcsa2), xt, yt)
           = union(tr_tcs(tcsa1, xt, yt), tr_tcs(tcsa2, xt, yt))
```

**END** *tcs_translators*

---

A component can be translated to another state type and agent type by using translation functions. The translation is fairly straightforward using *tmap*, *vmap*, *tr_ac*, and *tr_tcs*. The only twist is that the manner in which the *hidd* and *rely* fields of a component are mapped is more complicated than simply using *tr_ac* (see below). We use $tr\_cmp(cmp_1, xt, yt)$ to denote the translation of $cmp_1$ using $xt$ and $yt$. We define the result as follows:

- $init \leftarrow tmap(init(cmp_1), xt)$
- $cags \leftarrow tmap(cags(cmp_1), yt)$
- $view \leftarrow vmap(view(cmp_1), xt)$
- $guar \leftarrow tr\_ac(guar(cmp_1), xt, yt)$
- $wfar \leftarrow tr\_tcs(wfar(cmp_1), xt, yt)$
- $sfar \leftarrow tr\_tcs(sfar(cmp_1), xt, yt)$
- $rely \leftarrow tr\_ac(rely(cmp_1), xt, yt) \cup env\_stutter(cmp_1, xt, yt)$

  Here, $env\_stutter(cmp_1, xt, yt)$ returns the set of transitions $(st_1, st_2, ag)$ such that $yt$ does not map any element of $cags(cmp_1)$ to $ag$ and $st_1$ and $st_2$ are equivalent with respect to the view resulting from mapping $view(cmp_1)$ with $xt$. Intuitively, the resulting set is the set of stuttering steps by environment agents for the translated component. This set must be added when the translation is done since $yt$ could be a weak translator (as opposed to a translator). When $yt$ is a translator, $env\_stutter$ adds nothing, and $rely$ and $hidd$ are really just defined by $tr\_ac$. When $yt$ is a weak translator, $env\_stutter$ adds those stuttering steps for the translated component that are not mapped to by any of the stuttering steps of the original component (because the agent for the translated component has no representation in the original component). To ensure that $rely$ for the translated component contains all of the stuttering steps for the environment, it is necessary to explicitly add them in. Doing so does not alter the meaning of the component since it simply makes explicit that no-ops by environment agents are acceptable regardless of whether they are by agents known to the component.
- $hidd \leftarrow tr\_ac(hidd(cmp_1), xt, yt) \cup env\_stutter(cmp_1, xt, yt)$

Given this definition, it is straightforward to show that any translation of a component satisfies the requirements on components defined in Section 5. In other words, translating a component always results in a component.

Note that we only define $tr\_cmp$ when $xt$ is a translator (rather than a weak translator). The rationale here is the same as that given previously under the discussion of the definition of $tr\_ac$.

## THEORY *cmp_translators*

---

*cmp_translators*[*X1*: *NONEMPTY_TYPE*, *Y1*: *NONEMPTY_TYPE*,
*X*: *NONEMPTY_TYPE*, *Y*: *NONEMPTY_TYPE*]:

```
THEORY
  BEGIN

  IMPORTING translator_views[X1, X]

  IMPORTING tcs_translators[X1, Y1, X, Y]

  IMPORTING component[X1, Y1]

  IMPORTING component[X, Y]

  cmp1: VAR (comp_t[X1, Y1])

  xt: VAR (translator_t[X1, X])

  yt: VAR (weak_translator_t[Y1, Y])

  x1, x2: VAR X
  y: VAR Y

  env_stutter(cmp1,xt,yt): setof[[X,X,Y]] =
      (LAMBDA x1,x2,y: not member(y,tmap(yt,cags(cmp1))) and
         vmap(xt,view(cmp1))(x1,x2))

  tr_cmp(cmp1, xt, yt): base_comp_t[X, Y] =
     (# init := tmap(xt, init(cmp1)),
        cags := tmap(yt, cags(cmp1)),
        view := vmap(xt, view(cmp1)),
        hidd := union(tr_ac(hidd(cmp1),xt, yt),env_stutter(cmp1,xt,yt)),
        rely := union(tr_ac(rely(cmp1), xt, yt),env_stutter(cmp1,xt,yt)),
        guar := tr_ac(guar(cmp1), xt, yt),
        sfar := tr_tcs(sfar(cmp1), xt, yt),
        wfar := tr_tcs(wfar(cmp1), xt, yt)
#)

 tranc : VAR setof[[X1,X1,Y1]]
 ag_set : VAR setof[Y1]
 v : VAR (VIEWS[X1])

 tr_gen_view_restriction: THEOREM
   gen_view_restriction(tranc,v) implies
     gen_view_restriction(tr_ac(tranc,xt,yt),vmap(xt,v))

 tr_gen_stuttering_restriction: THEOREM
   gen_stuttering_restriction(ag_set,tranc,v) implies
     gen_stuttering_restriction(tmap(yt,ag_set),tr_ac(tranc,xt,yt),vmap(xt,v))

  tr_cmp_init: THEOREM init_restriction(tr_cmp(cmp1, xt, yt))

  tr_cmp_guar: THEOREM guar_restriction(tr_cmp(cmp1, xt, yt))

  tr_cmp_rely_hidd: THEOREM rely_hidd_restriction(tr_cmp(cmp1, xt, yt))

  tr_cmp_hidd: THEOREM hidd_restriction(tr_cmp(cmp1, xt, yt))

  tr_cmp_cags: THEOREM cags_restriction(tr_cmp(cmp1, xt, yt))

  tr_cmp_view_rely: THEOREM view_rely_restriction(tr_cmp(cmp1, xt, yt))

  tr_cmp_view_hidd: THEOREM view_hidd_restriction(tr_cmp(cmp1, xt, yt))

  tr_cmp_view_guar: THEOREM view_guar_restriction(tr_cmp(cmp1, xt, yt))

  tr_cmp_view_init: THEOREM view_init_restriction(tr_cmp(cmp1, xt, yt))
```

```
tr_cmp_view_wfar: THEOREM view_wfar_restriction(tr_cmp(cmp1, xt, yt))

tr_cmp_view_sfar: THEOREM view_sfar_restriction(tr_cmp(cmp1, xt, yt))

tr_cmp_guar_stuttering: THEOREM
  guar_stuttering_restriction(tr_cmp(cmp1, xt,yt))

tr_cmp_rely_stuttering: THEOREM
  rely_stuttering_restriction(tr_cmp(cmp1, xt,yt))

tr_cmp_type: THEOREM comp_t(tr_cmp(cmp1, xt, yt))

tran_cmp(cmp1, xt, yt): (comp_t[X, Y]) = tr_cmp(cmp1, xt, yt)

END cmp_translators
```

We use $pmap(p_1, sttran_1, agtran_1)$ to denote the behavior predicate to which behavior predicate $p_1$ is mapped by $sttran_1$ and $agtran_1$.

## THEORY *tprops*

```
tprops[ST: NONEMPTY_TYPE, ST1: NONEMPTY_TYPE,
       AG: NONEMPTY_TYPE, AG1: NONEMPTY_TYPE]:
THEORY
  BEGIN

  IMPORTING props[ST, AG]

  IMPORTING props[ST1, AG1]

  IMPORTING translators[ST1, ST]

  IMPORTING translators[AG1, AG]

  t1: VAR trace_t[ST1, AG1]

  t: VAR trace_t[ST, AG]

  p1: VAR prop_t[ST1, AG1]

  p2: VAR prop_t[ST1,AG1]

  p: VAR prop_t[ST, AG]

  sttran1: VAR (translator_t[ST1, ST])

  agtran1: VAR (weak_translator_t[AG1, AG])

  n: VAR nat

  bmap1_base(sttran1, agtran1):
      [trace_t[ST1, AG1] -> [trace_t[ST, AG] -> bool]] =
    (LAMBDA t1:
       (LAMBDA t:
          (FORALL n:
             sttran1(sts(t1)(n))(sts(t)(n))
                 AND agtran1(ags(t1)(n))(ags(t)(n)))))
```

```
bmap1(sttran1, agtran1):
    (weak_translator_t[(trace_t[ST1, AG1]), (trace_t[ST, AG])]) =
  bmap1_base(sttran1, agtran1)

bmap1_strong: THEOREM
  translator_t(agtran1)
  => translator_t[(trace_t[ST1, AG1]), (trace_t[ST, AG])](bmap1(sttran1,agtran1))

bmap(t1, sttran1, agtran1):
    setof[trace_t[ST, AG]] = bmap1(sttran1, agtran1)(t1)

pmap1(sttran1, agtran1): [prop_t[ST1, AG1] -> prop_t[ST, AG]] =
  (LAMBDA p1:
     (LAMBDA t:
           (EXISTS t1: bmap(t1, sttran1, agtran1)(t) AND p1(t1))))

pmap(p1, sttran1, agtran1):
    prop_t[ST, AG] = pmap1(sttran1, agtran1)(p1)


END tprops
```

Theory *tcprops* provides several theorems regarding translated components and the properties they satisfy. The following theorems are used later in the analysis of the example:

- (*tcprop*1) *If component cmp satisfies p1, p is the translation of p1 under sttran1 and agtran1, and tcmp is the translation of cmp under sttran1 and agtran1, then the composite of {tcmp} satisfies p.*
- (*tolerates_cags_trans_prop*) *If for every transition, (st1, st2, ag1), in hidd(cmp) either ag1 is in the set ags or st1 and st2 look the same to view(cmp), then for every transition, (st3, st4, ag2), in the hidd of the translation of cmp under sttran1 and agtran1 either ag2 is in the translation of ags under agtran1 or st3 and st4 look the same to the view of the translated cmp.*
- (*disjoint_cags*) *If ag2 is in cags of the translation of cmp under sttran1 and agtran1 and ag2 is in the translation of a set of agents ags under agtran1, then there exists an agent ag1 that is in cags(cmp) and in ags.*

## THEORY *tcprops*

```
tcprops[ST: NONEMPTY_TYPE, ST1: NONEMPTY_TYPE,
        AG: NONEMPTY_TYPE, AG1: NONEMPTY_TYPE]:
THEORY
  BEGIN

  IMPORTING tprops

  IMPORTING cprops

  IMPORTING cmp_translators

  IMPORTING compose_idempotent
```

```
tcmp: VAR (comp_t[ST,AG])

cmp: VAR (comp_t[ST1,AG1])

p1: VAR prop_t[ST1, AG1]

p: VAR prop_t[ST, AG]

t: VAR trace_t[ST, AG]

t1: VAR trace_t[ST1, AG1]

st1,st2: VAR ST1
st3,st4: VAR ST
ag1: VAR AG1
ag2: VAR AG
ags: VAR setof[AG1]

sttran1: VAR (translator_t[ST1, ST])

agtran1: VAR (translator_t[AG1, AG])


preimage_initial_okay : THEOREM
  (bmap(t1, sttran1, agtran1)(t)
       AND initial_okay(tran_cmp(cmp, sttran1, agtran1), t))
    IMPLIES initial_okay(cmp, t1)

preimage_steps_okay : THEOREM
  (bmap(t1, sttran1, agtran1)(t)
       AND steps_okay(tran_cmp(cmp, sttran1, agtran1), t))
    IMPLIES steps_okay(cmp, t1)

preimage_is_wfar : THEOREM
  (bmap(t1, sttran1, agtran1)(t)
       AND is_wfar(tran_cmp(cmp, sttran1, agtran1), t))
    IMPLIES is_wfar(cmp, t1)

preimage_is_sfar : THEOREM
  (bmap(t1, sttran1, agtran1)(t)
       AND is_sfar(tran_cmp(cmp, sttran1, agtran1), t))
    IMPLIES is_sfar(cmp, t1)

prop_for_preimage: LEMMA
  prop_for(tran_cmp(cmp, sttran1, agtran1))(t)
  => (EXISTS (t1: trace_t[ST1, AG1]): bmap(t1, sttran1, agtran1)(t)
                 AND prop_for(cmp)(t1))

tcprop1: LEMMA
  satisfies(cmp, p1)
       AND pmap(p1, sttran1, agtran1) = p
       AND tcmp = tran_cmp(cmp, sttran1, agtran1)
  => satisfies(compose(singleton(tcmp)), p)

tolerates_cags_trans_prop: LEMMA
  ((FORALL st1, st2, ag1:
      hidd(cmp)(st1, st2, ag1)
         => ags(ag1) OR view(cmp)(st1, st2)))
  IMPLIES
    (hidd(tran_cmp(cmp,sttran1,agtran1))(st3, st4, ag2)
      => tmap(agtran1,ags)(ag2)
          OR view(tran_cmp(cmp,sttran1,agtran1))(st3, st4))
```

```
disjoint_cags: LEMMA
  (cags(tran_cmp(cmp, sttran1, agtran1))(ag2)
       AND tmap(agtran1, ags)(ag2))
  => (EXISTS ag1:
       (cags(cmp)(ag1) AND ags(ag1)))

END tcprops
```

We prove the following theorems about translated predicates:

- (*sp_tran*) *If ysp is the translation of xsp by sttran1, then stbp(ysp) is the translation of stbp(xsp).*
- (*always_sp_tran*) *If ysp is the translation of xsp by sttran1, then alwayss(ysp) is the translation of alwayss(xsp).*
- (*always_tmap*) *The translation of alwayss(xsp) by sttran1 and agtran1 equals alwayss applied to the translation of xsp by sttran1.*
- (*pimplies_pmap*) *pmap distributes over pimplies.*
- (*ap_tran*) *If yap is the translation of xap by sttran1 and agtran1, then atbp(yap) is the translation of atbp(xap).*
- (*always_ap_tran*) *If yap is the translation of xap by sttran1 and agtran1, then alwaysa(yap) is the translation of alwaysa(xap).*

These theorems allows us to "compute" translations of behavior predicates in terms of translations of state and action predicates. In other words, proofs of state and action predicates for a given component can be used in proofs of behavior predicates for translated components. In the common case in which the translated components represent composite systems and pre-translated components represents individual components of the composite, this means that behavior predicates (including temporal properties) of the composite can be proved using as a basis state and action predicates for the individual components. This allows global, temporal properties to be proved by reasoning about individual transitions of individual components.

## THEORY *tpreds*

```
tpreds[ST: NONEMPTY_TYPE, ST1: NONEMPTY_TYPE,
       AG: NONEMPTY_TYPE, AG1: NONEMPTY_TYPE]:
THEORY
  BEGIN

  IMPORTING tprops[ST, ST1, AG, AG1]

  IMPORTING ac_translators[ST1, AG1, ST, AG]

  IMPORTING preds[ST, AG]

  IMPORTING preds[ST1, AG1]

  IMPORTING unity

  xsp: VAR STATE_PRED[ST1, AG1]
```

```
ysp: VAR STATE_PRED[ST, AG]

xap: VAR ACTION_PRED[ST1, AG1]

yap: VAR ACTION_PRED[ST, AG]

xp1, xp2: VAR prop_t[ST1, AG1]

yp1, yp2: VAR prop_t[ST, AG]

xst, xst1, xst2: VAR ST1

yst, yst1, yst2: VAR ST

xag: VAR AG1

yag: VAR AG

sttran1: VAR (translator_t[ST1, ST])

agtran1: VAR (translator_t[AG1, AG])

sp_tran: THEOREM
      (FORALL yst: tmap(sttran1, xsp)(yst) IFF ysp(yst))
          IMPLIES pmap(stbp(xsp), sttran1, agtran1) = (stbp(ysp))

always_sp_tran: THEOREM
      (FORALL yst: tmap(sttran1, xsp)(yst) IFF ysp(yst))
          IMPLIES pmap(alwayss(xsp), sttran1, agtran1) = (alwayss(ysp))

always_tmap: THEOREM
  pmap(alwayss(xsp), sttran1, agtran1) = alwayss(tmap(sttran1, xsp))

pimplies_pmap: THEOREM
  pmap(pimplies(xp1, xp2), sttran1, agtran1)
    = pimplies(pmap(xp1, sttran1, agtran1), pmap(xp2, sttran1, agtran1))

ap_tran: THEOREM
      (FORALL yst1, yst2, yag:
         tr_ac[ST1, AG1, ST, AG](xap, sttran1, agtran1)(yst1, yst2, yag)
             IFF yap(yst1, yst2, yag))
          IMPLIES pmap(atbp(xap), sttran1, agtran1) = (atbp(yap))

always_ap_tran: THEOREM
      (FORALL yst1, yst2, yag:
         tr_ac[ST1, AG1, ST, AG](xap, sttran1, agtran1)(yst1, yst2, yag)
             IFF yap(yst1, yst2, yag))
          IMPLIES pmap(alwaysa(xap), sttran1, agtran1) = (alwaysa(yap))

END tpreds
```

# Section 15 Composing Two Components

We now illustrate the use of translator functions by defining an analogue of $compose$ for a pair of components that potentially have different state and agent types. We suppose that the first component has types $ST_1$ and $AG_1$, the second component has types $ST_2$ and $AG_2$, and the composition is to have types $ST$ and $AG$. We also assume that there are:

- translator functions $sttran_1$ and $sttran_2$ mapping $ST_1$ and $ST_2$ to $ST$, and
- weak translator functions $agtran_1$ and $agtran_2$ mapping $AG_1$ and $AG_2$ to $AG$.

The approach is to use $tr\_cmp$ and the translator functions to translate the two components to components having types $ST$ and $AG$. Then, the resulting components can be combined using $compose$. We use:

$$compose2(cmp_1, cmp_2, sttran_1, sttran_2, agtran_1, agtran_2)$$

to denote this pairwise composition.[12]

We also state and prove a specialization of the general composition theorem described in Section 10 to pairwise composition.

## THEORY *compose2*

```
compose2[ST: NONEMPTY_TYPE, ST1: NONEMPTY_TYPE, ST2: NONEMPTY_TYPE,
         AG: NONEMPTY_TYPE, AG1: NONEMPTY_TYPE, AG2: NONEMPTY_TYPE]: THEORY
BEGIN

  IMPORTING cmp_translators[ST1,AG1,ST,AG]
  IMPORTING cmp_translators[ST2,AG2,ST,AG]
  IMPORTING compose[ST,AG]

  cset: VAR setof[(comp_t[ST,AG])]

  cmp, cmpa, cmpb: VAR (comp_t[ST,AG])

  cmp1 : VAR (comp_t[ST1,AG1])

  cmp2 : VAR (comp_t[ST2,AG2])

  sttran1 : VAR (translator_t[ST1,ST])
  agtran1 : VAR (weak_translator_t[AG1,AG])

  sttran2 : VAR (translator_t[ST2,ST])
  agtran2 : VAR (weak_translator_t[AG2,AG])
```

[12] Actually, we define $compose2$ by defining $init$, $cags$, $guar$, ...in terms of $tmap$, $tr\_ac$, ...This gives a definition that is analogous to the definition of pairwise composition given in previous versions of this report. Then, we prove that the definition is equivalent to simply translating with $tr\_cmp$ and applying $compose$ to the result. This provides an explicit connection between the work described in previous versions of this report and the work described here.

```
make_two_set(cmpa,cmpb) : setof[(comp_t[ST,AG])] =
  (LAMBDA cmp: cmp = cmpa or cmp = cmpb)

make_two_set_tr(cmp1,cmp2,sttran1,sttran2,agtran1,agtran2) :
  setof[(comp_t[ST,AG])] =
    make_two_set(tran_cmp(cmp1,sttran1,agtran1),tran_cmp(cmp2,sttran2,agtran2))

compose_init2(cmp1, cmp2, sttran1, sttran2, agtran1, agtran2):
    setof[ST] =
  intersection(tmap(sttran1, init(cmp1)), tmap(sttran2, init(cmp2)))

compose_init2_def: THEOREM
  compose_init(make_two_set_tr(cmp1,cmp2,sttran1,sttran2,agtran1,agtran2)) =
  compose_init2(cmp1,cmp2,sttran1,sttran2,agtran1,agtran2)

compose_guar2(cmp1, cmp2, sttran1, sttran2, agtran1, agtran2):
    setof[[ST, ST, AG]] =
      union(intersection(tr_ac(guar(cmp1),sttran1,agtran1),
                         union(tr_ac(hidd(cmp2),sttran2, agtran2),
                               env_stutter(cmp2,sttran2,agtran2))),
            union(intersection(tr_ac(guar(cmp2),sttran2,agtran2),
                               union(tr_ac(hidd(cmp1),sttran1,agtran1),
                                     env_stutter(cmp1,sttran1,agtran1))),
                  intersection(tr_ac(guar(cmp1),sttran1,agtran1),
                               tr_ac(guar(cmp2),sttran2,agtran2))))

compose_guar2_def: THEOREM
  compose_guar(make_two_set_tr(cmp1,cmp2,sttran1,sttran2,agtran1,agtran2)) =
  compose_guar2(cmp1,cmp2,sttran1,sttran2,agtran1,agtran2)

compose_rely2(cmp1, cmp2, sttran1, sttran2, agtran1, agtran2):
    setof[[ST, ST, AG]] =
  intersection(union(tr_ac(rely(cmp1),sttran1,agtran1),
                     env_stutter(cmp1,sttran1,agtran1)),
               union(tr_ac(rely(cmp2), sttran2, agtran2),
                     env_stutter(cmp2,sttran2,agtran2)))

compose_rely2_def: THEOREM
  compose_rely(make_two_set_tr(cmp1,cmp2,sttran1,sttran2,agtran1,agtran2)) =
  compose_rely2(cmp1,cmp2,sttran1,sttran2,agtran1,agtran2)

compose_cags2(cmp1, cmp2, sttran1, sttran2, agtran1, agtran2):
    setof[AG] = union(tmap(agtran1, cags(cmp1)), tmap(agtran2, cags(cmp2)))

compose_cags2_def: THEOREM
  compose_cags(make_two_set_tr(cmp1,cmp2,sttran1,sttran2,agtran1,agtran2)) =
  compose_cags2(cmp1,cmp2,sttran1,sttran2,agtran1,agtran2)

compose_view2(cmp1, cmp2, sttran1, sttran2, agtran1, agtran2):
    setof[[ST, ST]] =
  intersection(vmap(sttran1, view(cmp1)), vmap(sttran2, view(cmp2)))

compose_view2_def: THEOREM
  compose_view(make_two_set_tr(cmp1,cmp2,sttran1,sttran2,agtran1,agtran2)) =
  compose_view2(cmp1,cmp2,sttran1,sttran2,agtran1,agtran2)

compose_hidd2(cmp1, cmp2, sttran1, sttran2, agtran1, agtran2):
    setof[[ST, ST,AG]] =
  intersection(union(tr_ac(hidd(cmp1),sttran1,agtran1),
                     env_stutter(cmp1,sttran1,agtran1)),
               union(tr_ac(hidd(cmp2), sttran2, agtran2),
                     env_stutter(cmp2,sttran2,agtran2)))
```

```
  compose_hidd2_def: THEOREM
    compose_hidd(make_two_set_tr(cmp1,cmp2,sttran1,sttran2,agtran1,agtran2)) =
    compose_hidd2(cmp1,cmp2,sttran1,sttran2,agtran1,agtran2)

  compose_wfar2(cmp1, cmp2,sttran1,sttran2,agtran1,agtran2): setof[TRANSITION_CLASS[ST,AG]] =
    union(tr_tcs(wfar(cmp1), sttran1,agtran1),
          tr_tcs(wfar(cmp2), sttran2,agtran2))

  compose_wfar2_def: THEOREM
    compose_wfar(make_two_set_tr(cmp1,cmp2,sttran1,sttran2,agtran1,agtran2)) =
    compose_wfar2(cmp1,cmp2,sttran1,sttran2,agtran1,agtran2)

  compose_sfar2(cmp1, cmp2,sttran1,sttran2,agtran1,agtran2): setof[TRANSITION_CLASS[ST,AG]] =
    union(tr_tcs(sfar(cmp1), sttran1,agtran1),
          tr_tcs(sfar(cmp2), sttran2,agtran2))

  compose_sfar2_def: THEOREM
    compose_sfar(make_two_set_tr(cmp1,cmp2,sttran1,sttran2,agtran1,agtran2)) =
    compose_sfar2(cmp1,cmp2,sttran1,sttran2,agtran1,agtran2)

  composable_init2(cmp1, cmp2, sttran1, sttran2, agtran1, agtran2):
      bool =
    compose_init2(cmp1, cmp2, sttran1, sttran2, agtran1, agtran2)
        /= emptyset

  composable_init2_def: THEOREM
    agreeable_start(make_two_set_tr(cmp1,cmp2,sttran1,sttran2,agtran1,agtran2)) =
    composable_init2(cmp1,cmp2,sttran1,sttran2,agtran1,agtran2)

  composable2(cmp1, cmp2, sttran1, sttran2, agtran1, agtran2): bool =
    composable_init2(cmp1, cmp2, sttran1, sttran2, agtran1, agtran2)

  composable2_def: THEOREM
    composable(make_two_set_tr(cmp1,cmp2,sttran1,sttran2,agtran1,agtran2)) =
    composable2(cmp1,cmp2,sttran1,sttran2,agtran1,agtran2)

  c: VAR (composable2)

  compose_base2(c): base_comp_t[ST, AG] =
    (# init := compose_init2(c),
       guar := compose_guar2(c),
       rely := compose_rely2(c),
       cags := compose_cags2(c),
       view := compose_view2(c),
       wfar := compose_wfar2(c),
       sfar := compose_sfar2(c),
       hidd := compose_hidd2(c) #)

  compose_base2_def: THEOREM
compose_base2(c) =
    compose_base(make_two_set_tr(c))

  compose2(c): (comp_t[ST, AG]) = compose_base2(c)

  compose2_def: THEOREM compose2(c) =
    compose(make_two_set_tr(c))

END compose2
```

## THEORY *cmp_thm2*

```
cmp_thm2[ST: NONEMPTY_TYPE, ST1: NONEMPTY_TYPE, ST2 : NONEMPTY_TYPE,
    AG: NONEMPTY_TYPE, AG1: NONEMPTY_TYPE, AG2: NONEMPTY_TYPE]: THEORY
  BEGIN

  IMPORTING compose2[ST, ST1,ST2,AG,AG1,AG2]

  IMPORTING compose_idempotent[ST,AG]

  IMPORTING cmp_thm[ST, AG]

  p: VAR prop_t[ST, AG]

  cmp1: VAR (comp_t[ST1, AG1])

  cmp2: VAR (comp_t[ST2, AG2])

  st, st1, st2: VAR ST

  ag: VAR AG

  sttran1 : VAR (translator_t[ST1,ST])
  agtran1 : VAR (translator_t[AG1,AG])

  sttran2 : VAR (translator_t[ST2,ST])
  agtran2 : VAR (translator_t[AG2,AG])

  respects_restrictions1(cmp1, cmp2, sttran1, sttran2, agtran1, agtran2):
      bool =
    (FORALL st1, st2, ag:
       member((st1,st2,ag),tr_ac(guar(cmp1), sttran1, agtran1)) AND
           not member((st1, st2, ag),tr_ac(guar(cmp2), sttran2, agtran2)) and
           member((st1, st2, ag),tr_ac(hidd(cmp2), sttran2, agtran2)) implies
             member((st1, st2, ag), tr_ac(rely(cmp2), sttran2, agtran2)))

  respects_restrictions2(cmp1, cmp2, sttran1, sttran2, agtran1, agtran2):
      bool =
    (FORALL st1, st2, ag:
       member((st1,st2,ag),tr_ac(guar(cmp2), sttran2, agtran2)) AND
           not member((st1, st2, ag),tr_ac(guar(cmp1), sttran1, agtran1)) and
           member((st1, st2, ag),tr_ac(hidd(cmp1), sttran1, agtran1)) implies
             member((st1, st2, ag), tr_ac(rely(cmp1), sttran1, agtran1)))

  respects_and_tolerates_same2: THEOREM
       respects_restrictions2(cmp1,cmp2,sttran1,sttran2,agtran1,agtran2)
        implies tolerates(singleton(tran_cmp(cmp1,sttran1,agtran1)),
                          make_two_set_tr(cmp1,cmp2,sttran1,sttran2,agtran1,
                             agtran2))

  respects_and_tolerates_same1: THEOREM
       respects_restrictions1(cmp1,cmp2,sttran1,sttran2,agtran1,agtran2)
        implies tolerates(singleton(tran_cmp(cmp2,sttran2,agtran2)),
                          make_two_set_tr(cmp1,cmp2,sttran1,sttran2,agtran1,
                             agtran2))

  compose_thm1: THEOREM
        composable2(cmp1, cmp2,sttran1,sttran2,agtran1,agtran2) AND
     respects_restrictions2(cmp1, cmp2,sttran1,sttran2,agtran1,agtran2)
            IMPLIES
          (satisfies(tran_cmp(cmp1,sttran1,agtran1), p)
    IMPLIES satisfies(compose2(cmp1, cmp2,sttran1,sttran2,agtran1,agtran2), p))

  compose_thm2: THEOREM
        composable2(cmp1, cmp2,sttran1,sttran2,agtran1,agtran2) AND
```

```
        respects_restrictions1(cmp1, cmp2,sttran1,sttran2,agtran1,agtran2)
                IMPLIES
                  (satisfies(tran_cmp(cmp2,sttran2,agtran2), p)
      IMPLIES satisfies(compose2(cmp1, cmp2,sttran1,sttran2,agtran1,agtran2), p))

  compose_thm: THEOREM
          composable2(cmp1, cmp2,sttran1,sttran2,agtran1,agtran2) AND
        respects_restrictions1(cmp1, cmp2,sttran1,sttran2,agtran1,agtran2) and
        respects_restrictions2(cmp1, cmp2,sttran1,sttran2,agtran1,agtran2)
                IMPLIES
                  ((satisfies(tran_cmp(cmp1,sttran1,agtran1), p) OR
                  satisfies(tran_cmp(cmp2,sttran2,agtran2), p))
                      IMPLIES satisfies(compose2(cmp1,
cmp2,sttran1,sttran2,agtran1,agtran2), p))

  END cmp_thm2
```

# Section 16 An Example

We now give an example of the use of this composition framework. We will specify seven components including simplified versions of the DTOS kernel and a security server as well as five components from a Cryptographic Subsystem [8] that clients may use to encrypt information either to be sent across a network or used internal to the system (e.g., written to encrypted media). An overview of the Crypto Subsystem components is provided in Section 20.

After specifying the components we will

- define a common state space for the components,
- translate the components into that common space,
- show that the translated components are composable and that they tolerate each other,
- instantiate the composition theorem for the composite, and
- perform a partial analysis of a property of the entire system.

# Section 17
# Kernel

This section provides a specification of the DTOS kernel. The specification provided here is actually of a hypothetical, simpler kernel that is similar to the DTOS kernel. Using this simpler specification here allows the study to focus on composability issues rather than being mired in the details of DTOS. In addition to completely ignoring portions of DTOS, the description given here also deviates from the behavior of DTOS in certain areas. These deviations will be indicated in footnotes throughout this section.

## 17.1 State

### 17.1.1 Primitive Entities

The primitive entities in DTOS are:

**Tasks** — environments in which threads execute; a task consists of an address space, a port name space, and a set of threads

**Threads** — active entities comprised of an instruction pointer and a local register state

**Ports** — unidirectional communication channels between tasks used to implement IPC

**Messages** — entities transmitted through ports

**Security Identifiers (SIDs)** — abstract labels attached to entities to indicate their security attributes.

**Permissions** — the permissions that are verified by the kernel before it performs operations.

**Names** — the identifiers for ports.

**Rights** — capabilities to use ports for communicating in a particular direction (i.e., sending or receiving).

**Memories** — memory objects representing shared memory

**Pages** — logical units of memory; either a unit of physical memory or provided by a memory

**Devices** — resources such as terminals and printers that can be used to transmit information between the system and its environment

The last three of these will not be considered in this model.

### 17.1.2 Kernel Shared Information

Since virtually all components will interact with the kernel in some way we make certain data types and constants globally available in the PVS specification. This includes the structure of defined kernel requests, the state information shared by the kernel with each thread in the system and the structure of information returned by the kernel in response to requests. This section defines these data types and constants.

Each kernel receives requests from threads executing on the kernel. The set $pending\_requests$ denotes the requests that threads have initiated but for which the kernel has not yet started processing. For this example, the pending requests are:

- $send\_message\_req(smth, smna, smop, smrna, smusr\_msg)$ — indicates that $smth$ has made a request to send a message to the port named by $smna$ specifying $smop$, $smrna$, and $smusr\_msg$ as, respectively, the operation id, reply port, and message
- $receive\_message\_req(rmth, rmna)$ — indicates that $rmth$ has made a request to receive a message from the port named by $rmna$
- $provide\_access\_req(pact, paop, pacav, passport, passi, paosi, parav, parp)$ — indicates that the kernel has received a request to load an access vector with:
  - $pact$ indicating the client thread,
  - $paop$ indicating the operation id of the request message,
  - $pacav$ indicating the access vector of the sender,
  - $passport$ indicating the port through which the message was received,
  - $(passi, paosi, parav)$ indicating the computation being provided , and
  - $parp$ indicating the reply port.
- $set\_ssp\_req(ssct, ssop, ssav, sssp, ssnp, ssrp)$ — indicates that the kernel has received a request to set the security server port[13] with:
  - $ssct$ indicating the client thread,
  - $ssop$ indicating the operation id of the request message,
  - $ssav$ indicating the sending access vector,
  - $sssp$ indicating the host port (to which this request must be sent),
  - $ssnp$ indicating the port to which the security server port should be set, and
  - $ssrp$ indicating the reply port.
- $get\_ssp\_req(gsct, gsop, gsav, gssp, gsrp)$ — indicates that the kernel has received a request to retrieve the security server port[14] with:
  - $gsct$ indicating the client thread,
  - $gsop$ indicating the operation id of the request message,
  - $gsav$ indicating the sending access vector,
  - $gssp$ indicating the host port (to which this request must be sent), and
  - $gsrp$ indicating the reply port.

[13] In DTOS this request is actually only one option of a more general request to set a host special port.
[14] In DTOS this request is actually only one option of a more general request to retrieve a host special port.

The user messages referred to above contain a *user_data* field indicating the data in the body of the message and a *user_rights* field indicating the sequence of name-right pairs denoting port rights to be transferred in the message.

For each component there are certain pieces of kernel state to which it has access. These pieces of data are collected in the type *KERNEL_SHARED_STATE*. Note that each non-kernel component will have its own *KERNEL_SHARED_STATE* structure. These structures will be merged when composing the components. The kernel's *KERNEL_SHARED_STATE* structure will contain all of the information in the component structures. The *KERNEL_SHARED_STATE* consists of

- *pending_requests* indicating requests that have been made to the kernel and that have not yet been processed,
- *existing_threads* indicating the existing threads,
- *received_info*(*thread*) indicating the information returned by the last message receive request invoked by the thread, and
- *thread_status*(*thread*) indicating whether the thread is currently running or waiting for a response to a kernel request (values are *thread_running* and *thread_waiting*).

The *empty_kst* is defined to be the kernel shared state that has empty sets of existing threads and pending requests and empty domains for the functions *received_info* and *thread_status*.

The value of *received_info*(*thread*) is of type *RECEIVED_INFO* which is a structure containing the following fields:[15]

- *service_port* — the port through which *thread* last received a message,
- *sending_sid* — the SID of the sender of the last message *thread* received,
- *sending_av* — the sending access vector associated with the last message *thread* received,
- *user_msg* — the data and transferred rights in the last message *thread* received,
- *op* — the operation id specified in the last message *thread* received, and
- *reply_name* — the reply port specified in the last message *thread* received.
- *ri_status* — a flag indicating whether the thread has already processed the above information; the two possible values are *ri_processed* and *ri_unprocessed*.

The defined permissions are:[16]

- Task
  - *create_task_perm* — indicates the permission to create a new task that is initiated in the standard Mach style

[15] Rather than using a separate data structure, DTOS actually writes received messages into a task's address space. We introduce the *received_info* data structure to avoid writing functions to map messages to sequences of bytes and vice-versa. Also note that tasks in DTOS are responsible for keeping track themselves of whether they have processed a message. For convenience in specifying components, we assume the presence of the *ri_status* field to indicate whether a message has already been processed.

[16] DTOS defines many more permissions than are defined here.

- $create\_task\_secure\_perm$ — indicates the permission to create a new task that is initiated in the DTOS style

- IPC
  - $xfer\_send\_perm$ — indicates the permission to transfer a send right in a message (see Section 17.2.1)
  - $xfer\_receive\_perm$ — indicates the permission to transfer a receive right in a message (see Section 17.2.1)
  - $send\_perm$ — indicates the permission to send a message (see Section 17.2.1)
  - $receive\_perm$ — indicates the permission to receive a message (see Section 17.2.2)
- Host
  - $provide\_access\_perm$ — indicates the permission to load access vectors into the kernel's access vector cache (see Section 17.2.3)
  - $set\_ss\_perm$ — indicates the permission to set the master security server port (see Section 17.2.5)
  - $get\_ss\_perm$ — indicates the permission to retrieve the master security server port (see Section 17.2.6)

---

*Editorial Note:*
In comparison to the PVS theories in many of the later sections of this report, the theories in this section are quite large. Experience has convinced us that it is generally better PVS style to use small theories. It is easier to comprehend small theories, easier to intersperse text and PVS, and PVS seems to operate more efficiently on a large collection of small theories than on a small collection of large theories.

---

## THEORY *dtos_kernel_shared_state*

```
dtos_kernel_shared_state: THEORY
  BEGIN

  SID: NONEMPTY_TYPE

  sid_witness : SID

  PERMISSION: NONEMPTY_TYPE

  create_task_perm, create_task_secure_perm: PERMISSION
  xfer_send_perm, xfer_receive_perm, send_perm, receive_perm : PERMISSION
  provide_access_perm, set_ss_perm, get_ss_perm : PERMISSION

  ACCESS_VECTOR: TYPE = setof[PERMISSION]

  DATA: NONEMPTY_TYPE

  success_data : DATA
  null_data : DATA


  TIME: NONEMPTY_TYPE
```

```
NAME: NONEMPTY_TYPE

null_name : NAME

IMPORTING finite_sequence[NAME]

NAME_SEQ : TYPE = FSEQ[NAME]


RIGHT : TYPE = {send, receive}

USER_RIGHT : TYPE = [NAME, RIGHT]

IMPORTING finite_sequence[USER_RIGHT]

USER_RIGHTS: TYPE = FSEQ[USER_RIGHT]


name_to_send_right : [NAME -> USER_RIGHT]
name_to_send_right_seq: [NAME -> USER_RIGHTS]

USER_MSG: TYPE = [# user_data: DATA, user_rights: USER_RIGHTS #]

null_user_msg: USER_MSG =
  (# user_data := null_data, user_rights := null_seq #)

OP: NONEMPTY_TYPE

op_witness : OP
provide_access_op, set_host_special_port_op, get_host_special_port_op : OP
request_access_op : OP

RI_STATUS: TYPE = {ri_unprocessed, ri_processed}

RECEIVED_INFO:
    TYPE =
      [# service_port: NAME,
         sending_sid: SID,
         sending_av: ACCESS_VECTOR,
         user_msg: USER_MSG,
         op: OP,
         reply_name: NAME,
         ri_status: RI_STATUS #]

ri_witness : RECEIVED_INFO =
      (# service_port := null_name,
         sending_sid := sid_witness,
         sending_av := emptyset[PERMISSION],
         user_msg := null_user_msg,
         op := op_witness,
         reply_name := null_name,
         ri_status := ri_processed #)

THREAD_STATUS: TYPE = {thread_waiting, thread_running}

THREAD: NONEMPTY_TYPE

th: VAR THREAD

rna, na: VAR NAME

op: VAR OP

usr_msg: VAR USER_MSG
```

```
PORT: NONEMPTY_TYPE

HOST_SPECIAL_PORT: TYPE

KERNEL_REQ: DATATYPE
  BEGIN
    send_message_req(smth : THREAD, smna : NAME, smop : OP,
      smrna : NAME, smusr_msg : USER_MSG) : send_message_req?
    receive_message_req(rmth : THREAD, rmna : NAME) : receive_message_req?
    provide_access_req(pact: THREAD, paop : OP, pacav : ACCESS_VECTOR, passport : PORT,
      passi : SID, paosi : SID, parav : ACCESS_VECTOR, parp : PORT) : provide_access_req?
    set_ssp_req(ssct : THREAD, ssop : OP, ssav : ACCESS_VECTOR,
      sssp : PORT, ssnp : PORT, ssrp : PORT) : set_ssp_req?
    get_ssp_req(gsct : THREAD, gsop : OP, gsav : ACCESS_VECTOR,
      gssp : PORT, gsrp : PORT) : get_ssp_req?
  END KERNEL_REQ


KERNEL_SHARED_STATE:
    TYPE =
      [# pending_requests: setof[KERNEL_REQ],
         existing_threads: setof[THREAD],
         received_info: [(existing_threads) -> RECEIVED_INFO],
         thread_status: [(existing_threads) -> THREAD_STATUS] #]

empty_kst: KERNEL_SHARED_STATE =
  (# existing_threads := emptyset[THREAD],
     pending_requests := emptyset[KERNEL_REQ],
     received_info :=
             (LAMBDA (x: (emptyset[THREAD])):
                ri_witness),
     thread_status :=
             (LAMBDA (x: (emptyset[THREAD])):
                thread_running)
  #)

k_threads: (nonempty?[THREAD])

END dtos_kernel_shared_state
```

When we compose components we must merge their $KERNEL\_SHARED\_STATE$ information. The predicate $kst\_mergable$ is true if the set of kernel states are not contradictory. This happens when for every pair of kernel states in the set, either

- the two kernel states do not share any threads, or
- for each shared thread, the $received\_info$ and $thread\_status$ of that thread are the same in both kernel states.

For any set of mergable states, the function $kst\_merge$ returns the merged $KERNEL\_SHARED\_STATE$. It has the following definition:

- $pending\_requests$ is the union of the $pending\_requests$ in the states.
- $existing\_threads$ is the union of the $existing\_threads$ in the states.

- $ri = received\_info(th)$ in the merged state if and only if $ri = received\_info(th)$ in one of the input states.
- $stat = thread\_status(th)$ in the merged state if and only if $stat = thread\_status(th)$ in one of the input states.

The predicate $kst\_substate$ is true of the pair $(kst_1, kst_2)$ if

- the existing threads of $kst_1$ is a subset of those in $kst_2$,
- the pending requests of $kst_1$ is a subset of those in $kst_2$, and
- each of the functions $received\_info(kst_1)$ and $thread\_status(kst_1)$ can be extended to the corresponding function in $kst_2$.

A variety or results have been demonstrated regarding the merging of kernel shared states. Key among these are

- $empty\_kst\_substate$ — The empty kernel shared state is a substate of every kernel shared state.
- $kst\_substate\_refl$ — Every kernel shared state is a substate of itself.
- $kst\_merge\_contains$ — Every kernel shared state in a set $km$ is a substate of $kst\_merge(km)$.
- $kst\_mergable\_subset$ — If $km_1$ is a mergable set of kernel shared states, then so is every subset of $km_1$.
- $kst\_mergable\_substates$ — If every kernel shared state in the set $kstset$ is a substate of a kernel shared state $kst_2$, then the set containing $kst_2$ plus all the elements of $kstset$ is mergable.
- $kst\_merge\_substates$ — If every kernel shared state in $kstset$ is a $kst\_substate$ of $kst2$, then $kst2$ is equal to $kst\_merge$ of the set containing $kst_2$ plus all the elements of $kstset$.

Since the kernel shared state of each non-kernel component will be assumed to be a substate of the kernel shared state of the kernel, these theorems will be useful in the analysis of the composite system in Section 26.

## THEORY *kst_merge*

---

*kst_merge* : **THEORY**

**BEGIN**

**IMPORTING** *dtos_kernel_shared_state*
**IMPORTING** *more_set_lemmas*

*th* : **VAR** *THREAD*

*kst* : **VAR** *KERNEL_SHARED_STATE*

*i,j* : **VAR** *nat*

*S,T* : **VAR** *setof*[*THREAD*]

```
  kst1, kst2 : VAR KERNEL_SHARED_STATE

  kstset, kstset1, kstset2 : VAR setof[KERNEL_SHARED_STATE]


kst_mergable(kstset): bool =
  (FORALL th, kst1, kst2:
        (kstset(kst1) AND kstset(kst2)
         AND existing_threads(kst1)(th)
         AND existing_threads(kst2)(th))
          => (received_info(kst2)(th) = received_info(kst1)(th)
                AND thread_status(kst2)(th) = thread_status(kst1)(th)))


  km, km1, km2 : VAR (kst_mergable)
  ri: VAR RECEIVED_INFO
  thst: VAR THREAD_STATUS

%%% Note that if a thread is shared, it must have the same status
%%% and received info in all ksts for the merge to be successful.

kst_merge(km1) : KERNEL_SHARED_STATE =
 LET all_threads : setof[THREAD]
        = { th : THREAD | EXISTS (kst : (km1))
                                   : existing_threads(kst)(th)} IN
 (# pending_requests := { kr : KERNEL_REQ
                          | EXISTS (kst : (km1)) : pending_requests(kst)(kr)},
    existing_threads := all_threads,
    received_info :=
         (LAMBDA (th : (all_threads)) :
                epsilon({ri | FORALL (kst: (km1)) :
                               existing_threads(kst)(th)
                               IMPLIES ri = received_info(kst)(th)})),
    thread_status :=
         (LAMBDA (th : (all_threads)) :
                epsilon({thst | FORALL (kst: (km1)) :
                               existing_threads(kst)(th)
                               IMPLIES thst = thread_status(kst)(th)}))
 #)



kst_substate(kst1, kst2) : bool =
  subset?(existing_threads(kst1), existing_threads(kst2))
  AND subset?(pending_requests(kst1), pending_requests(kst2))
  AND FORALL (th : (existing_threads(kst1))) :
        (received_info(kst1)(th) = received_info(kst2)(th)
        AND thread_status(kst1)(th) = thread_status(kst2)(th))

  empty_kst_substate: THEOREM
    kst_substate(empty_kst, kst2)

  kst_substate_refl: THEOREM
    kst_substate(kst1,kst1)

  kst_merge_contains: THEOREM
    km(kst) => kst_substate(kst, kst_merge(km))

  kst_mergable_disjoint_threads : THEOREM
    (FORALL (kst1, kst2 : (kstset), th) :
        existing_threads(kst1)(th) AND existing_threads(kst2)(th)
```

```
      IMPLIES kst1 = kst2)
   IMPLIES kst_mergable(kstset)

kst_mergable_add : THEOREM
  (FORALL (kst2 : (km2)) :
      kst_mergable({kst | kst = kst1 or kst = kst2}))
  IMPLIES kst_mergable(add(kst1, km2))

kst_mergable_union : THEOREM
  (FORALL (kst : (km1)) :
              kst_mergable(add(kst, km2)))
  IMPLIES kst_mergable(union(km1, km2))

kst_mergable_subset: THEOREM
  subset?(kstset, km1)
  IMPLIES kst_mergable(kstset)

kst_mergable_substates : THEOREM
  (FORALL (kst1 : (kstset)) : kst_substate(kst1, kst2))
  IMPLIES kst_mergable(add(kst2, kstset))

kst_merge_substates_existing_threads : THEOREM
  (FORALL (kst1 : (kstset)) : kst_substate(kst1, kst2))
  IMPLIES existing_threads(kst2) = existing_threads(kst_merge(add(kst2, kstset)))

kst_merge_substates_pending_requests : THEOREM
  (FORALL (kst1 : (kstset)) : kst_substate(kst1, kst2))
  IMPLIES pending_requests(kst2) = pending_requests(kst_merge(add(kst2, kstset)))

kst_merge_substates_received_info : THEOREM
  (FORALL (kst1 : (kstset)) : kst_substate(kst1, kst2))
  IMPLIES received_info(kst2) = received_info(kst_merge(add(kst2, kstset)))

kst_merge_substates_thread_status : THEOREM
  (FORALL (kst1 : (kstset)) : kst_substate(kst1, kst2))
  IMPLIES thread_status(kst2) = thread_status(kst_merge(add(kst2, kstset)))

kst_merge_substates : THEOREM
  (FORALL (kst1 : (kstset)) : kst_substate(kst1, kst2))
  IMPLIES kst2 = kst_merge(add(kst2, kstset))

END kst_merge
```

### 17.1.3 Kernel Internal State

At any given time, only certain primitive entities are present in the system. The sets $existing\_tasks$, $existing\_threads$, $existing\_ports$, $existing\_messages$ denote the entities of each class that are present in the current system state.

Each existing task has the following information associated with it:

- $task\_threads(task)$ — the collection of threads that execute within the context of $task$.
- $task\_names(task)$ — the collection of names used by the task to denote ports.
- $dead\_names(task)$ — a set of names that are dead (i.e., no longer usable). These must be disjoint from the names in $task\_names(task)$.
- $named\_port(task)$ — a function that maps each name in $task\_names(task)$ to the port denoted by the name.

- $held\_rights(task)$ — a function that maps each name in $task\_names(task)$ to the rights $task$ holds to the port named by that name in its IPC name space.
- $task\_sid(task)$ — the SID (Security ID) associated with $task$.

Each existing port has the following information associated with it:

- $port\_sid(port)$ — the SID associated with $port$
- $queue(port)$ — the message queue containing the messages that have been sent to $port$ but not yet received

The constant $null\_port$ is used to denote a value of type $PORT$ that never exists. We also use the constant $null\_name$ to denote a value of type $name$ that is never associated with a port in a task's name space.

The kernel associates the following information with each message queued at a port:[17]

- $sending\_sid(msg)$ — the SID of the task that sent $msg$
- $av(msg)$ — the access vector indicating the permission the sender of $msg$ has to the port to which $msg$ is sent
- $op(msg)$ — the operation id specified by the sender of $msg$
- $sent\_data(msg)$ — the data contained in the body of $msg$
- $sent\_rights(msg)$ — the sequence of port rights transferred in the body of $msg$; each element is a $(port, right)$ pair
- $reply\_port(msg)$ — the reply port specified by the sender of $msg$

The constant $k\_task$ is a value of type $TASK$ that is used to indicate the kernel itself is the receiver for a port.[18] The constant $k\_threads$ is used to indicate the kernel is the sender of a message. [19]

An access vector cache records allowed permissions on a SID-to-SID basis. We use $cached\_access(sid_1, sid_2)$ to denote the access vector (set of permissions), if any, cached for the pair $(sid_1, sid_2)$.[20]

---

*Editorial Note:*
The $cache\_access$ function has been modeled here as a total function on SID pairs. It should probably be a partial function to prohibit permission checks where the kernel has not obtained an access vector from the security server for the relevant SIDs. This error is irrelevant to the results of the composability study.

---

[17] DTOS also records a "receiving SID" with each message. In addition, Mach records more information about messages than is described here. Also note that DTOS message bodies are typed rather than untyped as described here. In particular, port rights transferred in the body of a message are part of the data in the body of the message rather than being recorded separately.

[18] Although the kernel is not really a separate task in Mach, we model it as being an existing task here.

[19] Although it is not consistent with the DTOS implementation, we use a constant set of threads to denote kernel agents.

[20] The DTOS cache also records information regarding the cachability of permissions and times at which permissions become invalidated.

### 17.1.4 Host Special Ports

The kernel records a collection of special ports:[21]

- $ss\_name$ — the kernel's name for a send right to the master security server port
- $host\_name$ — the kernel's name for a receive right to the port through which the kernel services host requests

---

*Editorial Note:*
The constants $ss\_name$ and $host\_name$ should really be elements of the kernel state rather than constants in the model. This error is irrelevant to the results of the composition study.

---

### 17.1.5 Summary

The kernel state consists of the data structures described above, combined in the type $K\_INTERNAL\_STATE$ and stored in the $int\_st$ field, plus its $KERNEL\_SHARED\_STATE$[22], $ext\_st$, containing all $existing\_threads$, $pending\_requests$, $thread\_status$, and $received\_info$ information for the system. The valid states are defined by $K\_STATE$. In a valid state, in addition to the constraints described above as the data structures were described, the following must hold:

- The internal and external versions of $existing\_threads$ are equal.
- Every name in a task's name space denotes a nonempty sets of rights for an existing port.
- Every existing message is in the queue of an existing port.

All the data in $K\_STATE$ is visible to the kernel.

## THEORY *k_state*

```
k_state : THEORY

  BEGIN
% =====

  IMPORTING dtos_kernel_shared_state

% PRIMITIVE ENTITIES
% ========= ========

  TASK : TYPE+
  k_task : TASK

  k_port : PORT
  null_port : PORT
  k_port_non_null_axiom : AXIOM NOT k_port = null_port
```

[21] The special ports currently supported in DTOS are the audit server, master and client security server, and host control ports.

[22] The data type $K\_EXTERNAL\_STATE$ is equivalent to $KERNEL\_SHARED\_STATE$.

```
  MESSAGE : TYPE+

% OTHER ENTITIES
% ===== ========

  host_name : NAME
  ss_name : NAME
  names_distinct_axiom : AXIOM ( TRUE
    AND NOT host_name = ss_name
    AND NOT host_name = null_name
    AND NOT null_name = ss_name
  )


  K_RIGHT : TYPE = [PORT, RIGHT]
  IMPORTING finite_sequence[K_RIGHT]
  K_RIGHTS : TYPE = FSEQ[K_RIGHT]
  null_rights : K_RIGHTS = null_seq


  IMPORTING finite_sequence[MESSAGE]
  MESSAGES : TYPE = FSEQ[MESSAGE]




% COMPOSITE ENTITIES
% ========= ========


  K_REQ : TYPE = KERNEL_REQ


% THE EXTERNAL (SHARED) STATE
% === ======== ======== =====
%
% Can be seen by other components.

  K_EXTERNAL_STATE : TYPE = KERNEL_SHARED_STATE


% THE INTERNAL STATE
% === ======== =====
%
% Cannot be changed by other components (note: this overlaps the shared
% state in existing_threads. The overlapping elements are constrained to
% be the same in K_STATE below).

  K_INTERNAL_STATE_BASE : TYPE =
    [#
      existing_tasks : setof[TASK],
      existing_threads : setof[THREAD],
      existing_ports : setof[PORT],
      existing_messages : setof[MESSAGE],
      task_threads : [(existing_tasks) -> setof[(existing_threads)]],
      task_names : [(existing_tasks) -> setof[NAME]],
      dead_names : [(existing_tasks) -> setof[NAME]],
      named_port : [tk : (existing_tasks) -> [(task_names(tk)) -> PORT]],
      held_rights : [tk : (existing_tasks) -> [(task_names(tk)) -> setof[RIGHT]]],
      task_sid : [(existing_tasks) -> SID],
      port_sid : [(existing_ports) -> SID],
      cached_access : [SID, SID -> ACCESS_VECTOR],
      queue : [(existing_ports) -> MESSAGES],
```

```
      sending_sid : [(existing_messages) -> SID],
      av : [(existing_messages) -> ACCESS_VECTOR],
      op : [(existing_messages) -> OP],
      sent_data : [(existing_messages) -> DATA],
      sent_rights : [(existing_messages) -> K_RIGHTS],
      reply_port : [(existing_messages) -> PORT]
    #]

  K_INTERNAL_STATE(base : K_INTERNAL_STATE_BASE) : bool = ( TRUE
    AND existing_tasks(base)(k_task)
    AND task_names(base)(k_task)(host_name)
    AND existing_ports(base)(named_port(base)(k_task)(host_name))
    AND held_rights(base)(k_task)(host_name)(receive)
    AND task_names(base)(k_task)(ss_name)
    AND existing_ports(base)(named_port(base)(k_task)(ss_name))
    AND held_rights(base)(k_task)(ss_name)(send)
    AND k_threads = task_threads(base)(k_task)
    AND (FORALL (th : (existing_threads(base))) :
      EXISTS (tk : (existing_tasks(base))) : task_threads(base)(tk)(th))
    AND NOT existing_ports(base)(null_port)
    AND (FORALL (tk : (existing_tasks(base))) :
                  NOT task_names(base)(tk)(null_name))
    AND (FORALL (tk : (existing_tasks(base))) :
                  disjoint?(task_names(base)(tk), dead_names(base)(tk)))
    AND (FORALL (tk : (existing_tasks(base)), nm : (task_names(base)(tk))) :
                       existing_ports(base)(named_port(base)(tk)(nm))
                         AND nonempty?(held_rights(base)(tk)(nm)))
    AND (FORALL (msg : (existing_messages(base))) :
             EXISTS (p : (existing_ports(base))),
                    (n : nat | n > 0 AND n <= size(queue(base)(p))) :
                       elem(queue(base)(p))(n) = msg)
  )

% THE KERNEL STATE
% === ====== =====

  K_STATE_BASE : TYPE =
    [#
      int_st : (K_INTERNAL_STATE),
      ext_st : K_EXTERNAL_STATE
    #]

  K_STATE(base : K_STATE_BASE) : bool =
    existing_threads(int_st(base)) = existing_threads(ext_st(base))

  st1, st2: VAR (K_STATE)

  k_view(st1, st2) : bool =
    st1 = st2


  END k_state
% === =======
```

The theory $k\_state\_witness$ exhibits a state that satisfies the requirements on $K\_STATE$.

*Editorial Note:*
While $k\_state\_witness$ satisfies the requirements of $K\_STATE$, it does not satisfy all the requirements that we would intuitively place on a kernel state. For example, $ss\_name$ and $host\_name$ both map to the same port. It would be better if this were not the case. This is irrelevant for the results of the

composability study.

# THEORY *k_state_witness*

```
k_state_witness: THEORY

BEGIN

  IMPORTING k_state

  k_external_state_witness : K_EXTERNAL_STATE =
    (#
      pending_requests := emptyset[K_REQ],
      existing_threads := k_threads,
      received_info := (LAMBDA (th : (k_threads)) : ri_witness),
      thread_status := (LAMBDA (th : (k_threads)) : thread_running)
    #)


  k_internal_state_witness : (K_INTERNAL_STATE) =
    (#
      existing_tasks := {tk : TASK | tk = k_task},
      existing_threads := k_threads,
      existing_ports := {p : PORT | p = k_port },
      existing_messages := emptyset[MESSAGE],
      task_threads := (LAMBDA (tk : TASK | tk = k_task) : k_threads),
      task_names := (LAMBDA (tk : TASK | tk = k_task) :
                              {nm : NAME | nm = host_name OR nm = ss_name}),
      dead_names := (LAMBDA (tk : TASK | tk = k_task) : emptyset[NAME]),
      named_port := (LAMBDA (tk : TASK | tk = k_task) :
        (LAMBDA (nm : NAME | nm = host_name OR nm = ss_name) : k_port)
      ),
      held_rights := (LAMBDA (tk : TASK | tk = k_task) :
        (LAMBDA (nm : NAME | nm = host_name OR nm = ss_name) :
                  {r : RIGHT | r=send OR r=receive})
      ),
      task_sid := (LAMBDA (tk : TASK | tk = k_task) : sid_witness),
      port_sid := (LAMBDA (p : PORT | p = k_port) : sid_witness),
      cached_access := (LAMBDA (ssi : SID, osi : SID) : emptyset[PERMISSION]),
      queue := (LAMBDA (p : PORT | p = k_port) : null_seq[MESSAGE]),
      sending_sid := (LAMBDA (msg : (emptyset[MESSAGE])) : sid_witness),
      av := (LAMBDA (msg : (emptyset[MESSAGE])) : emptyset[PERMISSION]),
      op := (LAMBDA (msg : (emptyset[MESSAGE])) : op_witness),
      sent_data := (LAMBDA (msg : (emptyset[MESSAGE])) : null_data),
      sent_rights := (LAMBDA (msg : (emptyset[MESSAGE])) : null_rights),
      reply_port := (LAMBDA (msg : (emptyset[MESSAGE])) : null_port)
    #)

  k_internal_state_witness_prop : THEOREM
    EXISTS (st : (K_INTERNAL_STATE)) : TRUE


  k_state_witness : (K_STATE) =
    (#
      int_st := k_internal_state_witness,
      ext_st := k_external_state_witness
    #)

  k_state_witness_prop : THEOREM
```

**EXISTS** ($s$ : ($K\_STATE$)) : **TRUE**

**END** *k_state_witness* 60

## 17.2 Operations

This section describes the subset of kernel operations that are relevant to this example.

*Editorial Note:*
This section currently describes only successful processing of requests.

We first define several utility functions.[23] The first argument of each of these is the $K\_INTERNAL\_STATE$ that is used in determining the return value of the function.

- $name\_to\_port(st, name, right, task)$ — If, in the name space of $task$, $name$ denotes right $right$ to an existing port $p$, and the access vector for the SID of $task$ and the SID of $p$ contains the appropriate transfer right permission, then the value is $p$. Otherwise, the value is $null\_port$.
- $user\_to\_kernel(st, u\_rt\_seq, task)$ — models the kernel processing that converts a sequence of user rights into a sequence of kernel rights. At the point where this is called we have already checked that the sender holds at least one right for each name in the sequence.
- $kernel\_to\_user(ist, task, k\_rights)$ — models the kernel's conversion of kernel rights (internal port references) to user rights (local name references) with respect to the name space of $task$. The conditions imposed on this conversion (i.e., uniqueness of names in a name space) are given as an axiom following the declaration.

In addition to the above utility functions, the following conversion functions are defined independent of the system state:

- $data\_to\_sid\_sid\_av$ — models the interpretation of user specified data as a triple $(sid_1, sid_2, access\_vector)$.
- $sid\_sid\_to\_data$ — models the representation of a pair of SIDs by message data. (This is used when the kernel sends a message to the security server requesting an access vector.)
- $op\_to\_reply\_op$ — models the relationship between an operation ID, and an ID that is used to represent replies to that operation.

[23] A frequent construct in the PVS specifications is a long list of conjuncts or disjuncts. In this section the following convention has been used for formatting such lists:

- Long conjunctions are introduced by an open parenthesis followed by the key word **TRUE**.
- Each conjunct appears on a line by itself, introduced by the key word **AND**, and indented two spaces from the introductory line.
- Long disjunctions are introduced by an open parenthesis followed by the key word **FALSE**.
- Each disjunct appears on a line by itself, introduced by the key word **OR**, and indented two spaces from the introductory line.

---

*Editorial Note:*
The addition of the axiom *kernel_to_user_axiom* introduces the question of soundness of the kernel specification. We have not attempted to deal with this in any way in this report.

---

## THEORY *k_utilities*

```
k_utilities : THEORY

  BEGIN
% =====


% IMPORTS
% =======

  IMPORTING k_state


% UTILITIES
% =========

% name_to_port converts task's name into a port reference.  If the reference is not
% given by a valid name in task's name space or if the port does not exist the
% return is null_port.

  name_to_port(
        st       : (K_INTERNAL_STATE),
        name     : NAME,
        right    : RIGHT,
        task     : (existing_tasks(st))
  )              : PORT =
    IF (task_names(st)(task)(name) AND existing_ports(st)(named_port(st)(task)(name)))
    THEN
      LET
        port : PORT = named_port(st)(task)(name),
        av : ACCESS_VECTOR = cached_access(st)(task_sid(st)(task), port_sid(st)(port)),
        hr : setof[RIGHT] = held_rights(st)(task)(name)
      IN
      IF FALSE
        OR (right = receive AND hr(right) AND av(xfer_receive_perm))
        OR (right = send AND av(xfer_send_perm))
      THEN port
      ELSE null_port
      ENDIF
    ELSE null_port
    ENDIF

% user_to_kernel models the kernel processing which converts a user right
% sequence into a kernel right sequence.  At the point where this is called
% we have already checked that the sender holds at least one right for
% each name in the sequence.

  user_to_kernel(
        st              : (K_INTERNAL_STATE),     % The initial internal state
        u_rt_seq        : USER_RIGHTS,            % The user right sequence to be converted
        task            : (existing_tasks(st))    % The owning task of the sending thread
  )                     : K_RIGHTS =
    (#
```

```
      size := size(u_rt_seq),
      elem := (LAMBDA (x : nat | x > 0 AND x <= size(u_rt_seq)) :
        (
          name_to_port(st, proj_1(elem(u_rt_seq)(x)), proj_2(elem(u_rt_seq)(x)), task),
          proj_2(elem(u_rt_seq)(x))
        )
      )
    #)

% kernel_to_user is an unspecified function that models the kernel's
% conversion of kernel rights (internal port references) to user
% rights (local name references). The conditions imposed on this
% conversion (i.e., uniqueness of names in a namespace) are given
% as an axiom following the declaration.

  kernel_to_user(
        ist              : (K_INTERNAL_STATE),
        task             : (existing_tasks(ist)),
        k_rights         : K_RIGHTS
  )                      : USER_RIGHTS

  kernel_to_user_axiom : AXIOM
    FORALL (
      ist         : (K_INTERNAL_STATE),
      task        : (existing_tasks(ist)),
      u_rts       : USER_RIGHTS,
      k_rts       : K_RIGHTS | u_rts = kernel_to_user(ist, task, k_rts)) :
    ( TRUE
      AND size(k_rts) = size(u_rts)
      AND (FORALL (i1 : posnat, i2 : posnat | i1 <= size(k_rts) AND i2 <= size(k_rts)) :
        proj_1(elem(k_rts)(i1)) = proj_1(elem(k_rts)(i2))
        IFF
        proj_1(elem(u_rts)(i1)) = proj_1(elem(u_rts)(i2))
      )
      AND (FORALL (i : posnat | i <= size(k_rts)) :
        LET
          pt : PORT = proj_1(elem(k_rts)(i)),
          nm : NAME = proj_1(elem(u_rts)(i)),
          ps : setof[PORT] = {p : PORT |
                EXISTS (x : (task_names(ist)(task))) : p = named_port(ist)(task)(x) }
        IN TRUE
          AND NOT dead_names(ist)(task)(nm)
          AND ps(pt) IMPLIES (task_names(ist)(task)(nm) AND pt = named_port(ist)(task)(nm))
          AND task_names(ist)(task)(nm) IMPLIES (ps(pt) AND existing_ports(ist)(pt))
      )
    )

% This unspecified function models the (black box) conversion of user
% specified data into certain request parameters.

  data_to_sid_sid_av : [DATA -> [SID, SID, ACCESS_VECTOR]]

% This models the conversion of a sid pair to message data (used when the kernel sends
% a message to the security server requesting an access vector.)

  sid_sid_to_data : [SID, SID -> DATA]

% This unspecified function models the conversion of an op id into the corresponding
% id for the reply message.

  op_to_reply_op : [OP -> OP]

  END k_utilities
% === ===========
```

### 17.2.1 Send Message

The result of a task sending a message to a port is that the message is added to the sequence of messages queued at the port.

The sender of the message may transfer port rights to the receiver of the message by inserting the rights in the message. The sender may transfer a send right for any port to which it holds either a send or a receive right. The effect of transferring a send right is to provide the receiver a copy of the right while leaving a copy of the right in the sender's IPC name space. To transfer a receive right for a port, the sender must hold the receive right. The effect of transferring a receive right is to provide the receiver a copy of the right while removing the copy from the sender's IPC name space.[24]

Theory $k\_send\_message$ describes the changes made to IPC name spaces when $task$ sends $user\_msg?$. No changes are made to IPC name spaces for other tasks. The only changes made to $task$'s IPC name space are when $user\_msg?$ contains a receive right. Then, the receive right needs to be removed from $held\_rights$. If the task did not also hold a send right, then the name must be removed from the domain of $named\_port$.

The kernel is responsible for translating the port names specified in the body of the message into ports. While doing so, it is also responsible for checking that the sender has permission to transfer the right. If the name does not specify an existing port or the sender does not have permission to transfer the right, then the kernel maps the name to the $null\_port$.[25] If a task or port does not exist, then any access computation required for it returns a null access vector.

After processing a request for a thread in the waiting state, the kernel returns the thread to the running state. To enqueue a message at a port, the kernel must record the information associated with the message and add the message to the queue associated with the port.[26]

In the theory $k\_send\_message$

- $ksm\_interp\_request$ verifies that the request being processed is a send message request and extracts the information from the $K\_REQ$ structure.
- $ksm\_task\_thread$ checks the existence of the thread, task and port involved in the request and determines that the receiver for the port is not the kernel.
- $ksm\_sids$ checks that the task has permission to send to the port.
- $ksm\_name\_spaces$ checks that the cache contains an access vector for each right being transferred and updates the name space of the sending task.
- $ksm\_message$ creates the kernel message and adds it to the queue of the destination port.
- $k\_send\_message$ calls the above functions to model the full processing of the request.

[24] This ignores many details of how rights are transferred in DTOS. For example, send-once rights are not addressed and no facility is provided for the sender to specify a send right should be moved from its IPC name space rather than copied.

[25] As specified here, the kernel's access vector cache must contain information sufficient to check whether all of the specified rights can be transferred. Due to the possibility of a cache entry being invalidated in the middle of the processing, this is not how DTOS actually works.

[26] While specifying a reply port in Mach results in a send right for the reply port being transferred to the receiver, the model described here requires the reply port to be explicitly added to the list of rights transferred in the message.

# THEORY *k_send_message*

```
k_send_message   : THEORY

  BEGIN
% =====


% IMPORTS
% =======

  IMPORTING k_state
  IMPORTING k_utilities


   ksm_interp_request(
    est1, est2     : K_EXTERNAL_STATE,      % The externally visible components
    kreq           : K_REQ,                 % The kernel request being processed
    thread         : THREAD,                % The client thread
    name           : NAME,                  % Where thread is sending the message
    reply_name     : NAME,                  % Where to send reply message
    op             : OP,                    % NA (applies to k_kernel_request)
    usr_msg        : USER_MSG               % The rights and data being sent
        ): bool =
    % In this transition we process an old request without generating
    % a new request...
    NOT pending_requests(est2)(kreq)
    AND pending_requests(est1) = add(kreq, pending_requests(est2))
    % and its a request to send a message...
    AND send_message_req?(kreq)
    % with these particular parameters:
    AND thread = smth(kreq)
    AND name = smna(kreq)
    AND op = smop(kreq)
    AND reply_name = smrna(kreq)
    AND usr_msg = smusr_msg(kreq)


   ksm_task_thread(
    ist1, ist2     : (K_INTERNAL_STATE),    % The internal state components
    est1, est2     : K_EXTERNAL_STATE,      % The externally visible components
    thread         : THREAD,                % The client thread
    task           : TASK,                  % Thread's owning task
    name           : NAME,                  % Where thread is sending the message
    port           : PORT                   % The port refered to by name

        ): bool = TRUE

    %% Avoid generation of too many type check conditions in PVS
    AND (FORALL (x1: (existing_tasks(ist1)), y1: (existing_tasks(ist2))):
            existing_tasks(ist1)(y1) AND existing_tasks(ist2)(x1))
    AND (FORALL (x1: (existing_threads(ist1)), y1: (existing_threads(ist2))):
            existing_threads(ist1)(y1) AND existing_threads(ist2)(x1))
    AND (FORALL (x1: (existing_threads(est1)), y1: (existing_threads(est2))):
            existing_threads(est1)(y1) AND existing_threads(est2)(x1))
    AND existing_tasks(ist1)(k_task)
    AND existing_tasks(ist1)(task)
    AND existing_tasks(ist2)(task)

    % The thread exists...
    AND existing_threads(est1)(thread)
    AND existing_threads(est2)(thread)
```

```
AND existing_threads(ist1)(thread)
AND existing_threads(ist2)(thread)
AND existing_threads(est2) = existing_threads(est1)
% and had been waiting but now is running...
AND thread_status(est1)(thread) = thread_waiting
AND thread_status(est2) = thread_status(est1) WITH [ (thread) := thread_running ]
% and thread belongs to an existing task...
AND task_threads(ist1)(task)(thread)
AND task_threads(ist2)(task)(thread)
AND existing_tasks(ist2) = existing_tasks(ist1)
AND task_threads(ist2) = task_threads(ist1)
% and name is in tasks name space...
AND task_names(ist1)(task)(name)
% and refers to an existing port...
AND port = named_port(ist1)(task)(name)
AND existing_ports(ist1)(port)
AND existing_ports(ist2)(port)
AND existing_ports(ist2) = existing_ports(ist1)
% and the receiver for port is not the kernel.
AND NOT (EXISTS (nm : (task_names(ist1)(k_task))) : TRUE
  AND named_port(ist1)(k_task)(nm) = port
   AND held_rights(ist1)(k_task)(nm)(receive)
)


ksm_sids(
 ist1, ist2      : (K_INTERNAL_STATE),     % The internal state components
 task            : TASK,                   % Thread's owning task
 port            : PORT,                   % The port refered to by name
 sending_av      : ACCESS_VECTOR           % The av associated with (task, port)

     ): bool = TRUE

 %% Avoid generation of too many type check conditions in PVS
 AND (FORALL (x1: (existing_tasks(ist1)), y1: (existing_tasks(ist2))):
       existing_tasks(ist1)(y1) AND existing_tasks(ist2)(x1))
 AND (FORALL (x1: (existing_ports(ist1)), y1: (existing_ports(ist2))):
       existing_ports(ist1)(y1) AND existing_ports(ist2)(x1))
 AND existing_tasks(ist1)(task)
 AND existing_ports(ist1)(port)

 % Nobody changes the SID assignments...
 AND task_sid(ist2) = task_sid(ist1)
 AND port_sid(ist2) = port_sid(ist1)
 % so the sending access vector is
 AND sending_av = cached_access(ist1)(task_sid(ist1)(task), port_sid(ist1)(port))
 AND cached_access(ist2) = cached_access(ist1)
 % and it contains permission to send.
 AND sending_av(send_perm)


ksm_name_spaces(
 ist1, ist2            : (K_INTERNAL_STATE),     % The internal state components
 usr_msg               : USER_MSG,               % The rights and data being sent
 task                  : TASK,                   % Thread's owning task
 rt_seq                : USER_RIGHTS,            % The sequence of rights being sent
 xfer_receive_names    : setof[NAME],            % Receive rights being sent
 no_send_names         : setof[NAME]             % Rights that task looses
     ): bool = TRUE

 %% Avoid generation of too many type check conditions in PVS
 AND (FORALL (x1: (existing_tasks(ist1)), y1: (existing_tasks(ist2))):
       existing_tasks(ist1)(y1) AND existing_tasks(ist2)(x1))
 AND existing_tasks(ist1)(task)
```

```
AND existing_tasks(ist2)(task)
AND (FORALL (nm: (task_names(ist2)(task))): task_names(ist1)(task)(nm))


% Task is using names from his name space and the cache contains
% av's for any live rights being sent in the message:
AND rt_seq = user_rights(usr_msg)
AND (FORALL (n : nat | n > 0 AND n <= size(rt_seq)) : TRUE
  AND task_names(ist1)(task)(proj_1(elem(rt_seq)(n)))
  AND existing_ports(ist1)(named_port(ist1)(task)(proj_1(elem(rt_seq)(n)))) IMPLIES
    LET
      xname : NAME = proj_1(elem(rt_seq)(n)),
      xport : PORT = named_port(ist1)(task)(xname),
      psid : SID = port_sid(ist1)(xport),
      tsid : SID = task_sid(ist1)(task)
    IN
      nonempty?(cached_access(ist1)(tsid, psid))
)

% Name spaces have been updated.
% In particular, name spaces other than task's are unchanged...
AND (FORALL (x : (existing_tasks(ist2))) :
  (x = task OR (TRUE
    AND existing_tasks(ist1)(x)
    AND (FORALL (x1: (task_names(ist1)(x)), y1: (task_names(ist2)(x))):
          task_names(ist1)(x)(y1) AND task_names(ist2)(x)(x1))
    AND named_port(ist2)(x) = named_port(ist1)(x)
    AND held_rights(ist2)(x) = held_rights(ist1)(x))
  ))

% nobody's dead name set changes...
AND dead_names(ist2) = dead_names(ist1)
% some of task's names have their receive rights removed...
AND xfer_receive_names =
  { nm : (task_names(ist1)(task)) |
    EXISTS (i : nat | i > 0 AND i <= size(rt_seq)) :
              elem(rt_seq)(i) = ( nm, receive ) }
% some names are removed from task's name space...
AND no_send_names =
  { nm : NAME | xfer_receive_names(nm) AND NOT held_rights(ist1)(task)(nm)(send) }
AND task_names(ist2)(task) = difference(task_names(ist1)(task), no_send_names)
% (remove the receive rights)...
AND (FORALL (nm : (task_names(ist2)(task))) :
  task_names(ist1)(task)(nm)
AND held_rights(ist2)(task)(nm) = remove(receive, held_rights(ist1)(task)(nm)))
AND named_port(ist2)(task) = (LAMBDA (nm : (task_names(ist2)(task))) :
                                  named_port(ist1)(task)(nm))
AND held_rights(ist2)(task) = (LAMBDA (nm : (task_names(ist2)(task))) :
                                  held_rights(ist1)(task)(nm))



ksm_message(
 ist1, ist2      : (K_INTERNAL_STATE),    % The internal state components
 reply_name      : NAME,                  % Where to send reply message
 op              : OP,                    % NA (applies to k_kernel_request)
 usr_msg         : USER_MSG,              % The rights and data being sent
 task            : TASK,                  % Thread's owning task
 port            : PORT,                  % The port refered to by name
 sending_av      : ACCESS_VECTOR,         % The av associated with (task, port)
 rt_seq          : USER_RIGHTS,           % The sequence of rights being sent
 msg             : MESSAGE                % The internal representation of the message
```

```
        ): bool = TRUE

    %% Avoid generation of too many type check conditions in PVS
    AND existing_tasks(ist1)(task)
    AND existing_messages(ist2)(msg)
    AND existing_ports(ist1)(port)
    AND existing_ports(ist2)(port)


    % The kernel enqueues a new message.
    % The set of existing messages grows...
    AND NOT existing_messages(ist1)(msg)
    AND existing_messages(ist2) = add(msg, existing_messages(ist1))
    % the msg gets added to port's queue...
    AND queue(ist2) = queue(ist1) WITH [port := tack_on(msg, queue(ist1)(port))]
    % the sending sid gets recorded...
    AND sending_sid(ist2) = sending_sid(ist1) WITH [msg := task_sid(ist1)(task)]

    % the access vector gets recorded...
    AND av(ist2) = av(ist1) WITH [msg := sending_av]

    % the operation gets recorded...
    AND op(ist2) = op(ist1) WITH [msg := op]

    % the data gets recorded...
    AND sent_data(ist2) = sent_data(ist1) WITH [msg := user_data(usr_msg)]

    % the reply port is determined from the reply name specified...
    AND reply_port(ist2) = (LAMBDA (x : (existing_messages(ist2))) :
      IF existing_messages(ist1)(x) THEN
        reply_port(ist1)(x)
      ELSIF task_names(ist1)(task)(reply_name) THEN
        named_port(ist1)(task)(reply_name)
      ELSE
        null_port
      ENDIF
    )
    % the user rights are converted to kernel rights and recorded.
    AND sent_rights(ist2) = sent_rights(ist1)
                 WITH [msg := user_to_kernel(ist1, rt_seq, task)]


% THE k_send_message REQUEST
% === ============== =======
%
% k_send_message describes a transition in which a client has requested
% to send a message to a port for which the kernel is not the receiver.

  k_send_message(
    st1          : (K_STATE),            % The initial state of the transition
    st2          : (K_STATE),            % The final state of the transition
    ag           : (k_threads)           % The mediating agent
  )              : bool =

  EXISTS (
    ist1, ist2   : (K_INTERNAL_STATE),   % The internal state components
    est1, est2   : K_EXTERNAL_STATE,     % The externally visible components
    kreq         : K_REQ,                % The kernel request being processed
    thread       : THREAD,               % The client thread
    name         : NAME,                 % Where thread is sending the message
    reply_name   : NAME,                 % Where to send reply message
    op           : OP,                   % NA (applies to k_kernel_request)
    usr_msg      : USER_MSG,             % The rights and data being sent
    task         : TASK,                 % Thread's owning task
```

```
    port            : PORT,                 % The port refered to by name
    sending_av      : ACCESS_VECTOR,        % The av associated with (task, port)
    rt_seq          : USER_RIGHTS,          % The sequence of rights being sent
    xfer_receive_names       : setof[NAME], % Receive rights being sent
    no_send_names            : setof[NAME], % Rights that task looses
    msg             : MESSAGE               % The internal representation of the message
  )                 : ( TRUE

    % Establish the state variables.
    AND ist1 = int_st(st1)
    AND ist2 = int_st(st2)
    AND est1 = ext_st(st1)
    AND est2 = ext_st(st2)

    %% Avoid generation of too many type check conditions in PVS
    AND (FORALL (x1: (existing_threads(est1)), y1: (existing_threads(est2))):
          existing_threads(est1)(y1) AND existing_threads(est2)(x1))

    AND ksm_interp_request(est1, est2, kreq, thread, name, reply_name, op, usr_msg)


    AND ksm_task_thread(ist1, ist2, est1, est2, thread, task, name, port)


   AND ksm_sids(ist1, ist2, task, port, sending_av)

   AND ksm_name_spaces(ist1, ist2, usr_msg, task, rt_seq,
                            xfer_receive_names, no_send_names)



   AND ksm_message(ist1, ist2, reply_name, op, usr_msg, task, port,
                            sending_av, rt_seq, msg)

    % The components of state not mentioned above remain unchanged:
    AND received_info(est2) = received_info(est1)
  )


  END k_send_message
% === =============
```

### 17.2.2 Receive Message

The result of a task receiving a message from a port is that the message is removed from the sequence of messages queued at the port. The kernel is responsible for determining names in the receiver's IPC name space for each of the port rights contained in the message. Any port for which the receiver already had a name is mapped to the existing name. New names are assigned to ports new to the receiver and to ports that no longer exist.[27] The kernel adds each of the received rights to the receiver's IPC name space by adding the received rights for existing ports to $held\_rights$ and the remaining names to $dead\_names$. To dequeue a message, the kernel must delete the information it has recorded for the message and remove the message from the queue. If the kernel is not the receiver of the message, then no new pending requests are generated by the receipt of the message and the returned information is recorded for the receiver in $received\_info$.

[27] As with sending messages, many details of DTOS are ignored here. Of particular interest is the omission of permission checks on whether the receiver may hold a received right.

# THEORY *k_receive_message*

```
k_receive_message : THEORY

  BEGIN
% =====


% IMPORTS
% =======

  IMPORTING k_state
  IMPORTING k_utilities


% THE k_receive_message REQUEST
% === ================= =======
%
% k_receive_message describes a transition where a client requests to receive
% a message on a non-kernel port.

% utility

  krm_names_and_rights(
        task            : TASK,
        ist1, ist2      : (K_INTERNAL_STATE),   % The internal state components
        u_rights        : USER_RIGHTS,          % The sequence of rights being sent
        k_rights        : K_RIGHTS              % The kernel version of u_rights
        ) : bool = (TRUE

    AND existing_tasks(ist1)(task)
    AND existing_tasks(ist2)(task)
    AND size(k_rights) = size(u_rights)

    % Other name spaces do not change...
    AND (FORALL (x : (existing_tasks(ist1))) :
      x = task OR (TRUE
        AND existing_tasks(ist2)(x)
        AND task_names(ist2)(x) = task_names(ist1)(x)
        AND named_port(ist2)(x) = named_port(ist1)(x)
        AND held_rights(ist2)(x) = held_rights(ist1)(x)
        AND dead_names(ist2)(x) = dead_names(ist1)(x))
      )

    % But task's name space does change if rights were sent.
    % In particular, task_names gains a name for each live port right sent...
    AND (FORALL (nm : NAME) :
      task_names(ist2)(task)(nm)
      IFF
      ( FALSE
        OR task_names(ist1)(task)(nm)
        OR EXISTS (i : nat | i > 0 AND i <= size(k_rights)) : (TRUE
          AND proj_1(elem(u_rights)(i)) = nm
          AND existing_ports(ist2)(proj_1(elem(k_rights)(i))))
      ))
    % and dead_names gains a name for each dead port right sent...
    AND (FORALL (nm : NAME) :
      dead_names(ist2)(task)(nm)
      IFF
      ( FALSE
        OR dead_names(ist1)(task)(nm)
        OR EXISTS (i : nat | i > 0 AND i <= size (k_rights)) : (TRUE
```

```
          AND proj_1(elem(u_rights)(i)) = nm
          AND NOT existing_ports(ist2)(proj_1(elem(k_rights)(i))))
    ))
  % and the name/port correspondence grows...
  AND (FORALL (nm : (task_names(ist2)(task)), pt : PORT) :
    named_port(ist2)(task)(nm) = pt
    IFF
    ( FALSE
      OR (task_names(ist1)(task)(nm) AND named_port(ist1)(task)(nm) = pt)
      OR EXISTS (i : nat | i > 0 AND i <= size(k_rights)) : (TRUE
          AND proj_1(elem(u_rights)(i)) = nm
          AND proj_1(elem(k_rights)(i)) = pt)
    ))
  % and the held_rights for task grows.
  AND (FORALL (nm : (task_names(ist2)(task)), rt : RIGHT) :
    held_rights(ist2)(task)(nm)(rt)
    IFF
    ( FALSE
      OR (task_names(ist1)(task)(nm) AND held_rights(ist1)(task)(nm)(rt))
      OR EXISTS (i : nat | i > 0 AND i <= size(u_rights)) : (TRUE
          AND proj_1(elem(u_rights)(i)) = nm
          AND proj_2(elem(u_rights)(i)) = rt)
    ))
)

k_receive_message(
  st1          : (K_STATE),          % The initial state of the transition
  st2          : (K_STATE),          % The final state of the transition
  ag           : (k_threads)         % The mediating agent
)              : bool =

EXISTS (
  ist1, ist2   : (K_INTERNAL_STATE), % The internal state components
  est1, est2   : K_EXTERNAL_STATE,   % The externally visible components
  kreq         : K_REQ,              % The kernel request being processed
  thread       : THREAD,             % The client thread
  name         : NAME,               % Where thread is receiving the message
  task         : TASK,               % Thread's owning task
  port         : PORT,               % The port refered to by name
  receiving_av: ACCESS_VECTOR,       % The av associated with (task, port)
  u_rights     : USER_RIGHTS,        % The sequence of rights being sent
  k_rights     : K_RIGHTS,           % The kernel version of u_rights
  new_info     : RECEIVED_INFO,      % The message content being received
  msg          : MESSAGE             % The internal representation of the message
)              : ( TRUE

  % Establish some variables.
  AND ist1 = int_st(st1)
  AND ist2 = int_st(st2)
  AND est1 = ext_st(st1)
  AND est2 = ext_st(st2)

  %% Avoid generation of too many type check conditions in PVS
  AND (FORALL (x1: (existing_tasks(ist1)), y1: (existing_tasks(ist2))):
        existing_tasks(ist1)(y1) AND existing_tasks(ist2)(x1))
  AND (FORALL (x1: (existing_threads(ist1)), y1: (existing_threads(ist2))):
        existing_threads(ist1)(y1) AND existing_threads(ist2)(x1))
  AND (FORALL (x1: (existing_ports(ist1)), y1: (existing_ports(ist2))):
        existing_ports(ist1)(y1) AND existing_ports(ist2)(x1))

  % In this transformation we process an old kernel request without
  % generating a new request...
  AND NOT pending_requests(est2)(kreq)
  AND pending_requests(est1) = add(kreq, pending_requests(est2))
```

```
% and its a request to receive a message...
AND receive_message_req?(kreq)
% with these particular parameters:
AND thread = rmth(kreq)
AND name = rmna(kreq)

% The thread exists...
AND existing_threads(est1)(thread)
AND existing_threads(est2)(thread)
AND existing_threads(ist1)(thread)
AND existing_threads(est2) = existing_threads(est1)
% and had been waiting but now is running...
AND thread_status(est1)(thread) = thread_waiting
AND thread_status(est2) = thread_status(est1) WITH [ (thread) := thread_running ]
% and thread belongs to an existing task...
AND existing_tasks(ist1)(task)
AND existing_tasks(ist2)(task)
AND task_threads(ist1)(task)(thread)
AND existing_tasks(ist2) = existing_tasks(ist1)
AND task_threads(ist2) = task_threads(ist1)
% and name is in task's name space...
AND task_names(ist1)(task)(name)
% and refers to an existing port...
AND port = named_port(ist1)(task)(name)
AND existing_ports(ist1)(port)
AND existing_ports(ist2)(port)
AND existing_ports(ist2) = existing_ports(ist1)
% and the kernel is not the receiver for port.
AND existing_tasks(ist1)(k_task)
AND NOT (EXISTS (nm : (task_names(ist1)(k_task))) : ( TRUE
  AND named_port(ist1)(k_task)(nm) = port
  AND held_rights(ist1)(k_task)(nm)(receive)
))

% Nobody changes the SID assignments...
AND task_sid(ist2) = task_sid(ist1)
AND port_sid(ist2) = port_sid(ist1)
% so the receiving access vector is
AND receiving_av =
  cached_access(ist1)(task_sid(ist1)(task), port_sid(ist1)(port))
AND cached_access(ist2) = cached_access(ist1)
% and it contains permission to receive.
AND receiving_av(receive_perm)

% Thread has an ri_status of ri_processed
AND ri_status(received_info(est1)(thread)) = ri_processed

% There is a message on port's queue...
AND 1 <= size(queue(ist1)(port))
% which the kernel records...
AND msg = elem(queue(ist1)(port))(1)
% it exists...
AND existing_messages(ist1)(msg)
% so the kernel uses it to construct task's new received_info...
AND k_rights = sent_rights(ist1)(msg)
AND u_rights = kernel_to_user(ist1, task, k_rights)
AND size(k_rights) = size(u_rights)
AND new_info =
  (#
    service_port := name,
    sending_sid := sending_sid(ist1)(msg),
    sending_av := av(ist1)(msg),
    user_msg :=
      (#
```

```
          user_data := sent_data(ist1)(msg),
          user_rights := u_rights
        #),
      op := op(ist1)(msg),
      reply_name :=
        LET
          reply_set : setof[nat] = {i : nat | (i > 0 AND i <= size(k_rights))
                           AND proj_1(elem(k_rights)(i)) = reply_port(ist1)(msg)}
        IN
        IF nonempty?(reply_set) THEN
          proj_1(elem(u_rights)(choose(reply_set)))
        ELSE
          null_name
        ENDIF,
      ri_status := ri_unprocessed
    #)
  AND received_info(est2) = received_info(est1) WITH [ (thread) := new_info ]
  % and then deletes it from port's queue...
  AND nonemptyfseq(queue(ist1)(port))
  AND queue(ist2)(port) = pop(queue(ist1)(port))
  % leaving all other queues unchanged.
  AND (FORALL (x : (existing_ports(ist1))) :
    port = x OR
      (existing_ports(ist2)(x) AND queue(ist2)(x) = queue(ist1)(x)))

  % The msg dies...
  AND existing_messages(ist2) = remove(msg, existing_messages(ist1))
  % so the kernel updates the message functions:
  AND (FORALL (x : (existing_messages(ist2))) : TRUE
    AND existing_messages(ist1)(x)
    AND sending_sid(ist2)(x) = sending_sid(ist1)(x)
    AND av(ist2)(x) = av(ist1)(x)
    AND op(ist2)(x) = op(ist1)(x)
    AND sent_data(ist2)(x) = sent_data(ist1)(x)
    AND sent_rights(ist2)(x) = sent_rights(ist1)(x)
    AND reply_port(ist2)(x) = reply_port(ist1)(x)
  )

  AND krm_names_and_rights(task,ist1,ist2,u_rights,k_rights)

  )


 END k_receive_message
% === ================
```

If the kernel is the receiver of the message, then the request denoted by the message is recorded in *pending_requests*. The possible requests that could get recorded are:

- **provide_access** if the operation id specified in the message is *provide_access_op*
- **set_special_port** if the operation id specified in the message is *set_host_special_port_op*
- **get_special_port** if the operation id specified in the message is *get_host_special_port_op*

## THEORY *k_kernel_request*

```
k_kernel_request : THEORY
```

```
  BEGIN
% =====


% IMPORTS
% =======

  IMPORTING k_state
  IMPORTING k_utilities


% THE k_kernel_request REQUEST
% === ================ =======
%
% k_kernel_request describes a transition in which a client sends a message
% to a kernel port and generates a new pending request.

  k_kernel_request(
    st1           : (K_STATE),            % The initial state of the transition
    st2           : (K_STATE),            % The final state of the transition
    ag            : (k_threads)           % The mediating agent
  )               : bool =

  EXISTS (
    ist1, ist2    : (K_INTERNAL_STATE),   % The internal state components
    est1, est2    : K_EXTERNAL_STATE,     % The externally visible components
    kreq          : K_REQ,                % The kernel request being processed
    new_req       : K_REQ,                % The request derived from the message
    thread        : THREAD,               % The client thread
    name          : NAME,                 % Where thread is sending the message
    op            : OP,                   % The operation being requested by the client
    reply_name    : NAME,                 % Clients name for reply_port
    reply_port    : PORT,                 % Where to enqueue the reply message
    usr_msg       : USER_MSG,             % The rights and data being sent
    task          : TASK,                 % Thread's owning task
    port          : PORT,                 % The port refered to by name
    sending_av    : ACCESS_VECTOR,        % The av associated with (task, port)
    u_rights      : USER_RIGHTS,          % The sequence of rights being sent
    k_rights      : K_RIGHTS              % The kernel version of u_rights
  )               : ( TRUE

    % Establish some variables.
    AND ist1 = int_st(st1)
    AND ist2 = int_st(st2)
    AND est1 = ext_st(st1)
    AND est2 = ext_st(st2)

    %% Avoid excess TCCs
    AND (FORALL (x1: (existing_tasks(ist1)), y1: (existing_tasks(ist2))):
            existing_tasks(ist1)(y1) AND existing_tasks(ist2)(x1))
    AND (FORALL (x1: (existing_threads(ist1)), y1: (existing_threads(ist2))):
            existing_threads(ist1)(y1) AND existing_threads(ist2)(x1))
    AND (FORALL (x1: (existing_threads(est1)), y1: (existing_threads(est2))):
            existing_threads(est1)(y1) AND existing_threads(est2)(x1))
    AND (FORALL (x1: (existing_ports(ist1)), y1: (existing_ports(ist2))):
            existing_ports(ist1)(y1) AND existing_ports(ist2)(x1))

    % In this transition we process an old request...
    AND difference(pending_requests(est1), pending_requests(est2)) =
      { x : K_REQ | x = kreq }
    % and its a request to send a message...
    AND send_message_req?(kreq)
    % with these particular parameters:
```

```
AND thread = smth(kreq)
AND name = smna(kreq)
AND op = smop(kreq)
AND reply_name = smrna(kreq)
AND usr_msg = smusr_msg(kreq)

% The thread exists...
AND existing_threads(est1)(thread)
AND existing_threads(est2)(thread)
AND existing_threads(ist1)(thread)
AND existing_threads(est2) = existing_threads(est1)
% and had been waiting...
AND thread_status(est1)(thread) = thread_waiting
% and continues to wait until processing of the request
% produces a reply message...
AND thread_status(est2) = thread_status(est1)
% and thread belongs to an existing task...
AND existing_tasks(ist1)(task)
AND existing_tasks(ist2)(task)
AND task_threads(ist1)(task)(thread)
AND existing_tasks(ist2) = existing_tasks(ist1)
AND task_threads(ist2) = task_threads(ist1)
% and name is in tasks name space...
AND task_names(ist1)(task)(name)
% and refers to an existing port...
AND port = named_port(ist1)(task)(name)
AND existing_ports(ist1)(port)
AND existing_ports(ist2)(port)
AND existing_ports(ist2) = existing_ports(ist1)
% and the receiver for port is the kernel.
AND existing_tasks(ist1)(k_task)
AND (EXISTS (nm : (task_names(ist1)(k_task))) : ( TRUE
  AND named_port(ist1)(k_task)(nm) = port
  AND held_rights(ist1)(k_task)(nm)(receive)
))

% Nobody changes the SID assignments...
AND task_sid(ist2) = task_sid(ist1)
AND port_sid(ist2) = port_sid(ist1)
% so the sending access vector is
AND sending_av = cached_access(ist1)(task_sid(ist1)(task), port_sid(ist1)(port))
AND cached_access(ist2) = cached_access(ist1)
% and it contains permission to send.
AND sending_av(send_perm)

% Moreover, task is using names from his name space and the cache contains
% av's for any live rights being sent in the message:
% NOTE: The user_rights is only needed in the case of a set_host_special_port
% request, but I think the kernel checks the validity of any rights that are
% present prior to construction of the actual request, so the following
% check is needed in the general case.
AND u_rights = user_rights(usr_msg)
AND (FORALL (n : nat | n>0 AND n <= size(u_rights)) : TRUE
  AND task_names(ist1)(task)(proj_1(elem(u_rights)(n)))
  AND existing_ports(ist1)(named_port(ist1)(task)(proj_1(elem(u_rights)(n)))) IMPLIES
    LET
      xname : NAME = proj_1(elem(u_rights)(n)),
      xport : PORT = named_port(ist1)(task)(xname),
      psid : SID = port_sid(ist1)(xport),
      tsid : SID = task_sid(ist1)(task)
    IN
      nonempty?(cached_access(ist1)(tsid, psid))
)
% the user rights are converted to kernel rights.
```

```
AND k_rights = user_to_kernel(ist1, u_rights, task)

% Name spaces do not change.
AND (FORALL (tk : (existing_tasks(ist2))) :  TRUE
  AND existing_tasks(ist1)(tk)
  AND task_names(ist2)(tk) = task_names(ist1)(tk)
  AND named_port(ist2)(tk) = named_port(ist1)(tk)
  AND held_rights(ist2)(tk) = held_rights(ist1)(tk)
  AND dead_names(ist2)(tk) = dead_names(ist1)(tk)
)

% The parameters for the new kernel request are obtained from the user message
% and the other send_message_req parameters:
% The operation must be one of the three we are specifying...
AND ( FALSE
  OR op = provide_access_op
  OR op = set_host_special_port_op
  OR op = get_host_special_port_op
)
% the reply_port is determined from the reply_name
AND reply_port =
  IF task_names(ist1)(task)(reply_name) THEN
    named_port(ist1)(task)(reply_name)
  ELSE
    null_port
  ENDIF

% The other parameters are request specific:

% provide_access_req
AND ( op = provide_access_op IMPLIES
  new_req =
    LET
      (sid1, sid2, pav) = data_to_sid_sid_av(user_data(usr_msg))
    IN
      provide_access_req(thread, op, sending_av, port, sid1, sid2, pav, reply_port)
)

% set_ssp_req
AND ( op = set_host_special_port_op IMPLIES ( TRUE
  AND size(k_rights) = 1
  AND proj_2(elem(k_rights)(1)) = send
  AND new_req =
    LET
      npt : PORT = proj_1(elem(k_rights)(1))
    IN
      set_ssp_req(thread, op, sending_av, port, npt, reply_port)
  )
)

% get_ssp_req
AND ( op = get_host_special_port_op IMPLIES
  new_req =
      get_ssp_req(thread, op, sending_av, port, reply_port)
)

% The new pending_requests set contains new_req as a unique
% element not in the old set:
AND difference(pending_requests(est2), pending_requests(est1)) =
  {kr : K_REQ | kr = new_req}

% The components of state not mentioned above remain unchanged:
AND received_info(est2) = received_info(est1)
)
```

```
  END k_kernel_request
% === ================
```

### 17.2.3 Provide Access

The **provide_access** request is used to load access vectors into the kernel's access vector cache.[28] The request specifies a SID-SID-vector triple that should be added to the cache. Assuming the client has $provide\_access\_perm$ to the $host\_name$, the triple is added to the access vector cache. A reply message is sent indicating whether the request was successful and the request to which the reply message is a response. The latter is indicated by specifying the operation id of the reply message to be a function, $op\_to\_reply\_op$, of the operation id of the request message. In Mach, the $op\_to\_reply\_op$ function is implemented by adding a constant to the operation id in the request message. [29]

## THEORY *k_provide_access*

```
k_provide_access : THEORY

  BEGIN
% =====


% IMPORTS
% =======

  IMPORTING k_state
  IMPORTING k_utilities


% THE k_provide_access REQUEST
% === ================ =======
%
% k_provide_access describes a transition where a client has requested
% to add an access vector to the kernel's cache.

  k_provide_access (
    st1          : (K_STATE),            % The initial state of the transition
    st2          : (K_STATE),            % The final state of the transition
    ag           : (k_threads)           % The mediating agent
  )              : bool =

  EXISTS (
    ist1, ist2   : (K_INTERNAL_STATE),   % The internal state components
    est1, est2   : K_EXTERNAL_STATE,     % The externally visible components
    kreq         : K_REQ,                % The kernel request being processed
```

[28] Note that this is specified as being processed through the host name port although DTOS actually processes the request through a "kernel reply port". Also note that in the model presented here, the only change made to the cache is the addition of the provided access to the cache. In DTOS, it is also possible that an existing cache entry will be reclaimed to make space for the new entry.

[29] The current DTOS system does not send a reply message since this request is actually only called as a Security Server response to an earlier kernel request for an access computation. Also note that $provide\_access\_op$ is the same as $op\_to\_reply\_op(request\_access\_op)$ since the Security Server response to a request from the kernel to compute access must be interpreted by the kernel as a request to load an access vector.

```
    client       : THREAD,            % The client thread
    op           : OP,                % The operation; redundant (= provide_access_op)
    client_av    : ACCESS_VECTOR,     % The av associated with (client, svc_port)
    svc_port     : PORT,              % The port on which the request was received
    ssid         : SID,               % The input subject sid
    osid         : SID,               % The input object sid
    new_av       : ACCESS_VECTOR,     % The input access vector
    reply_port   : PORT,              % The port where reply message is enqueued
    msg          : MESSAGE            % The reply message enqueued at reply_port
  )              : ( TRUE

    % Establish some variables.
    AND ist1 = int_st(st1)
    AND ist2 = int_st(st2)
    AND est1 = ext_st(st1)
    AND est2 = ext_st(st2)

    %% Avoid generation of too many type check conditions in PVS
    AND (FORALL (x1: (existing_tasks(ist1)), y1: (existing_tasks(ist2))):
          existing_tasks(ist1)(y1) AND existing_tasks(ist2)(x1))
    AND (FORALL (x1: (existing_ports(ist1)), y1: (existing_ports(ist2))):
          existing_ports(ist1)(y1) AND existing_ports(ist2)(x1))
    AND (FORALL (x1: (existing_threads(est1)), y1: (existing_threads(est2))):
          existing_threads(est1)(y1) AND existing_threads(est2)(x1))

    % Many components are invariant:
    AND existing_threads(est2) = existing_threads(est1)
    AND existing_tasks(ist2) = existing_tasks(ist1)
    AND task_threads(ist2) = task_threads(ist1)
    AND existing_ports(ist2) = existing_ports(ist1)
    AND task_sid(ist2) = task_sid(ist1)
    AND port_sid(ist2) = port_sid(ist1)
    AND received_info(est2) = received_info(est1)
    AND (FORALL (x : (existing_tasks(ist2))) : TRUE
      AND existing_tasks(ist1)(x)
      AND task_names(ist2)(x) = task_names(ist1)(x)
      AND named_port(ist2)(x) = named_port(ist1)(x)
      AND held_rights(ist2)(x) = held_rights(ist1)(x)
      AND dead_names(ist2)(x) = dead_names(ist1)(x)
    )

    % In this transformation we process a kernel request...
    AND NOT pending_requests(est2)(kreq)
    AND pending_requests(est1) = add(kreq, pending_requests(est2))
    % and its a request to provide access vector information...
    AND provide_access_req?(kreq)
    % with these particular parameters:
    AND client = pact(kreq)
    AND op = paop(kreq)
    AND client_av = pacav(kreq)
    AND svc_port = passport(kreq)
    AND ssid = passi(kreq)
    AND osid = paosi(kreq)
    AND new_av = parav(kreq)
    AND reply_port = parp(kreq)

    % The client is an existing thread...
    AND existing_threads(ist1)(client)
    AND existing_threads(ist2)(client)
    AND existing_threads(est1)(client)
    AND existing_threads(est2)(client)
    % and reply_port is an existing port...
    AND existing_ports(ist1)(reply_port)
    % and client_av contains permission to provide access...
```

```
AND client_av(provide_access_perm)
% and the client had been waiting but now is running...
AND thread_status(est1)(client) = thread_waiting
AND thread_status(est2) = thread_status(est1) WITH [ (client) := thread_running ]
% and the request was received on the correct port...
AND existing_tasks(ist1)(k_task)
AND task_names(ist1)(k_task)(host_name)
AND svc_port = named_port(ist1)(k_task)(host_name)
% and the cache gets updated.
AND cached_access(ist2) = cached_access(ist1) WITH
  [ ((ssid, osid)) := new_av ]

% The kernel enqueues the reply message at reply_port:
% The set of existing messages grows...
AND NOT existing_messages(ist1)(msg)
AND existing_messages(ist2) = add(msg, existing_messages(ist1))
% the msg gets added to reply_port's queue...
AND queue(ist2) = (LAMBDA (pt : (existing_ports(ist1))) :
  IF (pt = reply_port) THEN
    tack_on(msg, queue(ist1)(reply_port))
  ELSE
    queue(ist1)(pt)
  ENDIF
)
% the sending sid gets recorded...
AND sending_sid(ist2) = (LAMBDA (x : (existing_messages(ist2))) :
  IF existing_messages(ist1)(x) THEN
    sending_sid(ist1)(x)
  ELSE
    sid_witness
  ENDIF
)
% no access vector is sent...
AND av(ist2) = (LAMBDA (x : (existing_messages(ist2))) :
  IF existing_messages(ist1)(x) THEN
    av(ist1)(x)
  ELSE
    emptyset[PERMISSION]
  ENDIF
)
% the operation gets recorded...
AND op(ist2) = (LAMBDA (x : (existing_messages(ist2))) :
  IF existing_messages(ist1)(x) THEN
    op(ist1)(x)
  ELSE
    op_to_reply_op(op)
  ENDIF
)
% the data (indicating success) gets recorded...
AND sent_data(ist2) = (LAMBDA (x : (existing_messages(ist2))) :
  IF existing_messages(ist1)(x) THEN
    sent_data(ist1)(x)
  ELSE
    success_data
  ENDIF
)
% the reply port is the same as the port where the message is enqueued...
AND reply_port(ist2) = (LAMBDA (x : (existing_messages(ist2))) :
  IF existing_messages(ist1)(x) THEN
    reply_port(ist1)(x)
  ELSE
    reply_port
  ENDIF
)
```

```
    % no rights are sent.
    AND sent_rights(ist2) = (LAMBDA (x : (existing_messages(ist2))) :
      IF existing_messages(ist1)(x) THEN
        sent_rights(ist1)(x)
      ELSE
        null_seq
      ENDIF
    )
  )

  END k_provide_access
% === ================
```

### 17.2.4 Request Access

The **request_access** transition is used to request access vectors from the Security Server.[30] This is modeled as simply the addition of a new pending request to send a message to the Security Server requesting an access vector.

## THEORY *k_request_access*

```
k_request_access : THEORY

  BEGIN
% =====


% IMPORTS
% =======

  IMPORTING k_state
  IMPORTING k_utilities


% THE k_request_access REQUEST
% === ================ =======
%
% k_request_access describes a transition where the kernel sends a message
% to the security server requesting an access vector computation on a pair
% of sid's.

  k_request_access (
    st1            : (K_STATE),            % The initial state of the transition
    st2            : (K_STATE),            % The final state of the transition
    ag             : (k_threads)           % The mediating agent
  )                : bool =

  EXISTS (
    ist1, ist2     : (K_INTERNAL_STATE),   % The internal state components
    est1, est2     : K_EXTERNAL_STATE,     % The externally visible components
    new_req        : K_REQ,                % The new kernel request produced
    sid1           : SID,                  % The sids for which the kernel is requesting
    sid2           : SID                   %    an access vector
  )                : ( TRUE
```

[30] Note that although this is modeled as if the kernel can request access vectors at whim, the only time the DTOS kernel actually requests access vectors is when they are needed to process a pending request.

```
    % Establish some variables.
    AND ist1 = int_st(st1)
    AND ist2 = int_st(st2)
    AND est1 = ext_st(st1)
    AND est2 = ext_st(st2)

    % Almost all components are invariant:
    AND existing_tasks(ist2) = existing_tasks(ist1)
    AND existing_threads(est2) = existing_threads(est1)
    AND received_info(est2) = received_info(est1)
    AND thread_status(est2) = thread_status(est1)
    AND existing_ports(ist2) = existing_ports(ist1)
    AND existing_messages(ist2) = existing_messages(ist1)
    AND task_threads(ist2) = task_threads(ist1)
    AND task_names(ist2) = task_names(ist1)
    AND dead_names(ist2) = dead_names(ist1)
    AND named_port(ist2) = named_port(ist1)
    AND held_rights(ist2) = held_rights(ist1)
    AND task_sid(ist2) = task_sid(ist1)
    AND port_sid(ist2) = port_sid(ist1)
    AND cached_access(ist2) = cached_access(ist1)
    AND queue(ist2) = queue(ist1)
    AND sending_sid(ist2) = sending_sid(ist1)
    AND av(ist2) = av(ist1)
    AND op(ist2) = op(ist1)
    AND sent_data(ist2) = sent_data(ist1)
    AND reply_port(ist2) = reply_port(ist1)
    AND sent_rights(ist2) = sent_rights(ist1)

    % In this transformation we produce a new request...
    AND NOT pending_requests(est1)(new_req)
    AND pending_requests(est2) = add(new_req, pending_requests(est1))
    % and its a send_message request to the security server.
    AND new_req = send_message_req(
      ag,                  % The active agent, a kernel thread, is making the request
      ss_name,             % The message is going to the security server
      request_access_op,%  The operation is a request for an access vector
      ss_name,             % The reply name is the same
      null_user_msg WITH [ (user_data) :=
        sid_sid_to_data(sid1, sid2) ]          % The user message being sent
    )
  )

  END k_request_access
% === ================
```

### 17.2.5 Set Security Server Port

The **set_security_server** request is used to set the master security server port. If the request succeeds, the port associated with $ss\_name$ is modified.

## THEORY *k_set_ss_port*

```
k_set_ss_port : THEORY

  BEGIN
% =====
```

```
% IMPORTS
% =======

  IMPORTING k_state
  IMPORTING k_utilities


% THE k_set_ss_port REQUEST
% === ============= =======
%
% k_set_ss_port describes a transition where a client has requested
% to set the kernels security server port.

  k_set_ss_port (
    st1          : (K_STATE),             % The initial state of the transition
    st2          : (K_STATE),             % The final state of the transition
    ag           : (k_threads)            % The mediating agent
  )              : bool =

  EXISTS (
    ist1, ist2   : (K_INTERNAL_STATE),    % The internal state components
    est1, est2   : K_EXTERNAL_STATE,      % The externally visible components
    kreq         : K_REQ,                 % The kernel request being processed
    client       : THREAD,                % The client thread
    op           : OP,                    % The operation; redundant (= set_ss_op)
    client_av    : ACCESS_VECTOR,         % The av associated with (client, svc_port)
    svc_port     : PORT,                  % The port on which the request was received
    new_port     : PORT,                  % The new security server port
    reply_port   : PORT,                  % The port where reply message is enqueued
    msg          : MESSAGE                % The reply message enqueued at reply_port
  )              : ( TRUE

    % Establish some variables.
    AND ist1 = int_st(st1)
    AND ist2 = int_st(st2)
    AND est1 = ext_st(st1)
    AND est2 = ext_st(st2)

    %% Avoid excess TCCs
    AND (FORALL (x1: (existing_tasks(ist1)), y1: (existing_tasks(ist2))):
          existing_tasks(ist1)(y1) AND existing_tasks(ist2)(x1))
    AND (FORALL (x1: (existing_ports(ist1)), y1: (existing_ports(ist2))):
          existing_ports(ist1)(y1) AND existing_ports(ist2)(x1))
    AND (FORALL (x1: (existing_threads(est1)), y1: (existing_threads(est2))):
          existing_threads(est1)(y1) AND existing_threads(est2)(x1))

    % Many components are invariant:
    AND existing_threads(est2) = existing_threads(est1)
    AND existing_tasks(ist2) = existing_tasks(ist1)
    AND task_threads(ist2) = task_threads(ist1)
    AND existing_ports(ist2) = existing_ports(ist1)
    AND task_sid(ist2) = task_sid(ist1)
    AND port_sid(ist2) = port_sid(ist1)
    AND received_info(est2) = received_info(est1)
    AND cached_access(ist2) = cached_access(ist1)

    % In this transformation we process a kernel request...
    AND NOT pending_requests(est2)(kreq)
    AND pending_requests(est1) = add(kreq, pending_requests(est2))
    % and it's a request to set the kernel's security server port
    AND set_ssp_req?(kreq)
    % with these particular parameters:
```

```
AND client = ssct(kreq)
AND op = ssop(kreq)
AND client_av = ssav(kreq)
AND svc_port = sssp(kreq)
AND new_port = ssnp(kreq)
AND reply_port = ssrp(kreq)

% The client is an existing thread...
AND existing_threads(ist1)(client)
AND existing_threads(est1)(client)
AND existing_threads(est2)(client)
% and reply_port is an existing port...
AND existing_ports(ist1)(reply_port)
% and new_port is an existing port...
AND existing_ports(ist1)(new_port)
% and client_av contains permission to set the ss port...
AND client_av(set_ss_perm)
% and the client had been waiting but now is running...
AND thread_status(est1)(client) = thread_waiting
AND thread_status(est2) = thread_status(est1) WITH [ (client) := thread_running ]
% and the request was received on the correct port...
AND existing_tasks(ist1)(k_task)
AND task_names(ist1)(k_task)(host_name)
AND svc_port = named_port(ist1)(k_task)(host_name)
% and the kernel's name space gets updated.
AND existing_tasks(ist2) = existing_tasks(ist1)
AND FORALL (x : (existing_tasks(ist2))) : ( TRUE
  AND existing_tasks(ist1)(x)
  AND task_names(ist2)(x) = task_names(ist1)(x)
  AND ( FALSE
    OR ( TRUE
      AND x = k_task
      AND task_names(ist2)(k_task)(ss_name)
      AND named_port(ist2)(k_task) = named_port(ist1)(k_task) WITH
        [ (ss_name) := new_port ]
    )
    OR named_port(ist2)(x) = named_port(ist1)(x)
  )
  AND held_rights(ist2)(x) = held_rights(ist1)(x)
  AND dead_names(ist2)(x) = dead_names(ist1)(x)
)

% The kernel enqueues the reply message at reply_port:
% The set of existing messages grows...
AND NOT existing_messages(ist1)(msg)
AND existing_messages(ist2) = add(msg, existing_messages(ist1))
% the msg gets added to reply_port's queue...
AND queue(ist2) = (LAMBDA (pt : (existing_ports(ist1))) :
  IF (pt = reply_port) THEN
    tack_on(msg, queue(ist1)(reply_port))
  ELSE
    queue(ist1)(pt)
  ENDIF
)
% the sending sid gets recorded...
AND sending_sid(ist2) = (LAMBDA (x : (existing_messages(ist2))) :
  IF existing_messages(ist1)(x) THEN
    sending_sid(ist1)(x)
  ELSE
    sid_witness
  ENDIF
)
% no access vector is sent...
AND av(ist2) = (LAMBDA (x : (existing_messages(ist2))) :
```

```
      IF existing_messages(ist1)(x) THEN
         av(ist1)(x)
      ELSE
         emptyset[PERMISSION]
      ENDIF
   )
   % the operation gets recorded...
   AND op(ist2) = (LAMBDA (x : (existing_messages(ist2))) :
      IF existing_messages(ist1)(x) THEN
         op(ist1)(x)
      ELSE
         op_to_reply_op(op)
      ENDIF
   )
   % the data (indicating success) gets recorded...
   AND sent_data(ist2) = (LAMBDA (x : (existing_messages(ist2))) :
      IF existing_messages(ist1)(x) THEN
         sent_data(ist1)(x)
      ELSE
         success_data
      ENDIF
   )
   % the reply port is the same as the port where the message is enqueued...
   AND reply_port(ist2) = (LAMBDA (x : (existing_messages(ist2))) :
      IF existing_messages(ist1)(x) THEN
         reply_port(ist1)(x)
      ELSE
         reply_port
      ENDIF
   )
   % no rights are sent.
   AND sent_rights(ist2) = (LAMBDA (x : (existing_messages(ist2))) :
      IF existing_messages(ist1)(x) THEN
         sent_rights(ist1)(x)
      ELSE
         null_seq
      ENDIF
   )
 )

 END k_set_ss_port
% === ============
```

### 17.2.6 Get Security Server Port

The **get_security_server** request is used to retrieve the master security server port.

## THEORY *k_get_ss_port*

```
k_get_ss_port : THEORY

 BEGIN
% =====


% IMPORTS
% =======

 IMPORTING k_state
```

```
IMPORTING k_utilities


% THE k_get_ss_port REQUEST
% === ============= =======
%
% k_get_ss_port describes a transition where a client has requested
% to get the kernels security server port.

k_get_ss_port (
   st1          : (K_STATE),              % The initial state of the transition
   st2          : (K_STATE),              % The final state of the transition
   ag           : (k_threads)             % The mediating agent
)               : bool =

EXISTS (
   ist1, ist2   : (K_INTERNAL_STATE),     % The internal state components
   est1, est2   : K_EXTERNAL_STATE,       % The externally visible components
   kreq         : K_REQ,                  % The kernel request being processed
   client       : THREAD,                 % The client thread
   op           : OP,                     % The operation; redundant (= get_ss_op)
   client_av    : ACCESS_VECTOR,          % The av associated with (client, svc_port)
   svc_port     : PORT,                   % The port on which the request was received
   reply_port   : PORT,                   % The port where reply message is enqueued
   msg          : MESSAGE                 % The reply message enqueued at reply_port
)               : ( TRUE

   % Establish some variables.
   AND ist1 = int_st(st1)
   AND ist2 = int_st(st2)
   AND est1 = ext_st(st1)
   AND est2 = ext_st(st2)

   %% Avoid excess TCCs
   AND (FORALL (x1: (existing_tasks(ist1)), y1: (existing_tasks(ist2))):
            existing_tasks(ist1)(y1) AND existing_tasks(ist2)(x1))
   AND (FORALL (x1: (existing_ports(ist1)), y1: (existing_ports(ist2))):
            existing_ports(ist1)(y1) AND existing_ports(ist2)(x1))
   AND (FORALL (x1: (existing_threads(est1)), y1: (existing_threads(est2))):
            existing_threads(est1)(y1) AND existing_threads(est2)(x1))

   % Many components are invariant:
   AND existing_threads(est2) = existing_threads(est1)
   AND existing_tasks(ist2) = existing_tasks(ist1)
   AND task_threads(ist2) = task_threads(ist1)
   AND existing_ports(ist2) = existing_ports(ist1)
   AND task_sid(ist2) = task_sid(ist1)
   AND port_sid(ist2) = port_sid(ist1)
   AND received_info(est2) = received_info(est1)
   AND cached_access(ist2) = cached_access(ist1)
   AND task_names(ist2) = task_names(ist1)
   AND named_port(ist2) = named_port(ist1)
   AND held_rights(ist2) = held_rights(ist1)
   AND dead_names(ist2) = dead_names(ist1)

   % In this transformation we process a kernel request...
   AND NOT pending_requests(est2)(kreq)
   AND pending_requests(est1) = add(kreq, pending_requests(est2))
   % and its a request to get the kernel's security server port
   AND get_ssp_req?(kreq)
   % with these particular parameters:
   AND client = gsct(kreq)
   AND op = gsop(kreq)
   AND client_av = gsav(kreq)
```

```
AND svc_port = gssp(kreq)
AND reply_port = gsrp(kreq)

% The client is an existing thread...
AND existing_threads(ist1)(client)
AND existing_threads(est1)(client)
AND existing_threads(est2)(client)
% and reply_port is an existing port...
AND existing_ports(ist1)(reply_port)
% and client_av contains permission to get the ss port...
AND client_av(get_ss_perm)
% and the client had been waiting but now is running...
AND thread_status(est1)(client) = thread_waiting
AND thread_status(est2) = thread_status(est1) WITH [ (client) := thread_running ]
% and the request was received on the correct port...
AND existing_tasks(ist1)(k_task)
AND task_names(ist1)(k_task)(host_name)
AND task_names(ist1)(k_task)(ss_name)
AND svc_port = named_port(ist1)(k_task)(host_name)

% The kernel enqueues the reply message at reply_port:
% The set of existing messages grows...
AND NOT existing_messages(ist1)(msg)
AND existing_messages(ist2) = add(msg, existing_messages(ist1))
% the msg gets added to reply_port's queue...
AND queue(ist2) = (LAMBDA (pt : (existing_ports(ist1))) :
  IF (pt = reply_port) THEN
    tack_on(msg, queue(ist1)(reply_port))
  ELSE
    queue(ist1)(pt)
  ENDIF
)
% the sending sid gets recorded...
AND sending_sid(ist2) = (LAMBDA (x : (existing_messages(ist2))) :
  IF existing_messages(ist1)(x) THEN
    sending_sid(ist1)(x)
  ELSE
    sid_witness
  ENDIF
)
% no access vector is sent...
AND av(ist2) = (LAMBDA (x : (existing_messages(ist2))) :
  IF existing_messages(ist1)(x) THEN
    av(ist1)(x)
  ELSE
    emptyset[PERMISSION]
  ENDIF
)
% the operation gets recorded...
AND op(ist2) = (LAMBDA (x : (existing_messages(ist2))) :
  IF existing_messages(ist1)(x) THEN
    op(ist1)(x)
  ELSE
    op_to_reply_op(op)
  ENDIF
)
% the data (indicating success) gets recorded...
AND sent_data(ist2) = (LAMBDA (x : (existing_messages(ist2))) :
  IF existing_messages(ist1)(x) THEN
    sent_data(ist1)(x)
  ELSE
    success_data
  ENDIF
)
```

```
    % the reply port is the same as the port where the message is enqueued...
    AND reply_port(ist2) = (LAMBDA (x : (existing_messages(ist2))) :
      IF existing_messages(ist1)(x) THEN
        reply_port(ist1)(x)
      ELSE
        reply_port
      ENDIF
    )
    % ss port is sent back
    AND sent_rights(ist2) = (LAMBDA (x : (existing_messages(ist2))) :
      IF existing_messages(ist1)(x) THEN
        sent_rights(ist1)(x)
      ELSE
        (#
          size := 1,
          elem := (LAMBDA (x : nat | x > 0 AND x <= 1) :
            (named_port(ist1)(k_task)(ss_name), send)
          )
        #)
      ENDIF
    )
  )


  END k_get_ss_port
% === ============
```

### 17.2.7 Summary of Operations

A kernel operation consists of any transition with a kernel thread serving as the agent such that the start and final states of the transition are either consistent with one of the operations defined above or look the same with respect to $k\_view$.

## THEORY *k_ops*

```
k_ops : THEORY

  BEGIN
% =====


% IMPORTS
% =======

  IMPORTING k_send_message
  IMPORTING k_receive_message
  IMPORTING k_kernel_request
  IMPORTING k_provide_access
  IMPORTING k_request_access
  IMPORTING k_set_ss_port
  IMPORTING k_get_ss_port


% VARIABLES
% =========

  st1, st2 : VAR (K_STATE)
```

```
  thread : VAR THREAD
  th: VAR (k_threads)


% THE OPERATIONS
% === ==========

  k_op(st1, st2, th) : bool = FALSE
     OR k_send_message(st1, st2, th)
     OR k_receive_message(st1, st2, th)
     OR k_kernel_request(st1, st2, th)
     OR k_provide_access(st1, st2, th)
     OR k_request_access(st1, st2, th)
     OR k_set_ss_port(st1, st2, th)
     OR k_get_ss_port(st1, st2, th)

  k_guar(st1,st2,thread) : bool =
   k_threads(thread) AND
        (k_view(st1, st2)
           OR k_op(st1, st2,thread))


  END k_ops
% === =====
```

## 17.3 Environment Assumptions

The environment to the kernel is constrained as follows:

- The only portions of the kernel state that can be modified by agents other than those in *kernel_thread* are *pending_requests*, *thread_status*, and *received_info*.
- The only change allowed to *pending_requests* is the addition of a send or receive message request in the name of the active agent. This active agent can be either the kernel itself or a thread executing on the kernel.
- The only change allowed to *thread_status* is the changing of the active agent's status from *thread_running* to *thread_waiting* to indicate it is waiting for the kernel to process a message.
- The only change allowed to *received_info* is the changing of the active agent's *ri_status* field from *ri_unprocessed* to *ri_processed* to indicate it has processed the information.

The submission of a kernel request by *thread* is modeled as the simultaneous addition of a request by *thread* to *pending_requests* and the changing of $thread\_status(thread)$ to *thread_waiting*. A thread *thread* may change $received\_info(thread)$ to *ri_processed* whenever it chooses. We take *hidd* for the kernel to be identical to *rely*. This is equivalent to assuming that it is not possible for any component to violate the environment assumptions of the kernel. This approach has the advantage that it reduces the number of proof obligations when the tolerance analysis is performed; the obligations to show that no component violates the kernel's assumptions essentially disappear since we can prove the obligations based only upon the *hidd* and *rely* of the kernel. The disadvantage is that we might miss errors in the other component specifications. It is probably reasonable to assume that in the implementation no component can violate the assumptions of the kernel as outlined above. Thus, these would be errors in the specification of a component which cannot be duplicated in the component's implementation.

We note that taking $hidd = rely$ results in a specification similar to those used by Shankar. However, whereas in Shankar's framework the equivalence is required for all components, we can choose to make this equivalence for individual components.

## THEORY *k_rely*

```
k_rely : THEORY

  BEGIN
% =====


% IMPORTS
% =======

  IMPORTING k_state


% VARIABLES
% =========

  st1, st2 : VAR (K_STATE)
  ist1, ist2 : VAR (K_INTERNAL_STATE)
  est1, est2 : VAR K_EXTERNAL_STATE
  ag, thread : VAR THREAD
  kern_req : VAR KERNEL_REQ


% ENVIRONMENTAL ASSUMPTIONS
% ============= ===========

% 1. Nobody else changes my internal state:
  k_rely_internal(ist1, ist2) : bool =
    ist1 = ist2

% 2. Nobody changes the set of existing threads:
  k_rely_existing_threads(est1, est2) : bool =
    existing_threads(est1) = existing_threads(est2)

% 3. The only change allowed to pending_requests is the addition
%      of a send or receive message request by the active agent:
  k_rely_pending_requests(est1, est2, ag) : bool =
    FORALL kern_req : ( TRUE
        AND pending_requests(est1)(kern_req) => pending_requests(est2)(kern_req)
        AND (NOT pending_requests(est1)(kern_req)
                AND pending_requests(est2)(kern_req)
             => ( FALSE
                  OR (send_message_req?(kern_req) AND smth(kern_req) = ag)
                  OR (receive_message_req?(kern_req) AND rmth(kern_req) = ag)
        )))

% 4. The only change allowed to thread_status is the changing of the active agent's
%      status from thread_running to thread_waiting:
  k_rely_thread_status(est1, est2, ag) : bool =
    FORALL (thread : THREAD) : ( FALSE
      OR NOT existing_threads(est1)(thread)
      OR NOT existing_threads(est2)(thread)
      OR thread_status(est1)(thread) = thread_status(est2)(thread)
      OR (TRUE
        AND thread_status(est1)(thread) = thread_running
```

```
        AND thread_status(est2)(thread) = thread_waiting
        AND thread = ag
      )
    )

% 5. The only change allowed to received_info is the changing of the active agent's
%       ri_status from ri_unprocessed to ri_processed.
  k_rely_received_info(est1, est2, ag) : bool =
    FORALL (thread : THREAD) : ( FALSE
      OR NOT existing_threads(est1)(thread)
      OR NOT existing_threads(est2)(thread)
      OR received_info(est1)(thread) = received_info(est2)(thread)
      OR ( TRUE
        AND ri_status(received_info(est1)(thread)) = ri_unprocessed
        AND ri_status(received_info(est2)(thread)) = ri_processed
        AND thread = ag
      )
    )


% THE ENVIRONMENTAL ASSUMPTIONS
% === ============= ===========
  k_rely(st1, st2, ag) : bool = TRUE
    AND NOT k_threads(ag)
    AND k_rely_internal(int_st(st1), int_st(st2))
    AND k_rely_existing_threads(ext_st(st1), ext_st(st2))
    AND k_rely_pending_requests(ext_st(st1), ext_st(st2), ag)
    AND k_rely_thread_status(ext_st(st1), ext_st(st2), ag)
    AND k_rely_received_info(ext_st(st1), ext_st(st2), ag)


k_rely_refl: THEOREM
   k_threads(ag) OR k_rely(st1,st1,ag)


%% HIDD
%% ====

%% We assume that clients of the kernel are unable to violate the
%% environment assumptions made above by the kernel.
k_hidd(st1,st2,ag) : bool =
   k_rely(st1, st2, ag)


  END k_rely
% === ======
```

## 17.4 Component Specification

We use the set $initial\_k\_states$ to denote the valid initial states for the kernel. A valid initial state has the following properties:

- There are no pending requests in the initial state.
- There are no messages queued at ports in the initial state.

A kernel is a component having state type $K\_STATE$, satisfying initial constraint $initial\_k\_states$, and executing only the transitions defined in Section 17.2.

## THEORY *k_spec*

```
k_spec : THEORY

  BEGIN
% =====


% IMPORTS
% =======

  IMPORTING k_state
  IMPORTING k_ops
  IMPORTING k_rely
  IMPORTING k_state_witness
  IMPORTING component_aux[(K_STATE), THREAD]


% VARIABLES
% =========

  st, st1, st2 : VAR (K_STATE)
  ag : VAR THREAD


% COMPONENT DEFINITIONS
% ========= ===========

% 1. init---Initial conditions (must be non-empty, so we need a witness):
  initial_k_states(st) : bool =
    FORALL (p : PORT | existing_ports(int_st(st))(p)) : queue(int_st(st))(p)=null_seq[MESSAGE]
    AND empty?(pending_requests(ext_st(st)))

% NOTE: Ought to be more conditions here.  E.g., AVC should contain
% permissions for the kernel to the security server and vice versa,
% and no other permissions; need to specify that tasks for the other
% components exist.  Then need to update the initial state witness.

  k_state_witness_initial : THEOREM
    initial_k_states(k_state_witness)


% THE KERNEL COMPONENT
% === ====== =========


base_k_comp : base_comp_t =
     (# init := initial_k_states,
        guar := k_guar,
        rely := k_rely,
        hidd := k_hidd,
        cags := k_threads,
        view := k_view,
        wfar := emptyset[TRANSITION_CLASS[(K_STATE), THREAD]],
        sfar := emptyset[TRANSITION_CLASS[(K_STATE), THREAD]] #)

  k_view_eq: THEOREM view_eq(base_k_comp)
```

```
k_comp_init: THEOREM init_restriction(base_k_comp)

k_comp_guar: THEOREM guar_restriction(base_k_comp)

k_comp_rely_hidd: THEOREM rely_hidd_restriction(base_k_comp)

k_comp_hidd: THEOREM hidd_restriction(base_k_comp)

k_comp_rely: THEOREM rely_restriction(base_k_comp)

k_comp_cags: THEOREM cags_restriction(base_k_comp)

k_comp_guar_stuttering: THEOREM guar_stuttering_restriction(base_k_comp)

k_comp_rely_stuttering: THEOREM rely_stuttering_restriction(base_k_comp)

k_comp : (comp_t) = base_k_comp

END k_spec
% === ======
```

# Section 18 Common Transitions

In this section we define some utility functions that will be used in the specifications of later components to describe the ways in which they manipulate their kernel interface. The function $effects\_on\_kernel\_state$ limits the changes that components make to their $KERNEL\_SHARED\_STATE$. Components can update the $pending\_requests$ and $thread\_status$ portions of the system state by introducing requests in the name of their threads. The only other change a component can make to the kernel state is the updating of $received\_info$ to indicate that it has processed a request it previously received.[31]

Two other utility functions help define specific modifications that components make to their $KERNEL\_SHARED\_STATE$:

- an agent (thread) of a component can mark its received information as processed via function $process\_request(thread, thread\_info_1, thread\_info_2)$,
- an agent of a component can process a received message and then reply to it by submitting a kernel request via function $make\_service\_request(cags,\ service\_ports,\ reply\_port,\ requested\_op,\ needed\_perm,\ user\_msg_1,\ user\_msg_2,\ kst_1,\ kst_2)$ where
  - the agent is in $cags$,
  - $user\_msg_1$ is the received message,
  - the message was received on a port in $service\_ports$,
  - $requested\_op$ is the operation in the received message,
  - $needed\_perm$ is a permission that the component requires in order to perform its service and send the reply and which must be in the access vector contained in the message,
  - $user\_msg_2$ is the reply message,
  - the reply is sent to the reply name in the received message, it has an operation based upon the operation of the received message, and its $reply\_port$ is the reply port sent in the reply message,
  - $kst_1$ and $kst_2$ are the component's kernel state in the starting and final states of the transition, respectively,
  - the reply message is added to $pending\_requests$,
  - the agent waits for its pending request to be processed.

*Editorial Note:*
The utility function $make\_service\_request$ could have been used a little more often in the Crypto Subsystem example. However, it turns out that there are relatively few transitions in the subsystem in

[31] Since $effects\_on\_kernel\_state$ is used to constrain the steps of all non-kernel components in this report, the assumption that $hidd$ and $rely$ are equal for the kernel could probably be removed without a tremendous increase in the complexity of the tolerance proofs. We could demonstrate that any component that obeys $effects\_on\_kernel\_state$ satisfies the kernel's environment assumptions. Of course, the use of $effects\_on\_kernel\_state$ in defining a component means that any transition of that component that is erroneously specified in a way that is inconsistent with $effects\_on\_kernel\_state$ will not actually end up as a transition of the component.

which a component receives a request and in the same transition can send the reply. More commonly, a component will first make requests of other components and wait for the responses before replying to the original request.

## THEORY *dtos_kernel_shared_ops*

```
dtos_kernel_shared_ops: THEORY
  BEGIN

  %% Note that the restrictions on transition described here apply
  %% only within a component's part of the kernel shared state.  The
  %% in the common state for the components we assume the kst of each
  %% non-kernel component to be a substate of the kernel's shared
  %% state.  The hidd relations are used to ensure that components
  %% being composed do not mess with another component's kst.

  IMPORTING dtos_kernel_shared_state

  THREAD_INFO:
      TYPE =
        [# existing_threads: setof[THREAD],
           received_info: [(existing_threads) -> RECEIVED_INFO],
           thread_status: [(existing_threads) -> THREAD_STATUS] #]

  thread_info1, thread_info2: VAR THREAD_INFO

  thread: VAR THREAD

  ri, ri1, ri2: VAR RECEIVED_INFO

  process_request(thread, thread_info1, thread_info2): bool =
    existing_threads(thread_info1)(thread)
        AND existing_threads(thread_info2) = existing_threads(thread_info1)
          AND
        (EXISTS ri1, ri2:
           ri_status(ri1) = ri_unprocessed
               AND (received_info(thread_info1))(thread) = ri1
                 AND ri_status(ri2) = ri_processed
                   AND received_info(thread_info2)
                     = received_info(thread_info1) WITH [thread := ri2])

  kst, kst1, kst2: VAR KERNEL_SHARED_STATE

  c_ags: VAR setof[THREAD]

  kernel_req: VAR KERNEL_REQ

  requested_op, op: VAR OP

  name, reply_port: VAR NAME

  user_msg, user_msg1, user_msg2: VAR USER_MSG

  null_thread_info: THREAD_INFO

  kst_to_ti(kst): THREAD_INFO =
    null_thread_info
      WITH [existing_threads := existing_threads(kst),
            received_info := received_info(kst),
```

```
              thread_status := thread_status(kst)]

effects_on_kernel_state(kst1, kst2, c_ags): bool =
   ((pending_requests(kst2) /= pending_requests(kst1)
           OR existing_threads(kst2) /= existing_threads(kst1)
             OR thread_status(kst2) /= thread_status(kst1))
          IMPLIES
        (EXISTS kernel_req, thread:
           pending_requests(kst2)
                = union(pending_requests(kst1), {x: KERNEL_REQ | kernel_req = x})
               AND
             ((send_message_req?(kernel_req) AND smth(kernel_req) = thread)
                  OR
                (receive_message_req?(kernel_req) AND rmth(kernel_req) = thread))
                 AND c_ags(thread)
                   AND existing_threads(kst1)(thread)
                     AND existing_threads(kst2)(thread)
                       AND thread_status(kst1)(thread) = thread_running
                         AND thread_status(kst2)
                           = thread_status(kst1) WITH [thread := thread_waiting]))
         AND
      ((existing_threads(kst2) /= existing_threads(kst1)
             OR received_info(kst2) /= received_info(kst1))
            IMPLIES
          (EXISTS thread:
             c_ags(thread)
                 AND process_request(thread, kst_to_ti(kst1), kst_to_ti(kst2))))

service_ports: VAR setof[NAME]

needed_perm: VAR PERMISSION

ssi, osi: VAR SID

reply_op(requested_op): OP

make_service_request(c_ags, service_ports, reply_port,
                              requested_op, needed_perm,
                              user_msg1, user_msg2, kst1, kst2):
     bool =
   (EXISTS thread, ri, ssi, osi, kernel_req:
      c_ags(thread)
          AND existing_threads(kst1)(thread)
            AND process_request(thread, kst_to_ti(kst1), kst_to_ti(kst2))
              AND thread_status(kst1)(thread) = thread_running
                AND ri = received_info(kst1)(thread)
                  AND service_ports(service_port(ri))
                    AND ri_status(ri) = ri_unprocessed
                      AND sending_av(ri)(needed_perm)
                        AND op(ri) = requested_op
                          AND user_msg1 = user_msg(ri)
                            AND kernel_req
                                =
                            send_message_req(thread, reply_name(ri),
                                                  reply_op(op(ri)),
                                                  reply_port, user_msg2)
                              AND pending_requests(kst2)
                                =
                              union(pending_requests(kst1),
                                     {x: KERNEL_REQ | x = kernel_req})
                                AND existing_threads(kst2)
                                  = existing_threads(kst1)
                                  AND existing_threads(kst2)(thread)
                                    AND thread_status(kst2)
```

```
                              = thread_status(kst1)
                              WITH [thread := thread_waiting])

 END dtos_kernel_shared_ops
```

The theory $messaging$ defines some utility functions for use by kernel client specifications in receiving and sending messages. The functions are:

- $receive\_msg(kst_1, kst_2, thread, name)$ — $thread$ requests to receive a message on a port name $name$ changing its kernel shared state from $kst_1$ to $kst_2$.
- $receive\_request(thread, ri, op\_id, perm, kst_1, kst_2)$ — $thread$ checks permission $perm$ and operation $op\_id$ on the received information in $ri$ and then uses $process\_request$ to mark $ri$ as processed.
- $send\_msg(kst_1, kst_2, thread, to, op\_id, reply\_port, msg)$ — $thread$ sends message $(op\_id, msg)$ to port $to$ with reply port $reply\_port$.

## THEORY *messaging*

```
messaging: THEORY
 BEGIN

 IMPORTING dtos_kernel_shared_ops

 kst1, kst2: VAR KERNEL_SHARED_STATE

 thread: VAR THREAD

 ri: VAR RECEIVED_INFO

 op_id: VAR OP

 perm: VAR PERMISSION

 name, reply_port, to: VAR NAME

 kernel_req: VAR KERNEL_REQ

 msg: VAR USER_MSG

 %% "Thread" requests to receive a message on a port named "name"
 %% in transition from kst1 to kst2.
 receive_msg(kst1, kst2, thread, name): bool =
   existing_threads(kst1)(thread)
      AND existing_threads(kst1) = existing_threads(kst2)
            AND received_info(kst1) = received_info(kst2)
              AND thread_status(kst1)(thread) = thread_running
              AND existing_threads(kst2)(thread)
                AND thread_status(kst2)
                  = thread_status(kst1) WITH [thread := thread_waiting]
                  AND pending_requests(kst2)
                    =
```

```
                    add(receive_message_req(thread, name),
                        pending_requests(kst1))

  % process a newly received request message
  receive_request(thread, ri, op_id, perm, kst1, kst2): bool =
    existing_threads(kst1)(thread)
            AND thread_status(kst1)(thread) = thread_running
              AND ri = received_info(kst1)(thread)
                AND ri_status(ri) = ri_unprocessed
                  AND sending_av(ri)(perm)
                    AND op(ri) = op_id
                      AND
                    process_request(thread,
                                    kst_to_ti(kst1), kst_to_ti(kst2))

  %% "Thread" sends message <op_id, msg> to port "to" with
  %%    reply port "reply_port" in transition from kst1 to kst2.
  send_msg(kst1, kst2, thread, to, op_id, reply_port, msg): bool =
    existing_threads(kst1) = existing_threads(kst2)
        AND existing_threads(kst1)(thread)
          AND thread_status(kst1)(thread) = thread_running
            AND existing_threads(kst2)(thread)
              AND thread_status(kst2)
                = thread_status(kst1) WITH [thread := thread_waiting]
                AND
              (LET
               kernel_req = send_message_req(thread,
                                             to, op_id,
                                             reply_port, msg)
                 IN pending_requests(kst2)
                      = add(kernel_req, pending_requests(kst1)))

  send_msg_ops_neq : LEMMA
    (send_msg(kst1, kst2, thread, to, op_id, reply_port, msg)
        AND send_message_req?(kernel_req)
        AND smop(kernel_req) /= op_id
        AND pending_requests(kst2)(kernel_req))
    IMPLIES pending_requests(kst1)(kernel_req)

  receive_msg_not_send : LEMMA
    (receive_msg(kst1, kst2, thread, name)
        AND send_message_req?(kernel_req)
        AND pending_requests(kst2)(kernel_req))
    IMPLIES pending_requests(kst1)(kernel_req)

  END messaging
```

The environment of a component $C$ is assumed to obey the following constraints. Note that these constraints will be used in defining the *rely* of $C$ and therefore apply only to agents that do not belong to $C$. The parameters $kst_1$ and $kst_2$ will be instantiated with the local $KERNEL\_SHARED\_STATE$ of $C$.

- Only the kernel may change the set of existing threads (function $existing\_threads\_rely$).
- The status of a thread in $C$ may not be changed from $thread\_running$ to $thread\_waiting$, and only the kernel may change it from $thread\_waiting$ to $thread\_running$ (function $thread\_status\_rely$).

- The $received\_info$ of a thread in $C$ may not be modified unless it has been marked as processed and the agent of the transition is a kernel thread (function $received\_info\_rely$).
- Only a kernel thread may remove a request from $pending\_requests$ and no other component can submit a request in the name of $C$ (function $pending\_requests\_rely$).

We also define here a common base for $hidd$. It states that only a kernel thread may modify the kernel shared state of a component. Furthermore, the kernel never adds a request to the $pending\_requests$ of the component. The relationship between $hidd\_base$ and $environment\_base$ is described by the theorems $hidd\_base\_prop$ and $hidd\_env\_prop$.

## THEORY *dtos_kernel_shared_rely*

```
dtos_kernel_shared_rely: THEORY
  BEGIN

  IMPORTING dtos_kernel_shared_state

  IMPORTING kst_merge

  c_ags: VAR setof[THREAD]

  thread, ag: VAR THREAD

  kst, kst1, kst2: VAR KERNEL_SHARED_STATE

  ri: VAR RECEIVED_INFO

  kernel_req: VAR KERNEL_REQ

  %% The following assume a context where the kst's are local
  %% to a single component.  Thus, although other components may
  %% alter their own kst's there is nothing they can do (with the
  %% exception of the kernel) to the kst of the component being
  %% defined.

  existing_threads_rely(ag, kst1, kst2): bool =
    existing_threads(kst1) = existing_threads(kst2)
    OR k_threads(ag)


  thread_status_rely(ag, kst1, kst2): bool =
    (FORALL thread:
       (existing_threads(kst1)(thread) AND existing_threads(kst2)(thread))
        IMPLIES
         (thread_status(kst1)(thread) = thread_running
                IMPLIES thread_status(kst2)(thread) = thread_running)
        AND
        (thread_status(kst1)(thread) = thread_waiting
             AND thread_status(kst2)(thread) = thread_running
             IMPLIES k_threads(ag)))

  received_info_rely(ag, kst1, kst2): bool =
    (FORALL thread, ri:
       existing_threads(kst1)(thread)
            AND received_info(kst1)(thread) = ri
               AND existing_threads(kst2)(thread)
            IMPLIES
          (received_info(kst2)(thread) = ri
```

```
              OR (ri_status(ri) = ri_processed
                        AND k_threads(ag))))

pending_requests_rely(ag, kst1, kst2): bool =
  (FORALL kernel_req:
     (pending_requests(kst1)(kernel_req)
          AND NOT pending_requests(kst2)(kernel_req)
          IMPLIES k_threads(ag))
         AND
     (pending_requests(kst2)(kernel_req)
           => pending_requests(kst1)(kernel_req))
      )

environment_base(ag, kst1, kst2): bool =
  existing_threads_rely(ag, kst1, kst2)
  AND thread_status_rely(ag, kst1, kst2)
      AND received_info_rely(ag, kst1, kst2)
        AND pending_requests_rely(ag, kst1, kst2)

environment_base_refl: THEOREM
  environment_base(ag, kst1, kst1)


hidd_base(ag, kst1, kst2): bool =
  (kst1 = kst2 OR k_threads(ag))
  AND (FORALL kernel_req:
         (pending_requests(kst2)(kernel_req)
           => pending_requests(kst1)(kernel_req))
      )

hidd_base_prop: THEOREM
  environment_base(ag, kst1, kst2)
      => hidd_base(ag, kst1, kst2)

hidd_env_prop: THEOREM
  hidd_base(ag, kst1, kst2)
      AND (FORALL thread:
             (existing_threads(kst1)(thread) AND existing_threads(kst2)(thread))
              IMPLIES
               (thread_status(kst1)(thread) = thread_running
                      IMPLIES thread_status(kst2)(thread) = thread_running))
      AND (FORALL thread, ri:
             existing_threads(kst1)(thread)
                 AND received_info(kst1)(thread) = ri
                   AND existing_threads(kst2)(thread)
                 IMPLIES
               (received_info(kst2)(thread) = ri
                     OR ri_status(ri) = ri_processed))
  => environment_base(ag, kst1, kst2)

END dtos_kernel_shared_rely
```

# Section 19 Security Server

This section describes the DTOS Security Server. The role of the Security Server is to provide an interpretation of SIDs as persistent security contexts and to perform security computations on SIDs.

## 19.1 State

The Security Server for each node manages a set, $valid\_sids$, of security identifiers representing the valid security contexts for the node. This set is partitioned into $valid\_object\_sids$ and $valid\_subject\_sids$. The Security Server maps each SID to a class denoted by $sid\_class(sid)$. One class is $subject\_class$. For each SID having class $subject\_class$, the Security Server records a domain and a level, $sid\_to\_ssc(sid)$, for the SID. For all other valid SIDs, the Security Server records a type and a level for the SID, $sid\_to\_osc(sid)$.

The Security Server maintains a Domain Definition Table (DDT) that indicates the accesses recorded for each domain-type pair. In the implementation, each set of accesses is specified as a collection of four access vectors (sets of permissions). These four vectors represent each of the possible results of the MLS comparison (levels are the same, subject level is strictly higher, object level is strictly higher, or the levels are incomparable). Here, we simply use $ddt(dmn, lvl_1, typ, lvl_2)$ to denote the access vector indicating the permissions a subject in domain $dmn$ and at level $lvl_1$ has to an object with type $typ$ at level $lvl_2$.

The Security Server maintains a cache of interpretations of AIDs as users received from the Authentication Server. The expression $cached\_aids$ denotes the set of AIDs for which an interpretation has been cached, and $aid\_to\_user(aid)$ denotes the user associated with $aid$ in the cache. To service a request for a security computation that depends on the AIDs, the Security Server must map the AIDs associated with the relevant SIDs to users; the AID-relevant security computations in the Security Server are defined in terms of users. The Security Server maintains a set of known users denoted by $known\_user$. It records the allowable security contexts for each known user. In particular, each user has an associated set of allowable domains, $allowed\_domains(user)$, and an associated set of allowable levels, $allowed\_levels(user)$. When a new task is created, it typically inherits its user from that of the creating subject. However, certain domains are privileged to set the user associated with the new task. We denote this set by $ccu\_privileged$.

The Security Server records a set of names, $ss\_service\_ports$, through which it provides service. Similarly, the Security Server records its name for the host name port, $ss\_host\_name$, and its name for the port through which it expects to receive reply messages, $ss\_reply\_port$.

We use $ss\_threads$ to denote the constant set of threads executing within the Security Server.[32]

The Security Server state consists of the data structures described above as well as its $KERNEL\_SHARED\_STATE$, $kst$, containing $existing\_threads$, $pending\_requests$, $thread\_status$, and $received\_info$. The valid states are defined by $SS\_STATE$. In a valid state,

[32] DTOS does not require this set of threads to be constant. For convenience, we are using threads to represent agents. Since our framework requires the set of agents associated with a component to be static, we require $ss\_thread$ to be constant.

- *valid_object_sids* is the set of all *valid_sids* with a *sid_class* other than *subject_class*,
- *valid_subject_sids* is the set of all *valid_sids* with a *sid_class* equal to *subject_class*, and
- the existing threads in *kst* are all in *ss_threads*.

## THEORY *security_server_state*

```
security_server_state: THEORY
  BEGIN

  IMPORTING dtos_kernel_shared_ops

  CLASS: NONEMPTY_TYPE

  subject_class: CLASS

  DOMAIN: NONEMPTY_TYPE

  domain_witness: DOMAIN

  TYP: NONEMPTY_TYPE

  typ_witness: TYP

  LEVEL: NONEMPTY_TYPE

  level_witness: LEVEL

  SSC: TYPE = [# dmn: DOMAIN, lvl: LEVEL #]

  ssc_witness: SSC = (# dmn := domain_witness, lvl := level_witness #)

  OSC: TYPE = [# typ: TYP, lvl: LEVEL #]

  osc_witness: OSC = (# typ := typ_witness, lvl := level_witness #)

  AID: NONEMPTY_TYPE

  aid_witness: AID

  USER: NONEMPTY_TYPE

  user_witness: USER

  ss_threads: setof[THREAD]

  ss_threads_nonempty: AXIOM ss_threads /= emptyset

  ss_threads_witness: (ss_threads)

  SS_STATE_BASE:
      TYPE =
        [# valid_sids: setof[SID],
           valid_object_sids: setof[SID],
           valid_subject_sids: setof[SID],
           sid_class: [(valid_sids) -> CLASS],
           sid_to_ssc: [(valid_subject_sids) -> SSC],
           sid_to_osc: [(valid_object_sids) -> OSC],
           sid_to_aid: [(valid_sids) -> AID],
```

```
            ddt: [[DOMAIN, LEVEL, TYP, LEVEL] -> ACCESS_VECTOR],
            cached_aids: setof[AID],
            aid_to_user: [(cached_aids) -> USER],
            known_users: setof[USER],
            allowed_levels: [(known_users) -> setof[LEVEL]],
            allowed_domains: [(known_users) -> setof[DOMAIN]],
            ccu_privileged: setof[DOMAIN],
            ss_service_ports: setof[NAME],
            ss_host_name: NAME,
            ss_reply_port: NAME,
            kst: KERNEL_SHARED_STATE #]

  ssstb: VAR SS_STATE_BASE

  sid: VAR SID

  SS_STATE(ssstb): bool =
    (valid_sids(ssstb)
          = union(valid_object_sids(ssstb), valid_subject_sids(ssstb))
          AND valid_object_sids(ssstb)
            =
          (LAMBDA sid:
             valid_sids(ssstb)(sid) AND sid_class(ssstb)(sid) /= subject_class)
            AND valid_subject_sids(ssstb)
              =
            (LAMBDA sid:
               valid_sids(ssstb)(sid)
                      AND sid_class(ssstb)(sid) = subject_class)
              AND subset?(existing_threads(kst(ssstb)), ss_threads))

  sst1, sst2: VAR (SS_STATE)

  ss_threads_prop: THEOREM
   subset?(existing_threads(kst(sst1)), ss_threads)

  ss_view(sst1,sst2) : bool =
    sst1 = sst2

  sss: VAR (SS_STATE)

  valid_subject_sid_def: LEMMA
         valid_subject_sids(sss)(sid)
             = (valid_sids(sss)(sid) AND sid_class(sss)(sid) = subject_class)

  valid_object_sid_def: LEMMA
         valid_object_sids(sss)(sid)
             = (valid_sids(sss)(sid) AND sid_class(sss)(sid) /= subject_class)

  END security_server_state
```

---

# THEORY *security_server_state_witness*

---

```
security_server_state_witness: THEORY

BEGIN

  IMPORTING security_server_state
```

```
ss_state_witness: (SS_STATE) =
  (# valid_sids := emptyset[SID],
     valid_object_sids := emptyset[SID],
     valid_subject_sids := emptyset[SID],
     sid_class := (LAMBDA (x: (emptyset[SID])): subject_class),
     sid_to_ssc := (LAMBDA (x: (emptyset[SID])): ssc_witness),
     sid_to_osc := (LAMBDA (x: (emptyset[SID])): osc_witness),
     sid_to_aid := (LAMBDA (x: (emptyset[SID])): aid_witness),
     ddt :=
       (LAMBDA (x: [DOMAIN, LEVEL, TYP, LEVEL]):
          emptyset[PERMISSION]),
     cached_aids := emptyset[AID],
     aid_to_user := (LAMBDA (x: (emptyset[AID])): user_witness),
     known_users := emptyset[USER],
     allowed_levels := (LAMBDA (x: (emptyset[USER])): emptyset[LEVEL]),
     allowed_domains := (LAMBDA (x: (emptyset[USER])): emptyset[DOMAIN]),
     ccu_privileged := emptyset[DOMAIN],
     ss_service_ports := emptyset[NAME],
     ss_host_name := null_name,
     ss_reply_port := null_name,
     kst := empty_kst
     #)

ss_state_witness_prop: THEOREM (EXISTS (ssstb: (SS_STATE))): TRUE)

END security_server_state_witness
```

## 19.2 Operations

This section describes the subset of Security Server transitions that are relevant to this example. The following transitions are defined:

- *ss_receive_request* — submit a kernel request to receive a message on a service port,
- *ss_compute_access* — receive a request for an access vector, determine the vector and send it in a reply message, and
- *ss_load_user* — receive a message containing the user associated with a given AID and store the user in the cache.

*Editorial Note:*
This section currently describes only successful processing of requests.

The *valid_sids*, *sid_class*, *valid_subject_sids*, *sid_to_ssc*, *valid_object_sids*, *sid_to_osc*, *known_users*, *allowed_levels*, *sid_to_aid*, *ddt*, *allowed_domains*, *ccu_privileged*, *ss_service_ports*, *ss_host_name*, and *ss_reply_port* components of the Security Server state are not altered by any transitions.[33]

At any time when the Security Server has a thread that is not already waiting for a message operation to be performed, that thread can receive a request through a Security Server service port (function *ss_receive_request*). The thread initiates this processing by setting its pending request to be a message receive and changing its state to *thread_waiting*.

[33] This requirement is much more stringent than that for the actual DTOS which allows the policy to be changed.

In response to a **compute_access** request, the Security Server computes the permissions allowed for the specified subject and object SIDs. The result is sent to the kernel in a message.

The expression $allowed(ssi, osi)$ denotes the access vector the Security Server database defines for $ssi$ to $osi$. The major portion of the computation consists of mapping $ssi$ and $osi$ to security contexts and using them to index into $ddt$. In the case of subject creation (permissions $create\_task\_perm$ and $create\_task\_secure\_perm$ on an object of class $subject\_class$) it is also necessary to ensure that the users are the same or the creating domain is in the set $ccu\_privileged$. These are the AID-relevant permissions and thus require checks based on the users.[34] Note that $allowed(ssi, osi)$ is undefined if some permissions for $osi$'s class are AID-relevant and $aid\_to\_user$ does not contain the interpretation of the associated AIDs as users. In this case, the computation of the access vector cannot proceed until $aid\_to\_user$ is updated to contain the necessary information.

---

*Editorial Note:*
The PVS specification is not consistent with the above paragraph since $allowed$ is a total function. This error is irrelevant to the results of the composability study.

---

When it receives a **load_user** request, the Security Server records the indicated binding between an AID and a user.

A Security Server operation consists of any one of the operations defined above. The $guar$ of the Security Server consists of those transitions with a Security Server thread serving as the agent such that the start and final states of the transition satisfy $ss\_step$ and $ss\_op$ or look the same with respect to $ss\_view$.

## THEORY *security_server_ops*

```
security_server_ops: THEORY
  BEGIN

  IMPORTING security_server_state

  st, st1, st2: VAR (SS_STATE)

  ag: VAR THREAD

  ss_static(st1, st2): bool =
    valid_sids(st1) = valid_sids(st2)
        AND sid_class(st1) = sid_class(st2)
          AND valid_subject_sids(st1) = valid_subject_sids(st2)
            AND sid_to_ssc(st1) = sid_to_ssc(st2)
              AND valid_object_sids(st1) = valid_object_sids(st2)
                AND sid_to_osc(st1) = sid_to_osc(st2)
                  AND known_users(st1) = known_users(st2)
                    AND allowed_levels(st1) = allowed_levels(st2)
                      AND sid_to_aid(st1) = sid_to_aid(st2)
                        AND ddt(st1) = ddt(st2)
                          AND allowed_domains(st1) = allowed_domains(st2)
                            AND ccu_privileged(st1) = ccu_privileged(st2)
                              AND ss_service_ports(st1) = ss_service_ports(st2)
                                AND ss_host_name(st1) = ss_host_name(st2)
```

[34] Other permissions besides $create\_task\_perm$ and $create\_task\_secure\_perm$ are AID-relevant in the actual DTOS system.

```
                                        AND ss_reply_port(st1) = ss_reply_port(st2)

ss_step(st1, st2): bool =
  ss_static(st1, st2)
      AND effects_on_kernel_state(kst(st1), kst(st2), ss_threads)

thread: VAR THREAD

name: VAR NAME

ss_receive_request(st1, st2): bool =
  ss_step(st1, st2)
      AND cached_aids(st1) = cached_aids(st2)
        AND aid_to_user(st1) = aid_to_user(st2)
          AND existing_threads(kst(st1)) = existing_threads(kst(st2))
            AND received_info(kst(st1)) = received_info(kst(st2))
              AND
            (EXISTS thread, name:
               ss_threads(thread)
                    AND ss_service_ports(st1)(name)
                      AND existing_threads(kst(st1))(thread)
                        AND thread_status(kst(st1))(thread) = thread_running
                          AND existing_threads(kst(st2))(thread)
                            AND thread_status(kst(st2))
                              = thread_status(kst(st1))
                              WITH [thread := thread_waiting]
                              AND pending_requests(kst(st2))
                                =
                              add(receive_message_req(thread, name),
                                  pending_requests(kst(st1))))

ssi, osi: VAR SID

base_allowed(ssi, osi, st): ACCESS_VECTOR =
  IF (valid_subject_sids(st)(ssi) AND valid_object_sids(st)(osi))
    THEN LET ssc: SSC = sid_to_ssc(st)(ssi), osc: OSC = sid_to_osc(st)(osi)
      IN ddt(st)(dmn(ssc), lvl(ssc), typ(osc), lvl(osc))
  ELSE emptyset[PERMISSION]
  ENDIF

perm: VAR PERMISSION

allowed(ssi, osi, st): ACCESS_VECTOR =
  IF NOT valid_object_sids(st)(osi)
          OR (valid_sids(st)(osi) AND sid_class(st)(osi) /= subject_class)
            OR NOT valid_subject_sids(st)(ssi)
    THEN base_allowed(ssi, osi, st)
  ELSIF (valid_sids(st)(ssi) AND NOT cached_aids(st)(sid_to_aid(st)(ssi)))
          OR
        (valid_sids(st)(osi) AND NOT cached_aids(st)(sid_to_aid(st)(osi)))
    THEN emptyset[PERMISSION]
  ELSIF aid_to_user(st)(sid_to_aid(st)(ssi))
          /= aid_to_user(st)(sid_to_aid(st)(osi))
          AND NOT ccu_privileged(st)(dmn(sid_to_ssc(st)(ssi)))
    THEN
    (LAMBDA perm:
       base_allowed(ssi, osi, st)(perm)
           AND perm /= create_task_perm AND perm /= create_task_secure_perm)
  ELSE base_allowed(ssi, osi, st)
  ENDIF

compute_access_op: OP

compute_access_perm: PERMISSION
```

```
sid_sid_to_data: [[SID, SID] -> DATA]

sid_sid_av_to_data: [[SID, SID, ACCESS_VECTOR] -> DATA]

av: VAR ACCESS_VECTOR

compute_access_msg(ssi, osi): USER_MSG =
  null_user_msg
    WITH [user_data := sid_sid_to_data(ssi, osi), user_rights := null_seq]

provide_access_msg(ssi, osi, av): USER_MSG =
  null_user_msg
    WITH [user_data := sid_sid_av_to_data(ssi, osi, av),
          user_rights := null_seq]

ss_compute_access(st1, st2): bool =
  ss_step(st1, st2)
    AND
    (EXISTS ssi, osi:
      (NOT valid_sids(st1)(ssi) OR cached_aids(st1)(sid_to_aid(st1)(ssi)))
      AND (NOT valid_sids(st1)(osi)
               OR cached_aids(st1)(sid_to_aid(st1)(osi)))
      AND make_service_request(ss_threads, ss_service_ports(st1),
                               ss_reply_port(st1),
                               compute_access_op, compute_access_perm,
                               compute_access_msg(ssi, osi),
                               provide_access_msg(ssi, osi,
                                                  allowed(ssi, osi, st1)),
                               kst(st1), kst(st2)))
        AND cached_aids(st1) = cached_aids(st2)
          AND aid_to_user(st1) = aid_to_user(st2)

load_user_op: OP

load_user_perm: PERMISSION

aid_user_to_data: [[AID, USER] -> DATA]

aid, aid1: VAR AID

user: VAR USER

load_user_msg(aid, user): USER_MSG =
  null_user_msg
    WITH [user_data := aid_user_to_data(aid, user), user_rights := null_seq]

ri: VAR RECEIVED_INFO

ss_load_user(st1, st2): bool =
  (EXISTS thread, ri, aid, user:
     ss_threads(thread)
         AND
       process_request(thread, kst_to_ti(kst(st1)), kst_to_ti(kst(st2)))
           AND existing_threads(kst(st2)) = existing_threads(kst(st1))
             AND thread_status(kst(st2)) = thread_status(kst(st1))
               AND existing_threads(kst(st1))(thread)
                 AND thread_status(kst(st1))(thread) = thread_running
                   AND ri = received_info(kst(st1))(thread)
                     AND ss_service_ports(st1)(service_port(ri))
                       AND ri_status(ri) = ri_unprocessed
                         AND sending_av(ri)(load_user_perm)
                           AND op(ri) = load_user_op
                             AND load_user_msg(aid, user) = user_msg(ri)
```

```
                                              AND cached_aids(st2)
                                                = union(cached_aids(st1), {x: AID | x = aid})
                                                AND cached_aids(st2)(aid)
                                                  AND aid_to_user(st2)
                                                    = aid_to_user(st1) WITH [aid := user]
                                                    AND pending_requests(kst(st2))
                                                      = pending_requests(kst(st1)))

  ss_op(st1, st2) : bool =
    ss_receive_request(st1,st2) or
      ss_compute_access(st1,st2) or
      ss_load_user(st1,st2)


  ss_guar(st1,st2,ag) : bool =
    ss_threads(ag) AND
          (ss_view(st1, st2)
            OR (ss_step(st1, st2) AND ss_op(st1, st2)))

  END security_server_ops
```

## 19.3 Environment Assumptions

The environment of the Security Server is assumed to alter no Security Server state information other than *kst* and to obey the constraints on changing *kst* that are given in *environment_base* on page 140. The *hidd* of the Security Server is defined similarly using *hidd_base*.

### THEORY *security_server_rely*

```
security_server_rely : THEORY

BEGIN

IMPORTING dtos_kernel_shared_rely

IMPORTING security_server_state

st1, st2 : VAR (SS_STATE)

ag : VAR THREAD

ss_environment(st1,st2,ag) : bool =
      environment_base(ag,kst(st1),kst(st2)) and
      st1 with [kst := kst(st2)] = st2

ss_environment_refl: THEOREM
      ss_environment(st1,st1,ag)

ss_hidd(st1,st2,ag) : bool =
      NOT ss_threads(ag)
          AND hidd_base(ag, kst(st1), kst(st2))
          AND st2 = st1 WITH [ kst := kst(st2) ]

ss_hidd_prop: THEOREM
      ss_hidd(st1,st2,ag)
          => k_threads(ag) OR ss_view(st1,st2)
```

```
ss_rely(st1,st2,ag) : bool =
    not ss_threads(ag) AND
    ss_environment(st1,st2,ag)

END security_server_rely
```

## 19.4 Component Specification

We use the set $initial\_ss\_states$ to denote the valid initial states for the Security Server. The constraints on initial states are as follows:

- There are no cached AIDs.
- No kernel requests are pending for any Security Server thread and no messages are waiting to be processed.

> *Editorial Note:*
> Additional constraints ought to be placed on the initial states. Examples include:
>
> - at least one Security Server thread is in the running state,
> - the DDT is configured correctly (e.g., allows the kernel to request computations, the Security Server to provide computations, the Authentication Server to load aid-to-user interpretations).
>
> These constraints are not really needed for the example as it appears here, but they would be necessary in a real specification of the Security Server.

All the data in $SS\_STATE\_BASE$ is visible to the Security Server.

A Security Server is a component having state $SS\_STATE$, satisfying initial constraint $initial\_ss\_states$, and executing only the transitions defined in Section 19.2.

### THEORY *security_server_spec*

```
security_server_spec : THEORY
BEGIN

IMPORTING dtos_kernel_shared_state
IMPORTING security_server_ops
IMPORTING security_server_rely
IMPORTING security_server_state_witness
IMPORTING component_aux[(SS_STATE),THREAD]

sst, sst1, sst2 : VAR (SS_STATE)
ag : VAR THREAD

%% This is only a partial definition. In particular we need to place
%% requirements on ddt to ensure that the system components can work
%% together.
initial_ss_states(sst) : bool =
  cached_aids(sst) = emptyset[AID]
    AND pending_requests(kst(sst)) = emptyset[KERNEL_REQ]
    AND (FORALL ag :
```

```
        existing_threads(kst(sst))(ag)  =>
                  ri_status(received_info(kst(sst))(ag))  =  ri_processed)

ss_state_witness_initial: THEOREM
  initial_ss_states(ss_state_witness)

base_ss_comp : base_comp_t =
     (# init := initial_ss_states,
        guar := ss_guar,
        rely := ss_rely,
        hidd := ss_hidd,
        cags := ss_threads,
        view := ss_view,
        wfar := emptyset[TRANSITION_CLASS[(SS_STATE), THREAD]],
        sfar := emptyset[TRANSITION_CLASS[(SS_STATE), THREAD]] #)

  ss_view_eq: THEOREM view_eq(base_ss_comp)

  ss_comp_init: THEOREM init_restriction(base_ss_comp)

  ss_comp_guar: THEOREM guar_restriction(base_ss_comp)

  ss_comp_rely_hidd: THEOREM rely_hidd_restriction(base_ss_comp)

  ss_comp_hidd: THEOREM hidd_restriction(base_ss_comp)

  ss_comp_rely: THEOREM rely_restriction(base_ss_comp)

  ss_comp_cags: THEOREM cags_restriction(base_ss_comp)

  ss_comp_guar_stuttering: THEOREM guar_stuttering_restriction(base_ss_comp)

  ss_comp_rely_stuttering: THEOREM rely_stuttering_restriction(base_ss_comp)

  ss_comp : (comp_t) = base_ss_comp

  ss_comp_hidd_prop: THEOREM
    hidd(ss_comp)(sst1, sst2, ag)
    => k_threads(ag) OR view(ss_comp)(sst1, sst2)

END security_server_spec
```

*Editorial Note:*
The Z version of this specification includes the proofs of several properties of the Security Server. These properties have not been translated and proven in PVS.

*Section* 20

# Overview of the Cryptographic Subsystem

The Cryptographic Subsystem enables clients to encrypt information either to be sent across a network or used internal to the system (e.g., written to encrypted media). The encryption and key generation mechanisms are determined by a security service usage policy. There are several components that are part of (or are closely related to) the Cryptographic Subsystem. They include the following:

**Security Service Usage Policy Server (SSUPS)** — Contains the security service usage policy. Other components request decisions from this server regarding the encryption algorithms and key mechanisms that may be used in particular situations. This component serves an analogous function for the Cryptographic Subsystem as the security server does for the node as a whole by making policy decisions that are enforced by separate policy-neutral servers.

**Cryptographic Controller (CC)** — Coordinates the actions of the other components of the Cryptographic Subsystem. It receives requests for cryptographic contexts, sends requests to other components of the subsystem in a attempt to create the context and, if the context is successfully created, returns a port right that may be used to encrypt information according to the context.

**Protection Tasks (PT)** — Encrypt and/or sign data according to particular algorithms. The subsystem will typically contain numerous protection tasks. A single cryptographic context will typically involve a sequence of protection tasks invoked in some fixed order.

**Key Servers (KS)** — Supply keys for use by the protection tasks. The subsystem will typically contain numerous key servers, and each cryptographic context will typically use several of them.

**Clients** — A client could be one of many types of components including a driver for encrypted media, a network server or a negotiation server (a system component that negotiates with its peer on another node to agree on the encryption and key mechanisms to be used). We will only model a very abstract client.

The basic operation of the system is as follows:

- A client sends a request to the SSUPS to use a particular sequence of protection mechanisms (encryption and keying algorithms) for a given cryptographic situation.
- If the mechanisms are acceptable to the SSUPS in the situation, it returns a handle (port) to the client.
- The client sends the handle in a request to the CC to initialize a cryptographic context.
- The CC sends a message to the handle asking for the sequence of protection mechanisms.
- The SSUPS sends the sequence in a reply message.

- The CC uses this information to send requests to key servers to get handles for obtaining keys.
- The handles are used, together with the sequence of mechanisms, to assemble a request to the first protection task to begin initializing a pipeline of protection tasks to implement the sequence of mechanisms.
- The protection tasks communicate to establish the pipeline. They also use the key server handles to obtain keys. When done, the first protection task returns a handle to the CC representing the start of the pipeline.
- The CC forwards the handle in a reply to the client.
- The client now sends protection requests to the handle.
- Each protection request is sent through the pipeline with each protection task doing its job.
- The final protection task in the pipeline sends the protected data in a reply message to the client.
- The client receives the reply message and stores the protected data. (We have left unspecified what the client might actually do with the protected data. In a specification of a particular type of client we could describe additional processing.)

To simplify the specification and analysis, we have ignored some of the flexibility of the Cryptographic Subsystem. For example, the CC allows the following two types of context creation requests:

- Those where an SSUPS handle is provided (described above).
- Those where no SSUPS handle is provided. This indicates that approval for the protection family has not yet been obtained from the SSUPS. The CC must determine an appropriate protection family and obtain approval before establishing the context.

We will consider only the first path.

There are several data types that are used by multiple components of the Cryptographic Subsystem, and we define them here. A value of type $SITUATION$ denotes the information that is relevant for determining how particular data needs to be protected. A situation is the input in a policy request to the SSUPS. It may include the security context of the client, the destination address if the information is to be sent over the network and the type of socket (i.e., stream, datagram). For our purposes we simply define a given type $SITUATION$. A value of type $PROTECTION$ is an association between an encryption mechanism, a key mechanism and a security protocol.[35] A value of type $PROT\_FAMILY$ is a sequence of protections and a value of type $PROT\_FAMILY\_SEQ$ is a sequence of protection families. The output of the SSUPS is a $PROT\_FAMILY\_SEQ$. The functions $sit\_pf\_to\_data$ and $pf\_to\_data$ return the representations of situations and protection families as message data. The requests that the various components service are declared here, but we will discuss them in later sections when we specify their behaviors.

[35] For the components of the subsystem specified in this report we only need the first two parts of this association.

---

*Editorial Note:*
While preparing the final version of this report a fairly pervasive error was discovered in the sending of messages in this example. The kernel specification requires that a send right be explicitly included in the list of transferred rights for any reply port supplied in the message. Most of the transitions below that send a message neglect to do this. In a complete analysis it is possible that such an oversight could mean that the safety properties proved really only hold because the processing "dies" as soon as a reply is sent (the reply name might not be in the replier's name space). However, this is unlikely. To ensure that this is not the case, analysts could state and prove "sanity check" properties such as "if a message with a non-NULL reply name is received, then the receiver must have send rights to the reply name. In any case, this is all irrelevant to the composition issues being explored in this study.

---

## THEORY *crypto_shared_state*

```
crypto_shared_state: THEORY
  BEGIN

  IMPORTING dtos_kernel_shared_state
  IMPORTING fseq_functions
  IMPORTING more_set_lemmas

  SITUATION : NONEMPTY_TYPE

  SEC_PROTOCOL: NONEMPTY_TYPE

  KEY_MECH: NONEMPTY_TYPE

  ENCRYPT_MECH: NONEMPTY_TYPE

  KEY : NONEMPTY_TYPE

  TEXT : NONEMPTY_TYPE

  SEED : NONEMPTY_TYPE

  null_text : TEXT

  key_witness: KEY

  key_mech_witness : KEY_MECH

  encrypt_mech_witness : ENCRYPT_MECH

  generate_key : [KEY_MECH, SEED -> KEY]

  protect_text : [ENCRYPT_MECH, KEY, TEXT -> TEXT]

  % Don't use security protocol at this time
  PROT : TYPE =
     [# key_mech : KEY_MECH,
        encrypt_mech : ENCRYPT_MECH,
        sec_protocol : SEC_PROTOCOL #]

  IMPORTING finite_sequence[PROT]

  PROT_FAMILY : TYPE = FSEQ[PROT]
  null_prot_family : PROT_FAMILY = finite_sequence[PROT].null_seq
```

```
IMPORTING finite_sequence[PROT_FAMILY]

PROT_FAMILY_SEQ: TYPE = FSEQ[PROT_FAMILY]
null_prot_family_seq : PROT_FAMILY_SEQ = finite_sequence[PROT_FAMILY].null_seq


sit_pf_to_data: [[SITUATION, PROT_FAMILY] -> DATA]

pf_to_data: [PROT_FAMILY -> DATA]

key_to_data: [KEY -> DATA]

text_to_data: [TEXT -> DATA]

k1, k2 : VAR KEY

key_to_data_inj: AXIOM
      key_to_data(k1) = key_to_data(k2)
            IMPLIES k1 = k2



x,n : VAR posnat

null_name_seq(x) : NAME_SEQ

null_name_seq_ax : AXIOM
      size(null_name_seq(x)) = x
      AND (FORALL n : (n > 0 and n <= size(null_name_seq(x)))
            => elem(null_name_seq(x))(n) = null_name)


prot_family: VAR PROT_FAMILY
sit: VAR SITUATION
name: VAR NAME
name_seq : VAR NAME_SEQ
data: VAR DATA
text: VAR TEXT
dest: VAR NAME
key : VAR KEY

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% SSUPS ops

% Request port indicating permission to a given prot_family
select_prot_family_op : OP
select_prot_family_perm : PERMISSION

select_prot_family_msg(sit, prot_family): USER_MSG =
      null_user_msg
        WITH [ (user_data) := sit_pf_to_data(sit, prot_family) ]

provide_pf_handle_op : OP
provide_pf_handle_perm : PERMISSION

provide_pf_handle_msg(name): USER_MSG =
      null_user_msg
        WITH [ (user_rights) := name_to_send_right_seq(name) ]
```

```
% Retrieve a previously negotiated prot_family

%% a retrieve_prot_family_msg = null_msg, so we do not need to declare it.

retrieve_prot_family_op : OP
retrieve_prot_family_perm : PERMISSION


% Reply with the previously negotiated prot_family
provide_prot_family_msg(prot_family): USER_MSG =
        null_user_msg
          WITH [ (user_data) := pf_to_data(prot_family) ]

provide_prot_family_op : OP
provide_prot_family_perm : PERMISSION

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% CC ops

%% A client requests a crypto context
create_crypto_context_msg(sit, name, prot_family): USER_MSG =
        (# user_data := sit_pf_to_data(sit, prot_family),
           user_rights := name_to_send_right_seq(name) #)


create_crypto_context_op : OP
create_crypto_context_perm : PERMISSION


%% Reply from create_crypto_context with a handle for a protection task.
provide_crypto_context_msg(name): USER_MSG =
        null_user_msg
          WITH [ (user_rights) := name_to_send_right_seq(name) ]


provide_crypto_context_op : OP
provide_crypto_context_perm : PERMISSION

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% PROTECTION TASK OPS

init_crypto_context_msg(name_seq): USER_MSG =
     null_user_msg
        WITH [ (user_rights) := map(name_to_send_right, name_seq) ]


init_crypto_context_op : OP
init_crypto_context_perm : PERMISSION


provide_crypto_handle_msg(name): USER_MSG =
        null_user_msg
          WITH [ (user_rights) := name_to_send_right_seq(name) ]


provide_crypto_handle_op : OP
provide_crypto_handle_perm : PERMISSION


protect_msg(text, dest): USER_MSG =
   (# user_data := text_to_data(text),
        user_rights := name_to_send_right_seq(dest) #)
```

```
  protect_op : OP
  protect_perm : PERMISSION


  provide_protected_data_msg(text): USER_MSG =
    null_user_msg
          WITH [ (user_data) := text_to_data(text) ]

  provide_protected_data_op : OP
  provide_protected_data_perm : PERMISSION

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


  % KEY SERVER OPS

  %% init_key_retrieval_msg = null_msg
  init_key_retrieval_op : OP
  init_key_retrieval_perm : PERMISSION


  provide_key_port_msg(name): USER_MSG =
        null_user_msg
          WITH [ (user_rights) := name_to_send_right_seq(name) ]


  provide_key_port_op : OP
  provide_key_port_perm : PERMISSION

  retrieve_key_op : OP
  retrieve_key_perm : PERMISSION

  provide_key_op : OP
  provide_key_perm : PERMISSION

  provide_key_msg(key): USER_MSG =
        null_user_msg
          WITH [ (user_data) := key_to_data(key) ]


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  %% Specify which OPs must be distinct. In general these constraints
  %% are NOT necessary in all implementations of the model since any
  %% two distinct operations frequently go to distinct servers.
  %% However, these assumptions simplify the proofs.

  ks_provide_ops_distinct: AXIOM
        provide_key_port_op /= provide_key_op


  op1, op2: VAR OP
  f : VAR [OP -> int]

  %% Make distinctness proofs easier
  self_congruence : LEMMA
    op1 = op2 IMPLIES f(op1) = f(op2)


  %% Define a function from some ops to distinct integers.
  %% Distinctness follows from the fact that it is a function.
  %% See 'congruence' in the prelude.
```

```
cc_op_num : [OP -> int]

cc_op_ax : AXIOM
  cc_op_num(retrieve_prot_family_op) = 1
  AND cc_op_num(init_crypto_context_op) = 2
  AND cc_op_num(init_key_retrieval_op) = 3
  AND cc_op_num(provide_crypto_context_op) = 4


cc_ops_distinct: LEMMA
  retrieve_prot_family_op /= init_crypto_context_op
  AND retrieve_prot_family_op /= init_key_retrieval_op
  AND retrieve_prot_family_op /= provide_crypto_context_op
  AND init_crypto_context_op /= init_key_retrieval_op
  AND init_crypto_context_op /= provide_crypto_context_op
  AND init_key_retrieval_op /= provide_crypto_context_op

END crypto_shared_state
```

*Section* 21

# Cryptographic Controller

This section describes the Synergy Cryptographic Controller (CC). The role of the CC is to coordinate the actions of the other components of the Cryptographic Subsystem. It receives requests for cryptographic contexts and, if the context is successfully created, returns a port right that may be used to encrypt information according to the context.

## 21.1 State

The CC for each node manages a set *active_ccc* of elements of type *RECEIVED_INFO* denoting the set of cryptographic context creation requests that are active. An *active_ccc* may have a protection family associated with it by the function *ccc_prot_family*. When a context has been successfully created, a port representing the context is associated with the *active_ccc* by the function *ccc_handle*.

The CC also maintains several port names for use in its protocol:

- *ssups* is a name for the port that the CC uses to send requests to the Security Service Usage Policy Server (SSUPS).
- *service_ports* is a set of ports upon which the CC will receive requests for creation of cryptographic contexts.
- *avail_port* is a set of ports that the CC may use as reply ports when making requests of other Cryptographic Subsystem components.
- *retrieve_pf_port* is a set of ports on which the CC is expecting a reply from the SSUPS containing a protection family. *pending_retrieve_pf* denotes the *active_ccc* to be associated with the reply when received.
- *key_init_port* is a set of ports on which the CC is expecting a reply from a key server containing a port to use in requesting keys for the given context. *pending_key_init* denotes the *active_ccc* and the element of the protection family associated with that *active_ccc* that are to be associated with the reply when received.
- *context_port* is a set of ports on which the CC is expecting a reply from a protection task containing a port to use for requesting encryptions within the given context. *pending_context_port* denotes the *active_ccc* to be associated with the reply when received.
- *ccc_init_cc_args*(*ac*) denotes the key ports that have been obtained at any given time for use in the context being created for the *active_ccc*, *ac*.

A protection family specifies key and encryption mechanisms that must be used when the family is applied within a context. The CC maintains mappings *key_mech_server* and *encrypt_mech_server* from these mechanisms to the ports that it will use in establishing a context that uses the mechanisms.

We use $cc\_threads$ to denote the constant set of threads executing within the CC.[36]

The CC state consists of the data structures described above as well as its $KERNEL\_SHARED\_STATE$, $kst$, containing $existing\_threads$, $pending\_requests$, $thread\_status$, and $received\_info$. The valid states are defined by $CC\_STATE$. In a valid state,

- the sets $retrieve\_pf\_port$, $key\_init\_port$, $context\_port$ and $service\_ports$ are each disjoint from $avail\_port$,
- for each $ccc$ the size of the $ccc\_init\_cc\_args$ for $ccc$ is the same as the size of the protection family for $ccc$, and
- the existing threads in $kst$ are all in $cc\_threads$.

All the data in $CC\_STATE$ is visible to the CC.

## THEORY *cc_state*

```
cc_state : THEORY
  BEGIN

  IMPORTING crypto_shared_state

  cc_threads: setof[THREAD]

  cc_threads_nonempty: AXIOM cc_threads /= emptyset

  cc_threads_witness: (cc_threads)

  CC_STATE_BASE: TYPE =
     [# % requests I am processing.  Index by this rather than situation
        % because two clients with the same situation could request
        % different preferred prot families.
        active_ccc: setof[RECEIVED_INFO],

        ssups: NAME,                         % where I send ssups requests
        service_ports: setof[NAME],          % where I receive my requests

        avail_port: setof[NAME],             % my supply of reply ports


        % ports on which I'm expecting a selected protection family
        % from ssups
        retrieve_pf_port: setof[NAME],
        pending_retrieve_pf: [(retrieve_pf_port) -> RECEIVED_INFO],

        %the selected family; only used for active_ccc's
        ccc_prot_family: [RECEIVED_INFO -> PROT_FAMILY],

        %the obtained crypto context handle
        ccc_handle: [RECEIVED_INFO -> NAME],

        % ports on which I'm expecting a key port from a key server
        key_init_port: setof[NAME],
        pending_key_init: [(key_init_port) -> [RECEIVED_INFO, posnat]],
```

[36] As with $ss\_threads$ this is not a requirement on an implementation of the CC but rather a convenience given that our framework requires the set of agents associated with a component to be static.

```
        % Key ports received so far; only used for active_ccc's
        ccc_init_cc_args: [RECEIVED_INFO -> NAME_SEQ],

        % ports on which I'm expecting a crypto_context port from a
        % protection task
        context_port: setof[NAME],
        pending_context_port: [(context_port) -> RECEIVED_INFO],

        % mapping mechanisms to port names
        key_mech_server: [KEY_MECH -> NAME],
        encrypt_mech_server: [ENCRYPT_MECH -> NAME],

        kst: KERNEL_SHARED_STATE
        #]

  ccstb : VAR CC_STATE_BASE

  ccc : VAR RECEIVED_INFO

  CC_STATE(ccstb): bool =
    disjoint?(avail_port(ccstb), retrieve_pf_port(ccstb))
        AND disjoint?(avail_port(ccstb), key_init_port(ccstb))
        AND disjoint?(avail_port(ccstb), context_port(ccstb))
        AND disjoint?(avail_port(ccstb), service_ports(ccstb))
        AND (FORALL ccc: (active_ccc(ccstb)(ccc)) IMPLIES
                 size(ccc_prot_family(ccstb)(ccc))
                   = size(ccc_init_cc_args(ccstb)(ccc)))
        AND subset?(existing_threads(kst(ccstb)), cc_threads)

  st1, st2: VAR (CC_STATE)

  cc_threads_prop: THEOREM
   subset?(existing_threads(kst(st1)), cc_threads)

  cc_view(st1,st2) : bool =
    st1 = st2

  END cc_state
```

# THEORY *cc_state_witness*

```
cc_state_witness: THEORY

BEGIN

  IMPORTING cc_state


  cc_state_witness: (CC_STATE) =
    (# active_ccc := emptyset[RECEIVED_INFO],
       ssups := null_name,
       service_ports := emptyset[NAME],
       avail_port := emptyset[NAME],
       retrieve_pf_port := emptyset,
       pending_retrieve_pf := (LAMBDA (x: (emptyset[NAME])): ri_witness),
       ccc_prot_family := (LAMBDA (x: RECEIVED_INFO): null_prot_family),
       ccc_handle := (LAMBDA (x: RECEIVED_INFO): null_name),
       key_init_port := emptyset,
```

```
    pending_key_init := (LAMBDA (x: (emptyset[NAME])): (ri_witness, 1)),
    ccc_init_cc_args := (LAMBDA (x: RECEIVED_INFO): null_name_seq(1)),
    context_port := emptyset,
    pending_context_port := (LAMBDA (x: (emptyset[NAME])): ri_witness),
    key_mech_server := (LAMBDA (km: KEY_MECH): null_name),
    encrypt_mech_server := (LAMBDA (em: ENCRYPT_MECH): null_name),
    kst := empty_kst
    #)

  cc_state_witness_prop : THEOREM
      (EXISTS (ccstb : (CC_STATE)) : TRUE)

END cc_state_witness
```

## 21.2 Operations

This section describes the subset of CC operations that are relevant to this example.

*Editorial Note:*
This section currently describes only successful processing of requests.

We first define several utility functions:

- *cc_step* defines the components of the CC state that are invariant in all transitions performed by CC agents. This includes *key_mech_server*, *encrypt_mech_server*, *ssups*, *service_ports* and *existing_threads*. Furthermore, when manipulating *kst*, the CC threads are assumed to use the correct protocol as described in *effects_on_kernel_state* in Section 17.1.2.
- *retrieve_pf_inv*, *key_init_inv*, *context_port_inv*, respectively, state that the ports on which the CC is waiting for a protection family, a key server handle or a context handle do not change.
- *receive_request* denotes a state transition where a CC thread has received a request for a CC service and the *sending_av* indicates the sender of the request has permission to make the request. The request message (i.e., *received_info*) is marked as processed.
- *send_msg* denotes a transition in which a thread sends a message.

## THEORY *cc_ops_base*

```
cc_ops_base: THEORY
  BEGIN

  IMPORTING cc_state

  IMPORTING dtos_kernel_shared_ops

  %%This should probably be in dtos_kernel_shared_ops
  IMPORTING messaging

  st, st1, st2: VAR (CC_STATE)
```

```
%%local state invariants
cc_static(st1, st2): bool =
  key_mech_server(st2) = key_mech_server(st1)
      AND encrypt_mech_server(st2) = encrypt_mech_server(st1)
        AND ssups(st2) = ssups(st1)
          AND service_ports(st2) = service_ports(st1)
            AND existing_threads(kst(st2)) = existing_threads(kst(st1))

%a step must obey local invariants and only make allowed
% mods to kernel state.
cc_step(st1, st2): bool =
  cc_static(st1, st2)
      AND effects_on_kernel_state(kst(st1), kst(st2), cc_threads)

retrieve_pf_inv(st1, st2): bool =
  retrieve_pf_port(st2) = retrieve_pf_port(st1)
      AND pending_retrieve_pf(st2) = pending_retrieve_pf(st1)

key_init_inv(st1, st2): bool =
  key_init_port(st2) = key_init_port(st1)
      AND pending_key_init(st2) = pending_key_init(st1)

context_port_inv(st1, st2): bool =
  context_port(st2) = context_port(st1)
      AND pending_context_port(st2) = pending_context_port(st1)

thread: VAR THREAD

prot_family: VAR PROT_FAMILY

ri: VAR RECEIVED_INFO

op_id: VAR OP

perm: VAR PERMISSION

name, reply_port, to: VAR NAME

kernel_req: VAR KERNEL_REQ

ccc: VAR RECEIVED_INFO

msg: VAR USER_MSG


% UTILITY FUNCTIONS

% processing a newly received CC request
receive_request(thread, ri, op_id, perm, st1, st2): bool =
  cc_step(st1, st2)
      AND cc_threads(thread)
        AND existing_threads(kst(st1))(thread)
          AND thread_status(kst(st1))(thread) = thread_running
            AND ri = received_info(kst(st1))(thread)
              AND ri_status(ri) = ri_unprocessed
                AND sending_av(ri)(perm)
                  AND op(ri) = op_id
                    AND
                  process_request(thread,
                                  kst_to_ti(kst(st1)), kst_to_ti(kst(st2)))

%% "Thread" sends message <op_id, msg> to port "to" with
%%    reply port "reply_port" in transition from st1 to st2.
```

```
  send_msg(st1, st2, thread, to, op_id, reply_port, msg): bool =
    send_msg(kst(st1), kst(st2), thread, to, op_id, reply_port, msg)

  END cc_ops_base
```

At any time when the CC has a thread that is not already waiting for a message operation to be performed, that thread can request to receive a message from a port. The thread initiates this processing by setting its pending request to be a message receive and changing its state to $thread\_waiting$.

## THEORY *cc_receive_request*

```
cc_receive_request: THEORY
  BEGIN

  IMPORTING cc_ops_base

  st1, st2: VAR (CC_STATE)

  thread: VAR THREAD

  name: VAR NAME

  cc_receive_request_aux1(st1, st2): bool =
    ccc_handle(st2) = ccc_handle(st1)
        AND ccc_init_cc_args(st2) = ccc_init_cc_args(st1)
          AND avail_port(st2) = avail_port(st1)

  cc_receive_request_aux2(st1, st2): bool =
    retrieve_pf_inv(st1, st2)
        AND key_init_inv(st1, st2) AND context_port_inv(st1, st2)

  cc_receive_request_submit(st1, st2): bool =
    EXISTS thread, name:
      cc_threads(thread)
            AND existing_threads(kst(st1))(thread)
              AND thread_status(kst(st1))(thread) = thread_running
                AND existing_threads(kst(st2))(thread)
                  AND thread_status(kst(st2))
                    = thread_status(kst(st1)) WITH [thread := thread_waiting]
                    AND pending_requests(kst(st2))
                      =
                    add(receive_message_req(thread, name),
                        pending_requests(kst(st1)))

  cc_receive_request(st1, st2): bool =
    cc_step(st1, st2)
        AND active_ccc(st1) = active_ccc(st2)
          AND ccc_prot_family(st1) = ccc_prot_family(st2)
            AND existing_threads(kst(st1)) = existing_threads(kst(st2))
              AND received_info(kst(st1)) = received_info(kst(st2))
                AND cc_receive_request_aux1(st1, st2)
                  AND cc_receive_request_aux2(st1, st2)
                    AND cc_receive_request_submit(st1, st2)

  END cc_receive_request
```

In response to a **create_crypto_context** request containing an SSUPS handle port, the CC

- records the request as a new *active_ccc*,
- assigns an available port as the *retrieve_pf_port* for this request and
- sends a **retrieve_prot_family** request to the SSUPS handle, passing the newly assigned *retrieve_pf_port* as the reply port.

## THEORY *cc_create_context_from_port*

```
cc_create_context_from_port: THEORY
  BEGIN

  IMPORTING cc_ops_base


  % VARIABLES

  st1, st2: VAR (CC_STATE)

  thread: VAR THREAD

  name: VAR NAME

  prot_family : VAR PROT_FAMILY

  sit: VAR SITUATION

  ri: VAR RECEIVED_INFO

  reply_port : VAR NAME

  kernel_req : VAR KERNEL_REQ


  mark_retrieve_pf_port(st1, st2, reply_port, ri) : bool =
    avail_port(st2) = remove(reply_port, avail_port(st1))
        AND retrieve_pf_port(st2) = add(reply_port, retrieve_pf_port(st1))
        AND retrieve_pf_port(st2)(reply_port)
        AND pending_retrieve_pf(st2) = pending_retrieve_pf(st1)
                                    WITH [(reply_port) := ri]

  cc_create_context_from_port(st1, st2): bool =
    (EXISTS thread, ri, sit, name, prot_family, reply_port, kernel_req:

     cc_step(st1, st2)

        AND receive_request(thread, ri, create_crypto_context_op,
                            create_crypto_context_perm, st1, st2)
        AND create_crypto_context_msg(sit, name, prot_family) = user_msg(ri)
        AND service_ports(st1)(service_port(ri))
        AND name /= null_name
        AND prot_family = null_prot_family
        AND avail_port(st1)(reply_port)
```

```
    % This says we should not currently be processing an identical ri.
    %   Is this correct?
    AND NOT active_ccc(st1)(ri)

    AND active_ccc(st2) = add(ri, active_ccc(st1))
    AND existing_threads(kst(st2)) = existing_threads(kst(st1))

    AND ccc_prot_family(st2) = ccc_prot_family(st1)
    AND ccc_handle(st2) = ccc_handle(st1)
    AND ccc_init_cc_args(st2) = ccc_init_cc_args(st1)

    AND key_init_inv(st1, st2)
    AND context_port_inv(st1,st2)

    AND send_msg(st1, st2, thread, name, retrieve_prot_family_op,
                                    reply_port, null_user_msg)

    AND mark_retrieve_pf_port(st1, st2, reply_port, ri)
    )

  END cc_create_context_from_port
```

Upon receiving a **provide_prot_family** message on a $retrieve\_pf\_port$ for request $ccc$, the CC

- stores the received protection family with $ccc$,
- initializes $ccc\_init\_cc\_args(ccc)$ to be a sequence of null names with the same length as the protection family, and
- disassociates the $retrieve\_pf\_port$ from $ccc$.

## THEORY *cc_provide_prot_family*

```
cc_provide_prot_family: THEORY
  BEGIN

  IMPORTING cc_ops_base


  % VARIABLES

  st1, st2: VAR (CC_STATE)

  thread: VAR THREAD

  svc_port : VAR NAME

  prot_family : VAR PROT_FAMILY

  ri: VAR RECEIVED_INFO

  kernel_req : VAR KERNEL_REQ

  ccc: VAR RECEIVED_INFO
```

```
unmark_retrieve_pf_port(st1, st2, svc_port) : bool =
   avail_port(st2) = add(svc_port, avail_port(st1))
   AND retrieve_pf_port(st2) = remove(svc_port, retrieve_pf_port(st1))
       AND pending_retrieve_pf(st2) =
               (LAMBDA (port : (retrieve_pf_port(st2))) :
                       pending_retrieve_pf(st1)(port))


% start processing the protection family by storing it and
% initializing ccc_init_cc_args.
start_process_prot_family(thread, ccc, prot_family, st1, st2): bool =
   active_ccc(st1)(ccc)
       AND active_ccc(st2) = active_ccc(st1)
         AND ccc_prot_family(st2)
           = ccc_prot_family(st1) WITH [ccc := prot_family]
           AND size(prot_family) > 0
             AND ccc_init_cc_args(st2)
               = ccc_init_cc_args(st1)
               WITH [ccc := null_name_seq(size(prot_family))]
               % ccc_handle is still null_name from initial state.
               AND ccc_handle(st2) = ccc_handle(st1)
                 AND existing_threads(kst(st1))(thread)
                   AND existing_threads(kst(st2)) = existing_threads(kst(st1))

cc_provide_prot_family(st1, st2): bool
= (EXISTS thread, ri, prot_family, svc_port, kernel_req, ccc:
     receive_request(thread, ri, provide_prot_family_op,
                             provide_prot_family_perm, st1, st2)
       AND provide_prot_family_msg(prot_family) = user_msg(ri)
       AND svc_port = service_port(ri)
       AND retrieve_pf_port(st1)(svc_port)
       AND ccc = pending_retrieve_pf(st1)(svc_port)

       AND key_init_inv(st1, st2)
       AND context_port_inv(st1,st2)
       AND pending_requests(kst(st2)) = pending_requests(kst(st1))

       AND start_process_prot_family(thread, ccc, prot_family, st1, st2)

       AND unmark_retrieve_pf_port(st1, st2, svc_port)
       )


END cc_provide_prot_family
```

If there is a protection in the protection family of *ccc* that does not yet have a key server handle, the CC can do the following:

- assign an available port as the *key_init_port* for that protection and
- send an **init_key_retrieval** request to the port for the key server associated with the key mechanism specified in the protection, passing the newly assigned *key_init_port* as the reply port.

## THEORY *cc_init_key_retrieval*

```
cc_init_key_retrieval : THEORY
  BEGIN

  IMPORTING cc_ops_base


  st1, st2 : VAR (CC_STATE)

  thread: VAR THREAD

  to, reply_port, key_svr, port: VAR NAME

  n : VAR posnat

  ccc : VAR RECEIVED_INFO

  prot : VAR PROT

  prot_family : VAR PROT_FAMILY

  kernel_req : VAR KERNEL_REQ

  op : VAR OP

  msg : VAR USER_MSG

  %% Some utility functions.

mark_key_init_port(st1, st2, reply_port, ccc, n) : bool =
    avail_port(st1)(reply_port)
        AND avail_port(st2) = remove(reply_port, avail_port(st1))
        AND key_init_port(st2) = add(reply_port, key_init_port(st1))
        AND key_init_port(st2)(reply_port)
        AND pending_key_init(st2) = pending_key_init(st1)
                                        WITH [(reply_port) := (ccc, n)]


need_key_init_port(st1, st2, ccc, n, prot) : bool =
  (EXISTS prot_family:
    active_ccc(st1)(ccc)
        AND prot_family = ccc_prot_family(st1)(ccc)
        AND (n > 0 AND n <= size(prot_family))
        AND prot = (elem(prot_family))(n)
        AND (n > 0 AND n <= size(ccc_init_cc_args(st1)(ccc)))
        AND (elem(ccc_init_cc_args(st1)(ccc)))(n) = null_name
        AND NOT (EXISTS port :
                key_init_port(st1)(port)
                AND pending_key_init(st1)(port) = (ccc, n)))



  %% send an init_key_retrieval request to a key server
  %% if we haven't already done so for this ccc.
  cc_init_key_retrieval(st1, st2) : bool =
   (EXISTS thread, key_svr, reply_port, ccc, n, prot :

     cc_step(st1, st2)

        AND need_key_init_port(st1, st2, ccc, n, prot)

        AND active_ccc(st2) = active_ccc(st1)
        AND ccc_prot_family(st2) = ccc_prot_family(st1)
        AND ccc_init_cc_args(st2) = ccc_init_cc_args(st1)
        AND existing_threads(kst(st2)) = existing_threads(kst(st1))
```

```
    AND ccc_handle(st2) = ccc_handle(st1)

    AND retrieve_pf_inv(st1, st2)
    AND context_port_inv(st1,st2)

    %% send init_key_retrival to key server
    AND key_svr = key_mech_server(st1)(key_mech(prot))
    AND send_msg(st1, st2, thread, key_svr, init_key_retrieval_op,
                 reply_port, null_user_msg)


    %% mark reply_port as being used for (ccc, n)
    AND mark_key_init_port(st1, st2, reply_port, ccc, n)
    )



 END cc_init_key_retrieval
```

Upon receiving a **provide_key_port** request on a *key_init_port* for protection $n$ of the protection family of *ccc* (where no such request has previously been processed for protection $n$), the CC

- stores the provided key server handle in position $n$ of *ccc_init_cc_args*(*ccc*),
- disassociates the *key_init_port* from *ccc*.

## THEORY *cc_provide_key_port*

```
cc_provide_key_port: THEORY
 BEGIN

 IMPORTING crypto_shared_state
 IMPORTING cc_ops_base


 % VARIABLES

 st1, st2: VAR (CC_STATE)

 thread: VAR THREAD

 svc_port : VAR NAME

 key_port,port : VAR NAME

 ri: VAR RECEIVED_INFO

 ccc: VAR RECEIVED_INFO

 cc_args : VAR NAME_SEQ

 n : VAR posnat

 unmark_key_init_port(st1, st2, svc_port) : bool =
```

```
  avail_port(st2) = add(svc_port, avail_port(st1))
       AND key_init_port(st2) = remove(svc_port, key_init_port(st1))
       AND pending_key_init(st2) =
                 (LAMBDA (port : (key_init_port(st2))) :
                           pending_key_init(st1)(port))

%% record a key_port from a provide_key_port message.

cc_provide_key_port(st1, st2): bool =
  (EXISTS thread, ri, svc_port, ccc, cc_args, key_port, n:

       receive_request(thread, ri, provide_key_port_op,
                           provide_key_port_perm, st1, st2)

       %%preconditions
       AND provide_key_port_msg(key_port) = user_msg(ri)
       AND svc_port = service_port(ri)
       AND key_init_port(st1)(svc_port)
       AND (ccc, n) = pending_key_init(st1)(svc_port)
       AND active_ccc(st1)(ccc)
       AND cc_args = ccc_init_cc_args(st1)(ccc)
       AND (n > 0 AND n <= size(cc_args))
       AND elem(cc_args)(n) = null_name
       AND existing_threads(kst(st1))(thread)

       %% invariants
       AND active_ccc(st2) = active_ccc(st1)
       AND avail_port(st2) = avail_port(st1)
       AND ccc_prot_family(st2) = ccc_prot_family(st1)
       AND existing_threads(kst(st2)) = existing_threads(kst(st1))

       AND ccc_handle(st2) = ccc_handle(st1)

       AND retrieve_pf_inv(st1, st2)
       AND context_port_inv(st1,st2)
       AND pending_requests(kst(st2)) = pending_requests(kst(st1))

       AND (n > 0 AND n <= size(cc_args))
       AND ccc_init_cc_args(st2) = ccc_init_cc_args(st1)
                 WITH [ ccc :=
                           (# size := size(cc_args),
                              elem := elem(cc_args) WITH
                                        [(n) := key_port] #) ]

       AND unmark_key_init_port(st1, st2, svc_port)
       )

END cc_provide_key_port
```

If a key server handle has been stored for all protections in the protection family of $ccc$, the CC can do the following:

- assign an available port as the $context\_port$ for $ccc$ and
- send an **init_crypto_context** request to the port associated with the encryption mechanism of the first protection in the protection family of $ccc$, passing the newly assigned $context\_port$ as the reply port. The request contains a sequence of port names $k_1, p_2, k_2, \ldots, p_n, k_n$ where $p_i$ is the port associated with the encryption mechanism of the $i$-th protection in the protection family of $ccc$, and $k_i$ is the key server handle in position $i$ of $ccc\_init\_cc\_args(ccc)$.

# THEORY *cc_init_crypto_context*

```
cc_init_crypto_context : THEORY
  BEGIN

  IMPORTING cc_ops_base


  st, st1, st2 : VAR (CC_STATE)

  thread: VAR THREAD

  first_prot_task, reply_port : VAR NAME

  n, i : VAR posnat

  ccc : VAR RECEIVED_INFO

  prot, first_prot : VAR PROT

  prot_family : VAR PROT_FAMILY

  kernel_req : VAR KERNEL_REQ

  key_port_seq, name_seq : VAR NAME_SEQ



  %% Some utility functions.

  prot_to_prot_task(st, prot) : NAME =
    encrypt_mech_server(st)(encrypt_mech(prot))


  mark_context_port(st1, st2, reply_port, ccc) : bool =
    avail_port(st1)(reply_port)
         AND avail_port(st2) = remove(reply_port, avail_port(st1))
         AND context_port(st2) = add(reply_port, context_port(st1))
         AND context_port(st2)(reply_port)
         AND pending_context_port(st2) = pending_context_port(st1)
                                   WITH [(reply_port) := ccc]

  %% Assemble list of alternating prot_task ports and key ports
  %%   omitting first prot_task and starting with a key port.
  merged_seq(st, prot_family, key_port_seq, name_seq) : bool =
    (EXISTS (f : [{i | i>0 and i <= 2 * size(prot_family) - 1} -> NAME]) :
                  (size(name_seq) > 0
                          AND size(name_seq) = 2 * size(prot_family) - 1)
                  AND elem(name_seq) = f
                  AND (FORALL n :
                          (n > 1 and n <= size(prot_family)) IMPLIES
                          f(2*n-2) = prot_to_prot_task(st,elem(prot_family)(n)))
                  AND (size(name_seq) > 0
                          AND size(name_seq) = 2 * size(key_port_seq) - 1)
                  AND (FORALL n :
                          (n > 0 and n <= size(key_port_seq)) IMPLIES
                          f(2*n-1) = elem(key_port_seq)(n)))


  assemble_crypto_context_info(st1, st2, ccc, first_prot_task,
                                   name_seq) : bool =
```

```
   (EXISTS prot_family, first_prot :
     active_ccc(st1)(ccc)
       AND prot_family = ccc_prot_family(st1)(ccc)
       AND (FORALL n :
               (n > 0 AND n <= size(ccc_init_cc_args(st1)(ccc)))
               IMPLIES (elem(ccc_init_cc_args(st1)(ccc)))(n) /= null_name)
       AND (1 > 0 and 1 <= size(prot_family))
       AND first_prot = (elem(prot_family))(1)
       AND first_prot_task = prot_to_prot_task(st1, first_prot)
       AND merged_seq(st1, prot_family, ccc_init_cc_args(st1)(ccc), name_seq))


 %% send an init_crypto_context request to the first prot task
 %% in the selected prot family for a ccc. All key ports
 %% must already be obtained.

 cc_init_crypto_context(st1, st2) : bool =
  (EXISTS thread, reply_port, ccc, first_prot_task, name_seq :

    cc_step(st1, st2)

       AND assemble_crypto_context_info(st1, st2, ccc, first_prot_task,
                                        name_seq)

       AND active_ccc(st2) = active_ccc(st1)
       AND ccc_prot_family(st2) = ccc_prot_family(st1)
       AND ccc_init_cc_args(st2) = ccc_init_cc_args(st1)
       AND existing_threads(kst(st2)) = existing_threads(kst(st1))

       AND ccc_handle(st2) = ccc_handle(st1)

       AND retrieve_pf_inv(st1, st2)
       AND key_init_inv(st1, st2)

       %% send init_crypto_context to prot task
       AND send_msg(st1, st2, thread, first_prot_task, init_crypto_context_op,
                                      reply_port,
                                      init_crypto_context_msg(name_seq))

       %% mark reply_port as being used for this context initialization
       AND mark_context_port(st1, st2, reply_port, ccc)
       )



 END cc_init_crypto_context
```

Upon receiving a **provide_crypto_handle** request on a *context_port* for request *ccc*, the CC

- stores the received handle with *ccc*,
- sends a **provide_crypto_context** message containing the handle to the reply port that was specified in the *ccc* request, and
- disassociates the *context_port* from *ccc*.

## THEORY *cc_provide_crypto_handle*

```
cc_provide_crypto_handle: THEORY
  BEGIN

  IMPORTING cc_ops_base


  % VARIABLES

  st1, st2: VAR (CC_STATE)

  thread: VAR THREAD

  port, name: VAR NAME

  ccc, ri: VAR RECEIVED_INFO

  svc_port : VAR NAME


  unmark_context_port(st1, st2, svc_port) : bool =
    avail_port(st2) = add(svc_port, avail_port(st1))
          AND context_port(st2) = remove(svc_port, context_port(st1))
          AND pending_context_port(st2) =
                   (LAMBDA (port : (context_port(st2))) :
                             pending_context_port(st1)(port))


  cc_provide_crypto_handle(st1, st2): bool =
    (EXISTS thread, ri, name, svc_port, ccc:

          receive_request(thread, ri, provide_crypto_handle_op,
                             provide_crypto_handle_perm, st1, st2)
          AND provide_crypto_handle_msg(name) = user_msg(ri)
          AND name /= null_name
          AND svc_port = service_port(ri)
          AND context_port(st1)(svc_port)
          AND ccc = pending_context_port(st1)(svc_port)
          AND active_ccc(st1)(ccc)
          AND ccc_handle(st1)(ccc) = null_name

          AND active_ccc(st2)(ccc)
          AND ccc_handle(st2) = ccc_handle(st1)
                   WITH [(ccc) := name]

          AND active_ccc(st2) = active_ccc(st1)
          AND ccc_prot_family(st2) = ccc_prot_family(st1)
          AND ccc_init_cc_args(st2) = ccc_init_cc_args(st1)

          AND retrieve_pf_inv(st1, st2)
          AND key_init_inv(st1, st2)

          AND existing_threads(kst(st2)) = existing_threads(kst(st1))

          AND send_msg(st1, st2, thread, reply_name(ccc),
                        provide_crypto_context_op, null_name,
                        provide_crypto_context_msg(name))

          AND unmark_context_port(st1, st2, svc_port)
          )


  END cc_provide_crypto_handle
```

A CC operation consists of any one of the operations defined above. The *guar* of the CC consists of those transitions with a CC thread serving as the agent such that the start and final states of the transition satisfy *cc_step* and *cc_op* or look the same with respect to *cc_view*.

## THEORY *cc_ops*

```
cc_ops: THEORY
  BEGIN

  IMPORTING cc_receive_request
  IMPORTING cc_create_context_from_port
  IMPORTING cc_init_crypto_context
  IMPORTING cc_init_key_retrieval
  IMPORTING cc_provide_prot_family
  IMPORTING cc_provide_key_port
  IMPORTING cc_provide_crypto_handle

  st1, st2 : VAR (CC_STATE)

  ag: VAR THREAD

  cc_op(st1, st2) : bool =
        cc_receive_request(st1, st2)
        OR cc_create_context_from_port(st1, st2)
        OR cc_provide_prot_family(st1, st2)
        OR cc_init_key_retrieval(st1, st2)
        OR cc_provide_key_port(st1, st2)
        OR cc_init_crypto_context(st1, st2)
        OR cc_provide_crypto_handle(st1, st2)

  cc_guar(st1,st2,ag) : bool =
   cc_threads(ag) AND
        (cc_view(st1, st2)
          OR (cc_step(st1, st2) AND cc_op(st1, st2)))

  END cc_ops
```

## 21.3 Environment Assumptions

The environment of the CC is assumed to alter no CC state information other than *kst* and to obey the constraints on changing *kst* that are given in *environment_base* on page 140. The *hidd* of the CC is defined similarly using *hidd_base*.

## THEORY *cc_rely*

```
cc_rely : THEORY

BEGIN

IMPORTING dtos_kernel_shared_rely
```

```
IMPORTING cc_state

st1, st2 : VAR (CC_STATE)

ag : VAR THREAD

cc_environment(st1,st2,ag) : bool =
    environment_base(ag,kst(st1),kst(st2)) and
    st1 with [kst := kst(st2)] = st2

cc_environment_refl: THEOREM
    cc_environment(st1,st1,ag)

cc_hidd(st1,st2,ag) : bool =
    NOT cc_threads(ag)
        AND hidd_base(ag, kst(st1), kst(st2))
        AND st2 = st1 WITH [ kst := kst(st2) ]

cc_hidd_prop: THEOREM
    cc_hidd(st1,st2,ag)
        => k_threads(ag) OR cc_view(st1,st2)


cc_rely(st1,st2,ag) : bool =
    not cc_threads(ag) AND
    cc_environment(st1,st2,ag)

END cc_rely
```

## 21.4 Component Specification

We use the set *initial_cc_states* to denote the valid initial states for the CC. A valid initial state has the following properties:

- There are no active requests and no recorded protection families, cryptographic handles and *init_CC_args*.
- The sets *retrieve_pf_port*, *key_init_port* and *context_port* are all empty.
- No kernel requests are pending for any CC thread and no messages are waiting to be processed.

A CC is a component having state type *CC_STATE*, satisfying initial constraint *initial_cc_states*, and executing only the transitions defined in Section 21.2.

### THEORY *cc_spec*

```
cc_spec : THEORY
BEGIN

IMPORTING dtos_kernel_shared_state
IMPORTING cc_ops
IMPORTING cc_rely
IMPORTING cc_state_witness
IMPORTING component_aux[(CC_STATE),THREAD]
```

```
st, st1, st2 : VAR (CC_STATE)
ag : VAR THREAD

initial_cc_states(st) : bool =
      active_ccc(st) = emptyset[RECEIVED_INFO]
      AND retrieve_pf_port(st) = emptyset
      AND ccc_prot_family(st) = (LAMBDA (x : RECEIVED_INFO) : null_prot_family)
      AND ccc_handle(st) = (LAMBDA (x : RECEIVED_INFO) : null_name)
      AND key_init_port(st) = emptyset
      AND ccc_init_cc_args(st) = (LAMBDA (x : RECEIVED_INFO) : null_name_seq(1))
      AND context_port(st) = emptyset
      AND pending_requests(kst(st)) = emptyset[KERNEL_REQ]
      AND (FORALL ag :
            existing_threads(kst(st))(ag) =>
                    ri_status(received_info(kst(st))(ag)) = ri_processed)

cc_state_witness_initial: THEOREM
   initial_cc_states(cc_state_witness)

base_cc_comp : base_comp_t =
       (# init := initial_cc_states,
          guar := cc_guar,
          rely := cc_rely,
          hidd := cc_hidd,
          cags := cc_threads,
          view := cc_view,
          wfar := emptyset[TRANSITION_CLASS[(CC_STATE), THREAD]],
          sfar := emptyset[TRANSITION_CLASS[(CC_STATE), THREAD]] #)

   cc_view_eq: THEOREM view_eq(base_cc_comp)

   cc_comp_init: THEOREM init_restriction(base_cc_comp)

   cc_comp_guar: THEOREM guar_restriction(base_cc_comp)

   cc_comp_rely_hidd: THEOREM rely_hidd_restriction(base_cc_comp)

   cc_comp_hidd: THEOREM hidd_restriction(base_cc_comp)

   cc_comp_rely: THEOREM rely_restriction(base_cc_comp)

   cc_comp_cags: THEOREM cags_restriction(base_cc_comp)

   cc_comp_guar_stuttering: THEOREM guar_stuttering_restriction(base_cc_comp)

   cc_comp_rely_stuttering: THEOREM rely_stuttering_restriction(base_cc_comp)

   cc_comp : (comp_t) = base_cc_comp

   cc_comp_hidd_prop: THEOREM
      hidd(cc_comp)(st1, st2, ag)
      => k_threads(ag) OR view(cc_comp)(st1, st2)

END cc_spec
```

Section 22
# Protection Tasks

This section describes the Synergy Protection Tasks Component (PT). The role of the each protection task is to encrypt and/or sign data according to some particular algorithm. The subsystem will typically contain numerous protection tasks.[37] A single cryptographic context will typically involve a sequence of protection tasks invoked in some fixed order.

## 22.1 State

The state type for this component will be defined in two parts, the internal state specific to each individual protection task, and the combined kernel interfaces of all the tasks. We begin with the task-specific portion, specified in type $PT\_THREAD\_STATE$.

Each individual protection task has a $service\_port$ on which it receives $init\_cc$ requests, and it implements a particular protection mechanism indicated by $encrypt\_mech$. Each task maintains a set $pt\_handles$ of port names that represent handles it has given out to provide access to its particular step in the use of a cryptographic context. The following information is associated with each handle:

- $pt\_reply\_to$ — where to send the handle when the downstream context is ready,
- $pt\_args$ — a list of port names passed in as arguments in the $init\_cc$ request representing the ports to be used for the downstream protection tasks,
- $pt\_key\_server\_reply\_port$ — the name of a port where the protection task is waiting to receive a key for use with its step in the cryptographic context associated with the handle.

The set $pt\_keyed$ represents the set of handles for which a key has been received. The expression $pt\_key(h)$ denotes the key associated with handle $h$. If $pt\_args(h)$ is nonempty, then, once a key has been received for $h$, an $init\_cc$ request will be sent to the service port of the next protection task. Assuming the CC is operating correctly, this port will be named by the second element of $pt\_args$. In this request, a reply port is provided and the reply port name is stored in $pt\_next\_pt\_reply\_port(h)$. Once a reply has been received from the next protection task, the port name in the reply is stored in $pt\_next\_pt(h)$ and $h$ is added to the set $pt\_pipeline\_initialized$.

Each protection task also maintains a set, $avail\_port$, of ports that are available for use as reply ports and handles.

All of the above state information, defined by the type $PT\_THREAD\_STATE$, is maintained by each element of $pt\_threads$. In a valid $PT\_THREAD\_STATE$,

[37] This component was specified before the current version of the framework containing $n$-way composition was written. The earlier versions of the framework contained a composition operator that applied to only a *pair* of components. With this operator it was important for practical considerations to have a small, fixed set of components. For this reason, all the protection tasks have been modeled as a single component. One disadvantage of this is that the model does not clearly capture the separation of the protection tasks into separate system components as completely as one might like. For example, the $hidd$ of the PT component restricts only *non*-PT agents, and the $view$ does not prevent one PT thread from seeing the state of all the others. With the new version of the framework, it is feasible to define and compose an array of protection task components. Time did not allow us to convert the definition of the PT component to an array of components in this way. Similar comments apply to the Key Servers Component.

- $pt\_handles$ is disjoint from $avail\_port$,
- $pt\_keyed$ and $pt\_pipeline\_initialized$ are subsets of $pt\_handles$ and
- the $service\_port$ is not in $avail\_port$.

A PT state consists of a mapping $thst$ from the $pt\_threads$ to their associated $PT\_THREAD\_STATE$ as well as the $KERNEL\_SHARED\_STATE$, $kst$. The latter contains $existing\_threads$, $pending\_requests$, $thread\_status$, and $received\_info$. The valid states for the protection tasks component are defined by $PT\_STATE$. In a valid state, the $existing\_threads$ must be a subset of $pt\_threads$.

All the data in $PT\_STATE$ is visible to the PT.

## THEORY *pt_state*

```
pt_state : THEORY
  BEGIN

  IMPORTING crypto_shared_state

  pt_threads: (nonempty?[THREAD])

  pt_threads_witness: (pt_threads)

  pt_threads_nonempty: AXIOM pt_threads /= emptyset


  % Each thread is a separate protection task with the following state information
  PT_THREAD_STATE_BASE : TYPE =
     [# service_port: NAME,              % where I receive my init_cc requests

        encrypt_mech : ENCRYPT_MECH,   % mechanism I provide to clients

        avail_port: setof[NAME],              % my supply of unused handles

        pt_handles : setof[NAME],             % handles I've given out.

        pt_reply_to : [(pt_handles) -> NAME],              % where to send the handle

        pt_args : [(pt_handles) -> NAME_SEQ],              % names passed in as arguments

        pt_key_server_reply_port : [(pt_handles) -> NAME], % where to receive the key.

        pt_keyed : setof[NAME],               % handles for which I have a key

        pt_key : [(pt_keyed) -> KEY],         % the key for each handle

        pt_next_pt_reply_port : [(pt_keyed) -> NAME], % where to receive handle
                                                               % from next pt.

        pt_pipeline_initialized : setof[NAME],            % the handles for which the pipeline
                                                     % has been initialized.
        pt_next_pt : [(pt_pipeline_initialized) -> NAME]               % handle for next prot task

        #]

  ptths : VAR PT_THREAD_STATE_BASE
```

```
  PT_THREAD_STATE(ptths) : bool =
    disjoint?(avail_port(ptths), pt_handles(ptths))
        AND NOT avail_port(ptths)(service_port(ptths))
        AND subset?(pt_keyed(ptths), pt_handles(ptths))
        AND subset?(pt_pipeline_initialized(ptths), pt_handles(ptths))

  PT_STATE_BASE : TYPE =
     [# thst : [(pt_threads) -> (PT_THREAD_STATE)],

        kst: KERNEL_SHARED_STATE
        #]

  ptstb : VAR PT_STATE_BASE

  PT_STATE(ptstb): bool =
    subset?(existing_threads(kst(ptstb)), pt_threads)

  st1, st2: VAR (PT_STATE)

  pt_view(st1,st2) : bool =
    st1 = st2

  END pt_state
```

# THEORY *pt_state_witness*

```
pt_state_witness: THEORY

BEGIN

  IMPORTING pt_state

  th,th1,th2 : VAR (pt_threads)

  pt_thread_state_witness: (PT_THREAD_STATE) =
    (# service_port := null_name,
       encrypt_mech := encrypt_mech_witness,
       avail_port := emptyset[NAME],
       pt_handles := emptyset[NAME],
       pt_reply_to := (LAMBDA (x: (emptyset[NAME])): null_name),
       pt_args := (LAMBDA (x: (emptyset[NAME])): null_name_seq(1)),
       pt_key_server_reply_port := (LAMBDA (x: (emptyset[NAME])): null_name),
       pt_keyed := emptyset[NAME],
       pt_key := (LAMBDA (x: (emptyset[NAME])): key_witness),
       pt_next_pt_reply_port := (LAMBDA (x: (emptyset[NAME])): null_name),
       pt_pipeline_initialized := emptyset[NAME],
       pt_next_pt := (LAMBDA (x: (emptyset[NAME])): null_name)
       #)

  pt_state_witness: (PT_STATE) =
    (# thst := (LAMBDA th : pt_thread_state_witness),
       kst := empty_kst
       #)

  pt_state_witness_prop : THEOREM
        (EXISTS (ptstb : (PT_STATE)) : TRUE)
```

**END** *pt_state_witness*

## 22.2 Operations

This section describes the subset of PT operations that are relevant to this example.

> *Editorial Note:*
> This section currently describes only successful processing of requests.

We first define several utility functions:

- $pt\_static$ — defines the following state invariants: $service\_port$ and $encrypt\_mech$ do not change for any thread, and the set of $existing\_threads$ does not change.
- $pt\_step$ — a PT thread obeys $pt\_static$, only makes allowed modifications to kernel state and does not modify the $thst$ of any other thread.
- $pt\_handles\_inv(st_1, st_2)$ — $pt\_handles$, $pt\_reply\_to$, $pt\_args$, and $pt\_key\_server\_reply\_port$ do not change for any thread.
- $pt\_keyed\_inv(st_1, st_2)$ — the key information and $pt\_next\_pt\_reply\_port$ do not change for any thread.
- $pt\_pipeline\_initialized\_inv(st_1, st_2)$ — $pt\_pipeline\_initialized$ and $pt\_next\_pt$ do not change for any thread.
- $pt\_initialize\_pipeline(st_1, st_2, thread, handle, next\_pt)$ — $handle$ is added to the set of initialized pipelines with next protection task $next\_pt$, and a $provide\_crypto\_handle$ message is sent to $pt\_reply\_to(handle)$.
- $pt\_receive\_request\_util(thread, ri, op\_id, perm, st_1, st_2)$ — $thread$ checks permission $perm$ and operation $op\_id$ on the received information in $ri$ and then uses $process\_request$ to mark $ri$ as processed.

## THEORY *pt_ops_base*

```
pt_ops_base: THEORY
  BEGIN

  IMPORTING pt_state

  IMPORTING dtos_kernel_shared_ops

  IMPORTING messaging

  st1, st2: VAR (PT_STATE)

  thread, th, th1, th2 : VAR (pt_threads)


  %%local state invariants
  pt_static(st1, st2): bool =
```

```
   (FORALL th: service_port(thst(st2)(th)) = service_port(thst(st1)(th))
        AND encrypt_mech(thst(st2)(th)) = encrypt_mech(thst(st1)(th)))
   AND existing_threads(kst(st2)) = existing_threads(kst(st1))

%a step must obey local invariants and only make allowed
% mods to kernel state or its own thst.
pt_step(st1, st2, thread): bool =
   pt_static(st1, st2)
        AND effects_on_kernel_state(kst(st1), kst(st2), pt_threads)
        AND (FORALL th:
                (NOT (th = thread) IMPLIES
                        thst(st1)(th) = thst(st2)(th)))


ri: VAR RECEIVED_INFO

op_id: VAR OP

perm: VAR PERMISSION

next_pt, handle: VAR NAME



% UTILITY FUNCTIONS

pt_handles_inv(st1, st2): bool =
   (FORALL th:
        pt_handles(thst(st2)(th)) = pt_handles(thst(st1)(th))
        AND pt_reply_to(thst(st2)(th)) = pt_reply_to(thst(st1)(th))
        AND pt_args(thst(st2)(th)) = pt_args(thst(st1)(th))
        AND pt_key_server_reply_port(thst(st2)(th)) = pt_key_server_reply_port(thst(st1)(th))
   )

pt_keyed_inv(st1, st2): bool =
   (FORALL th:
        pt_keyed(thst(st2)(th)) = pt_keyed(thst(st1)(th))
        AND pt_key(thst(st2)(th)) = pt_key(thst(st1)(th))
        AND pt_next_pt_reply_port(thst(st2)(th))
                = pt_next_pt_reply_port(thst(st1)(th)))


pt_pipeline_initialized_inv(st1, st2): bool =
   (FORALL th:
        pt_pipeline_initialized(thst(st2)(th)) = pt_pipeline_initialized(thst(st1)(th))
        AND pt_next_pt(thst(st2)(th))
                = pt_next_pt(thst(st1)(th)))


pt_initialize_pipeline(st1, st2, thread, handle, next_pt): bool =
        pt_pipeline_initialized(thst(st2)(thread)) =
                add(handle, pt_pipeline_initialized(thst(st1)(thread)))
        AND pt_next_pt(thst(st2)(thread)) =
                pt_next_pt(thst(st2)(thread)) WITH [handle := next_pt]

        AND pt_handles(thst(st1)(thread))(handle)
        AND send_msg(kst(st1), kst(st2), thread, pt_reply_to(thst(st1)(thread))(handle),
                        provide_crypto_handle_op,
                        null_name, provide_crypto_handle_msg(handle))


% processing a newly received request
pt_receive_request_util(thread, ri, op_id, perm, st1, st2): bool =
   receive_request(thread, ri, op_id, perm, kst(st1), kst(st2))
```

```
  END pt_ops_base
```

At any time when the PT has a thread that is not already waiting for a message operation to be performed, that thread can request to receive a message from a port. The thread initiates this processing by setting its pending request to be a message receive and changing its state to $thread\_waiting$.

## THEORY *pt_receive_request*

```
pt_receive_request: THEORY
  BEGIN

  IMPORTING pt_ops_base

  st1, st2: VAR (PT_STATE)

  thread: VAR (pt_threads)

  name: VAR NAME

  pt_receive_request_submit(st1, st2, thread): bool =
    EXISTS name:
      receive_msg(kst(st1), kst(st2), thread, name)

  pt_receive_request(st1, st2, thread): bool =
   thst(st2) = thst(st1)
        AND pt_receive_request_submit(st1, st2, thread)

  END pt_receive_request
```

When a protection task thread receives a valid $init\_crypto\_context\_op$ request on its $service\_port$, it

- allocates an available port to serve as a new handle $h$,
- sets $pt\_reply\_to(h)$ to the reply port provided in the request,
- stores the arguments of the request in $pt\_args(h)$,
- allocates another available port to serve as the reply port in a $retrieve\_key\_op$ request to a key server, and
- sends a $retrieve\_key\_op$ request to the first name in the argument list.

## THEORY *pt_init_crypto_context*

```
pt_init_crypto_context: THEORY
```

```
BEGIN

IMPORTING pt_ops_base


% VARIABLES

st1, st2: VAR (PT_STATE)

thread: VAR (pt_threads)

name_seq: VAR NAME_SEQ

ri: VAR RECEIVED_INFO

key_server, new_handle, reply_port : VAR NAME


initialize_handle(st1, st2, thread, new_handle, ri, name_seq, reply_port) : bool =
  avail_port(thst(st1)(thread))(new_handle)
      AND avail_port(thst(st2)(thread)) = remove(new_handle, avail_port(thst(st1)(thread)))
      AND pt_handles(thst(st2)(thread)) = add(new_handle, pt_handles(thst(st1)(thread)))
      AND pt_handles(thst(st2)(thread))(new_handle)
      AND pt_reply_to(thst(st2)(thread)) =
              pt_reply_to(thst(st1)(thread)) WITH [(new_handle) := reply_name(ri)]
      AND pt_args(thst(st2)(thread)) =
              pt_args(thst(st1)(thread)) WITH [(new_handle) := name_seq]

      AND avail_port(thst(st1)(thread))(reply_port)
      AND avail_port(thst(st2)(thread)) = remove(reply_port, avail_port(thst(st1)(thread)))
      AND pt_key_server_reply_port(thst(st2)(thread)) =
              pt_key_server_reply_port(thst(st1)(thread)) WITH [(new_handle) := reply_port]


pt_init_crypto_context(st1, st2, thread): bool =
  (EXISTS ri, reply_port, key_server, new_handle, name_seq:


      pt_receive_request_util(thread, ri, init_crypto_context_op,
                              init_crypto_context_perm, st1, st2)
      AND init_crypto_context_msg(name_seq) = user_msg(ri)
      AND service_port(thst(st1)(thread)) = service_port(ri)
      AND 1 <= size(name_seq)
      AND key_server = elem(name_seq)(1)

      AND initialize_handle(st1, st2, thread, new_handle, ri, name_seq, reply_port)

      AND existing_threads(kst(st2)) = existing_threads(kst(st1))
      AND pt_keyed_inv(st1, st2)
      AND pt_pipeline_initialized_inv(st1, st2)

      AND send_msg(kst(st1), kst(st2), thread, key_server, retrieve_key_op,
                                            reply_port, null_user_msg)
      )


END pt_init_crypto_context
```

When a protection task thread receives a valid $protect\_op$ request, containing $text$ and $dest$, on one of its handle ports $h$, it

- encrypts $text$ according to its $encrypt\_mech$ using $pt\_key(h)$ yielding $protected\_text$, and

- if $pt\_next\_pt(h)$ is not $null\_name$
  - it sends a $protect\_op$ message containing $protected\_text$ and $dest$ to $pt\_next\_pt(h)$,
  - otherwise, it sends a $provide\_protected\_data\_op$ message containing $protected\_text$ to $dest$.

## THEORY *pt_protect*

```
pt_protect: THEORY
  BEGIN

  IMPORTING pt_ops_base


  % VARIABLES

  st1, st2: VAR (PT_STATE)

  thread: VAR (pt_threads)

  ri: VAR RECEIVED_INFO

  handle, dest, next_pt : VAR NAME

  protected_text, text : VAR TEXT


  pt_more_protecters(st1, st2, thread, handle, protected_text, dest): bool =
        pt_pipeline_initialized(thst(st1)(thread))(handle)
        AND pt_next_pt(thst(st1)(thread))(handle) /= null_name

        AND send_msg(kst(st1), kst(st2), thread, pt_next_pt(thst(st1)(thread))(handle),
                           protect_op, null_name,
                           protect_msg(protected_text, dest))


  pt_last_protecter(st1, st2, thread, handle, protected_text, dest): bool =
        pt_pipeline_initialized(thst(st1)(thread))(handle)
        AND pt_next_pt(thst(st1)(thread))(handle) = null_name

        AND send_msg(kst(st1), kst(st2), thread, dest,
                           provide_protected_data_op, null_name,
                           provide_protected_data_msg(protected_text))



  pt_protect(st1, st2, thread): bool =
    (EXISTS ri, handle, protected_text, text, dest:

        pt_receive_request_util(thread, ri, protect_op,
                           protect_perm, st1, st2)
        AND protect_msg(text, dest) = user_msg(ri)
        AND handle = service_port(ri)
        AND pt_handles(thst(st1)(thread))(handle)
        AND pt_keyed(thst(st1)(thread))(handle)
        AND pt_pipeline_initialized(thst(st1)(thread))(handle)
```

```
        AND existing_threads(kst(st2)) = existing_threads(kst(st1))
        AND pt_handles_inv(st1, st2)
        AND pt_keyed_inv(st1, st2)
        AND pt_pipeline_initialized_inv(st1, st2)

        AND protected_text =
              protect_text(encrypt_mech(thst(st1)(thread)),
                           pt_key(thst(st1)(thread))(handle),
                           text)

        AND (pt_more_protecters(st1, st2, thread, handle, protected_text, dest)
              OR pt_last_protecter(st1, st2, thread, handle, protected_text, dest))

    )

  END pt_protect
```

When a protection task thread receives a valid $provide\_crypto\_handle\_op$ request on a port $pt\_next\_pt\_reply\_port(h)$ for one of its handles $h$, it initializes the pipeline associated with $h$ (see the utility function $pt\_initialize\_pipeline$).

## THEORY *pt_provide_crypto_handle*

```
pt_provide_crypto_handle: THEORY
  BEGIN

  IMPORTING pt_ops_base


  % VARIABLES

  st1, st2: VAR (PT_STATE)

  thread: VAR (pt_threads)

  ri: VAR RECEIVED_INFO

  handle, next_pt : VAR NAME


  pt_provide_crypto_handle(st1, st2, thread): bool =
    (EXISTS ri, handle, next_pt:

        pt_receive_request_util(thread, ri, provide_crypto_handle_op,
                                provide_crypto_handle_perm, st1, st2)
        AND provide_crypto_handle_msg(next_pt) = user_msg(ri)
        AND pt_keyed(thst(st1)(thread))(handle)
        AND pt_next_pt_reply_port(thst(st1)(thread))(handle) = service_port(ri)

        AND existing_threads(kst(st2)) = existing_threads(kst(st1))
        AND pt_handles_inv(st1, st2)
        AND pt_keyed_inv(st1, st2)

        AND pt_initialize_pipeline(st1, st2, thread, handle, next_pt)
    )
```

**END** *pt_provide_crypto_handle*

---

When a protection task thread receives a valid $provide\_key\_op$ request on a port $pt\_key\_server\_reply\_port(h)$ for one of its handles $h$, then

- if $pt\_args(h)$ has length of at least 2, it
  - stores the key,
  - allocates a reply port which is stored in $pt\_next\_pt\_reply\_port(h)$, and
  - sends an $init\_crypto\_context\_op$ request (supplying an argument list consisting of its own $pt\_args(h)$ with the first two names stripped off) to the second name in $pt\_args(h)$;
- otherwise, it
  - stores the key,
  - sets $pt\_next\_pt\_reply\_port(h)$ to be $null\_name$, and
  - initializes the pipeline associated with $h$ (with itself as the final task in the pipeline).

## THEORY *pt_provide_key*

---

```
pt_provide_key: THEORY
  BEGIN

  IMPORTING pt_ops_base


  % VARIABLES

  st1, st2: VAR (PT_STATE)

  thread: VAR (pt_threads)

  args: VAR NAME_SEQ

  ri: VAR RECEIVED_INFO

  key: VAR KEY

  handle, reply_port, next_pt_reply_port : VAR NAME


  pt_store_key(st1, st2, thread, handle, key, next_pt_reply_port) : bool =
        pt_keyed(thst(st2)(thread)) = add(handle, pt_keyed(thst(st1)(thread)))
        AND pt_key(thst(st2)(thread)) = pt_key(thst(st1)(thread)) WITH [handle := key]
        AND pt_next_pt_reply_port(thst(st2)(thread)) =
                  pt_next_pt_reply_port(thst(st1)(thread))
                        WITH [handle := next_pt_reply_port]


  pt_more_pts(st1, st2, thread, handle, key): bool =
    (EXISTS reply_port, args:
        pt_handles(thst(st1)(thread))(handle)
```

```
        AND args = pt_args(thst(st1)(thread))(handle)
        AND 2 <= size(args)

        AND pt_store_key(st1, st2, thread, handle, key, reply_port)

        AND pt_pipeline_initialized_inv(st1, st2)

        AND avail_port(thst(st1)(thread))(reply_port)
        AND avail_port(thst(st2)(thread)) = remove(reply_port, avail_port(thst(st1)(thread)))

        AND send_msg(kst(st1), kst(st2), thread, elem(args)(2), init_crypto_context_op,
                         reply_port, init_crypto_context_msg(pop(pop(args))))
        )

  pt_last_pt(st1, st2, thread, handle, key): bool =
    (EXISTS args:
        pt_handles(thst(st1)(thread))(handle)
        AND args = pt_args(thst(st1)(thread))(handle)
        AND size(args) = 1
        AND pt_store_key(st1, st2, thread, handle, key, null_name)

        AND pt_initialize_pipeline(st1, st2, thread, handle, null_name)
    )

  pt_provide_key(st1, st2, thread): bool =
    (EXISTS ri, handle, key:

        pt_receive_request_util(thread, ri, provide_key_op,
                         provide_key_perm, st1, st2)
        AND provide_key_msg(key) = user_msg(ri)
        AND pt_handles(thst(st1)(thread))(handle)
        AND pt_key_server_reply_port(thst(st1)(thread))(handle) = service_port(ri)

        AND existing_threads(kst(st2)) = existing_threads(kst(st1))
        AND pt_handles_inv(st1, st2)

        AND (pt_more_pts(st1, st2, thread, handle, key)
              OR pt_last_pt(st1, st2, thread, handle, key))
        )

  END pt_provide_key
```

---

A PT operation consists of any one of the operations defined above. The *guar* of the PT consists of those transitions with a PT thread serving as the agent such that the start and final states of the transition satisfy *pt_step* and *pt_op* or look the same with respect to *pt_view*.

## THEORY *pt_ops*

---

```
pt_ops: THEORY
  BEGIN

  IMPORTING pt_receive_request
  IMPORTING pt_init_crypto_context
  IMPORTING pt_provide_key
  IMPORTING pt_provide_crypto_handle
  IMPORTING pt_protect
```

```
  st1, st2 : VAR (PT_STATE)
  thread : VAR THREAD
  th: VAR (pt_threads)

  pt_op(st1, st2, th) : bool =
    pt_receive_request(st1, st2, th)
        OR pt_init_crypto_context(st1, st2, th)
        OR pt_provide_key(st1, st2, th)
        OR pt_provide_crypto_handle(st1, st2, th)
        OR pt_protect(st1, st2, th)

  pt_guar(st1,st2,thread) : bool =
    pt_threads(thread) AND
    (pt_view(st1,st2)
        OR (pt_step(st1, st2, thread) AND
              pt_op(st1, st2, thread)))

  END pt_ops
```

## 22.3 Environment Assumptions

The environment of the PT is assumed to alter no PT state information other than $kst$ and to obey the constraints on changing $kst$ that are given in $environment\_base$ on page 140. The $hidd$ of the PT is defined similarly using $hidd\_base$.

## THEORY *pt_rely*

```
pt_rely : THEORY

BEGIN

IMPORTING dtos_kernel_shared_rely

IMPORTING pt_state

st1, st2 : VAR (PT_STATE)

ag : VAR THREAD

pt_environment(st1,st2,ag) : bool =
    environment_base(ag,kst(st1),kst(st2)) and
    st1 with [kst := kst(st2)] = st2

pt_environment_refl: THEOREM
    pt_environment(st1,st1,ag)

pt_hidd(st1,st2,ag) : bool =
    NOT pt_threads(ag)
        AND hidd_base(ag, kst(st1), kst(st2))
        AND st2 = st1 with [ kst := kst(st2) ]

pt_hidd_prop: THEOREM
    pt_hidd(st1,st2,ag)
        => k_threads(ag) OR pt_view(st1,st2)


pt_rely(st1,st2,ag) : bool =
    not pt_threads(ag) AND
```

```
    pt_environment(st1,st2,ag)

END pt_rely
```

## 22.4 Component Specification

We use the set *initial_pt_states* to denote the valid initial states for the PT. A valid initial state has the following properties:

- No PT thread has an active handle.
- No kernel requests are pending for any PT thread and no messages are waiting to be processed.

A PT is a component having state type *PT_STATE*, satisfying initial constraint *initial_pt_states*, and executing only the transitions defined in Section 22.2.

### THEORY *pt_spec*

```
pt_spec : THEORY
BEGIN

IMPORTING dtos_kernel_shared_state
IMPORTING pt_ops
IMPORTING pt_rely
IMPORTING pt_state_witness
IMPORTING component_aux[(PT_STATE), THREAD]


st, st1, st2 : VAR (PT_STATE)
ag : VAR THREAD
thread : VAR (pt_threads)

initial_pt_states(st) : bool =
  (FORALL thread:
    pt_handles(thst(st)(thread)) = emptyset[NAME]
    AND pending_requests(kst(st)) = emptyset[KERNEL_REQ]
    AND (FORALL ag :
        existing_threads(kst(st))(ag) =>
              ri_status(received_info(kst(st))(ag)) = ri_processed))

pt_state_witness_initial: THEOREM
  initial_pt_states(pt_state_witness)

base_pt_comp : base_comp_t =
     (# init := initial_pt_states,
        guar := pt_guar,
        rely := pt_rely,
        hidd := pt_hidd,
        cags := pt_threads,
        view := pt_view,
        wfar := emptyset[TRANSITION_CLASS[(PT_STATE), THREAD]],
        sfar := emptyset[TRANSITION_CLASS[(PT_STATE), THREAD]] #)

  pt_view_eq: THEOREM view_eq(base_pt_comp)
```

```
  pt_comp_init: THEOREM init_restriction(base_pt_comp)

  pt_comp_guar: THEOREM guar_restriction(base_pt_comp)

  pt_comp_rely_hidd: THEOREM rely_hidd_restriction(base_pt_comp)

  pt_comp_hidd: THEOREM hidd_restriction(base_pt_comp)

  pt_comp_rely: THEOREM rely_restriction(base_pt_comp)

  pt_comp_cags: THEOREM cags_restriction(base_pt_comp)

  pt_comp_guar_stuttering: THEOREM guar_stuttering_restriction(base_pt_comp)

  pt_comp_rely_stuttering: THEOREM rely_stuttering_restriction(base_pt_comp)

  pt_comp : (comp_t) = base_pt_comp

  pt_comp_hidd_prop: THEOREM
    hidd(pt_comp)(st1, st2, ag)
    => k_threads(ag) OR view(pt_comp)(st1, st2)

END pt_spec
```

# Section 23 Key Servers

This section describes the Synergy Key Servers (KS). The role of each key server is to provide keys to the protection tasks according to a given key generation algorithm.

## 23.1 State

As for the PT component the state type for this component will be defined in two parts, the internal state specific to each individual key server, and the combined kernel interfaces of all the tasks. We begin with the task-specific portion, specified in type $KS\_THREAD\_STATE$.

Each individual key server has a $service\_port$ on which it receives $init\_key\_retrieval\_op$ requests, and it implements a particular key generation algorithm indicated by $server\_mech$. Each task maintains a set $key\_handles$ of port names that represent handles it has given out to provide access to its services to a given protection task within a given cryptographic context. A key is associated with each handle by the function $handle\_to\_key$.

Each key server also maintains a set, $avail\_port$, of ports that are available for use as handles.

All of the above state information, defined by the type $KS\_THREAD\_STATE$, is maintained by each element of $ks\_threads$. In a valid $KS\_THREAD\_STATE$,

- the set $key\_handles$ is disjoint from $avail\_port$ and
- the $service\_port$ is not in $avail\_port$.

A KS state consists of a mapping $thst$ from the $ks\_threads$ to their associated $KS\_THREAD\_STATE$ as well as the $KERNEL\_SHARED\_STATE$, $kst$. The latter contains $existing\_threads$, $pending\_requests$, $thread\_status$, and $received\_info$. The valid states for the protection tasks component are defined by $KS\_STATE$. In a valid state, the $existing\_threads$ must be a subset of $ks\_threads$.

All the data in $KS\_STATE$ is visible to the KS.

## THEORY *ks_state*

```
ks_state : THEORY
  BEGIN

  IMPORTING crypto_shared_state

  ks_threads: setof[THREAD]

  ks_threads_nonempty: AXIOM ks_threads /= emptyset
                                                              10
  ks_threads_witness: (ks_threads)
```

```
% Each thread is a separate key server with the following state information
KS_THREAD_STATE_BASE : TYPE =
  [# service_port: NAME,          % where I receive my requests

     server_mech: KEY_MECH,  % mechanism I provide to clients

     avail_port: setof[NAME],             % my supply of unused handles

     key_handles : setof[NAME],           % handles I've given out.

     handle_to_key : [(key_handles) -> KEY]        % the key associated with each handle

     #]

ksths : VAR KS_THREAD_STATE_BASE

KS_THREAD_STATE(ksths) : bool =
  disjoint?(avail_port(ksths), key_handles(ksths))
     AND NOT avail_port(ksths)(service_port(ksths))


KS_STATE_BASE : TYPE =
  [# thst : [(ks_threads) -> (KS_THREAD_STATE)],

     kst: KERNEL_SHARED_STATE
     #]

ksstb : VAR KS_STATE_BASE

KS_STATE(ksstb): bool =
  subset?(existing_threads(kst(ksstb)), ks_threads)

st1, st2: VAR (KS_STATE)

ks_view(st1,st2) : bool =
  st1 = st2

END ks_state
```

## THEORY *ks_state_witness*

```
ks_state_witness: THEORY

BEGIN

  IMPORTING ks_state

  th,th1,th2 : VAR (ks_threads)

  ks_thread_state_witness: (KS_THREAD_STATE) =
    (# service_port := null_name,
       server_mech :=  key_mech_witness,
       avail_port := emptyset[NAME],
       key_handles := emptyset[NAME],
       handle_to_key := (LAMBDA (x: (emptyset[NAME])): key_witness)
       #)

  ks_state_witness: (KS_STATE) =
    (# thst := (LAMBDA th : ks_thread_state_witness),
```

```
    kst := empty_kst
    #)

  ks_state_witness_prop : THEOREM
        (EXISTS (ksstb : (KS_STATE)) : TRUE)

END ks_state_witness
```

## 23.2 Operations

This section describes the subset of KS operations that are relevant to this example.

*Editorial Note:*
This section currently describes only successful processing of requests.

We first define several utility functions:

- $ks\_static$ — defines the following state invariants: $service\_port$ and $server\_mech$ do not change for any thread, and the set of $existing\_threads$ does not change.
- $ks\_step$ — a KS thread obeys $ks\_static$, only makes allowed modifications to kernel state and does not modify the $thst$ of any other thread.
- $ks\_handles\_inv(st_1, st_2)$ — $key\_handles$ and $handle\_to\_key$ do not change for any thread.
- $ks\_receive\_request\_util(thread, ri, op\_id, perm, st_1, st_2)$ — $thread$ checks permission $perm$ and operation $op\_id$ on the received information in $ri$ and then uses $process\_request$ to mark $ri$ as processed.

## THEORY *ks_ops_base*

```
ks_ops_base: THEORY
  BEGIN

  IMPORTING ks_state

  IMPORTING dtos_kernel_shared_ops

  IMPORTING messaging

  st1, st2: VAR (KS_STATE)

  th, th1, th2 : VAR (ks_threads)

  thread: VAR THREAD

  %%local state invariants
  ks_static(st1, st2): bool =
    (FORALL th: service_port(thst(st2)(th)) = service_port(thst(st1)(th))
         AND server_mech(thst(st2)(th)) = server_mech(thst(st1)(th)))
    AND existing_threads(kst(st2)) = existing_threads(kst(st1))

  %a step must obey local invariants and only make allowed
```

```
% mods to kernel state or its own thst.
ks_step(st1, st2, thread): bool =
  ks_static(st1, st2)
      AND effects_on_kernel_state(kst(st1), kst(st2), ks_threads)
      AND (FORALL th:
              (NOT (th = thread) IMPLIES
                      thst(st1)(th) = thst(st2)(th)))


key_handle_inv(st1, st2): bool =
  (FORALL th:
      key_handles(thst(st2)(th)) = key_handles(thst(st1)(th))
      AND handle_to_key(thst(st2)(th)) = handle_to_key(thst(st1)(th)))

ri: VAR RECEIVED_INFO

op_id: VAR OP

perm: VAR PERMISSION


% UTILITY FUNCTIONS


% processing a newly received request
ks_receive_request_util(thread, ri, op_id, perm, st1, st2): bool =
  receive_request(thread, ri, op_id, perm, kst(st1), kst(st2))

END ks_ops_base
```

At any time when the KS has a thread that is not already waiting for a message operation to be performed, that thread can request to receive a message from a port. The thread initiates this processing by setting its pending request to be a message receive and changing its state to *thread_waiting*.

## THEORY *ks_receive_request*

```
ks_receive_request: THEORY
  BEGIN

  IMPORTING ks_ops_base

  st1, st2: VAR (KS_STATE)

  thread: VAR (ks_threads)

  name: VAR NAME

  ks_receive_request_submit(st1, st2, thread): bool =
    EXISTS name:
      receive_msg(kst(st1), kst(st2), thread, name)

  ks_receive_request(st1, st2, thread): bool =
   thst(st2) = thst(st1)
      AND ks_receive_request_submit(st1, st2, thread)
```

```
END ks_receive_request
```

When a key server thread receives a valid *init_key_retrieval_op* request on its *service_port*, it[38]

- allocates an available port to serve as a new handle $h$,
- generates a key which it associates with $h$, and
- sends a *provide_key_port_op* request containing $h$ to the reply port in the *init_key_retrieval_op* request.

## THEORY *ks_init_key_retrieval*

```
ks_init_key_retrieval: THEORY
  BEGIN

%% The real crypto subsystem allows key servers to immediately retrieve a key to
%% be associated with the handle, wait until a key is actually requested
%% or fork a thread to retrieve a key to be associated with the handle.
%% For simplicity, we assume that the first option is always followed.

  IMPORTING ks_ops_base

  % VARIABLES

  st1, st2: VAR (KS_STATE)

  thread: VAR (ks_threads)

  ri: VAR RECEIVED_INFO

  handle, reply_port : VAR NAME

  key: VAR KEY

  seed : VAR SEED

  new_handle(st1, st2, handle, thread) : bool =
    avail_port(thst(st2)(thread)) = remove(handle, avail_port(thst(st1)(thread)))
        AND key_handles(thst(st2)(thread)) = add(handle, key_handles(thst(st1)(thread)))

  assign_key(st1, st2, handle, key, thread) : bool =
    key_handles(thst(st2)(thread))(handle)
    AND handle_to_key(thst(st2)(thread)) = handle_to_key(thst(st1)(thread))
                 WITH [(handle) := key]

  ks_init_key_retrieval(st1, st2, thread): bool =
    (EXISTS ri, reply_port, handle, key, seed:
```

[38] The real Crypto Subsystem allows key servers to immediately retrieve a key to be associated with the handle, wait until a key is actually requested or fork a thread to retrieve a key to be associated with the handle. For simplicity, we have modeled only the first option.

```
    ks_receive_request_util(thread, ri, init_key_retrieval_op,
                            init_key_retrieval_perm, st1, st2)
      AND null_user_msg = user_msg(ri)
      AND service_port(thst(st1)(thread)) = service_port(ri)
      AND reply_port = reply_name(ri)
      AND avail_port(thst(st1)(thread))(handle)
      AND key = generate_key(server_mech(thst(st1)(thread)), seed)

      AND new_handle(st1, st2, handle, thread)
      AND assign_key(st1, st2, handle, key, thread)

      AND send_msg(kst(st1), kst(st2), thread, reply_port, provide_key_port_op,
                   null_name,
                   provide_key_port_msg(handle))
      )

  END ks_init_key_retrieval
```

When a key server thread receives a valid $retrieve\_key\_op$ request on one of its handles $h$, it responds by sending a $provide\_key\_op$ message to the reply port containing the key associated with $h$.

# THEORY *ks_retrieve_key*

```
ks_retrieve_key: THEORY
  BEGIN

%% The real crypto subsystem allows key servers to immediately retrieve a key to
%% be associated with the handle, wait until a key is actually requested
%% or fork a thread to retrieve a key to be associated with the handle.
%% For simplicity, we assume that the first option is always followed.

  IMPORTING ks_ops_base


  % VARIABLES

  st1, st2: VAR (KS_STATE)

  thread: VAR (ks_threads)

  ri: VAR RECEIVED_INFO

  handle, reply_port : VAR NAME

  key: VAR KEY


  ks_retrieve_key(st1, st2, thread): bool =
    (EXISTS ri, reply_port, handle, key:

      ks_receive_request_util(thread, ri, retrieve_key_op,
                              retrieve_key_perm, st1, st2)
        AND null_user_msg = user_msg(ri)
        AND key_handles(thst(st1)(thread))(service_port(ri))
```

```
    AND reply_port = reply_name(ri)

    AND key = handle_to_key(thst(st1)(thread))(service_port(ri))
    AND key_handle_inv(st1,st2)
    AND avail_port(thst(st2)(thread)) = avail_port(thst(st1)(thread))

    AND send_msg(kst(st1), kst(st2), thread, reply_port, provide_key_op,
                 null_name,
                 provide_key_msg(key))
    )

  END ks_retrieve_key
```

A KS operation consists of any one of the operations defined above. The *guar* of the KS consists of those transitions with a KS thread serving as the agent such that the start and final states of the transition satisfy *ks_step* and *ks_op* or look the same with respect to *ks_view*.

## THEORY *ks_ops*

```
ks_ops: THEORY
  BEGIN

  IMPORTING ks_receive_request
  IMPORTING ks_init_key_retrieval
  IMPORTING ks_retrieve_key

  st1, st2 : VAR (KS_STATE)
  th : VAR (ks_threads)
  thread: VAR THREAD

  ks_op(st1, st2, th) : bool =
    ks_receive_request(st1, st2, th)
      OR ks_init_key_retrieval(st1, st2, th)
      OR ks_retrieve_key(st1, st2, th)

  ks_guar(st1,st2,thread) : bool =
    ks_threads(thread) AND
    (ks_view(st1,st2)
      OR (ks_step(st1, st2, thread) AND
          ks_op(st1, st2, thread)))

  END ks_ops
```

## 23.3 Environment Assumptions

The environment of the KS is assumed to alter no KS state information other than *kst* and to obey the constraints on changing *kst* that are given in *environment_base* on page 140. The *hidd* of the KS is defined similarly using *hidd_base*.

## THEORY *ks_rely*

```
ks_rely : THEORY

BEGIN

IMPORTING dtos_kernel_shared_rely

IMPORTING ks_state

st1, st2 : VAR (KS_STATE)

ag : VAR THREAD

ks_environment(st1,st2,ag) : bool =
    environment_base(ag,kst(st1),kst(st2)) and
    st1 with [kst := kst(st2)] = st2

ks_environment_refl: THEOREM
    ks_environment(st1,st1,ag)

ks_hidd(st1,st2,ag) : bool =
    NOT ks_threads(ag)
        AND hidd_base(ag, kst(st1), kst(st2))
        AND st2 = st1 with [ kst := kst(st2) ]

ks_hidd_prop: THEOREM
    ks_hidd(st1,st2,ag)
        => k_threads(ag) OR ks_view(st1,st2)


ks_rely(st1,st2,ag) : bool =
    not ks_threads(ag) AND
    ks_environment(st1,st2,ag)

END ks_rely
```

## 23.4 Component Specification

We use the set $initial\_ks\_states$ to denote the valid initial states for the KS. A valid initial state has the following properties:

- There are no active key handles.
- No kernel requests are pending for any KS thread and no messages are waiting to be processed.

A KS is a component having state type $KS\_STATE$, satisfying initial constraint $initial\_ks\_states$, and executing only the transitions defined in Section 23.2.

## THEORY *ks_spec*

```
ks_spec : THEORY
BEGIN

IMPORTING dtos_kernel_shared_state
IMPORTING ks_ops
IMPORTING ks_rely
```

```
IMPORTING ks_state_witness
IMPORTING component_aux[(KS_STATE),THREAD]

st, st1, st2 : VAR (KS_STATE)
ag : VAR THREAD
thread : VAR (ks_threads)

initial_ks_states(st) : bool =
  (FORALL thread:
    key_handles(thst(st)(thread)) = emptyset[NAME]
    AND handle_to_key(thst(st)(thread)) = (LAMBDA (x : (emptyset[NAME])) : key_witness)
    AND pending_requests(kst(st)) = emptyset[KERNEL_REQ]

    AND (FORALL ag :
        existing_threads(kst(st))(ag) =>
                  ri_status(received_info(kst(st))(ag)) = ri_processed))

ks_state_witness_initial: THEOREM
  initial_ks_states(ks_state_witness)

base_ks_comp : base_comp_t =
     (# init := initial_ks_states,
        guar := ks_guar,
        rely := ks_rely,
        hidd := ks_hidd,
        cags := ks_threads,
        view := ks_view,
        wfar := emptyset[TRANSITION_CLASS[(KS_STATE), THREAD]],
        sfar := emptyset[TRANSITION_CLASS[(KS_STATE), THREAD]] #)

  ks_view_eq: THEOREM view_eq(base_ks_comp)

  ks_comp_init: THEOREM init_restriction(base_ks_comp)

  ks_comp_guar: THEOREM guar_restriction(base_ks_comp)

  ks_comp_rely_hidd: THEOREM rely_hidd_restriction(base_ks_comp)

  ks_comp_hidd: THEOREM hidd_restriction(base_ks_comp)

  ks_comp_rely: THEOREM rely_restriction(base_ks_comp)

  ks_comp_cags: THEOREM cags_restriction(base_ks_comp)

  ks_comp_guar_stuttering: THEOREM guar_stuttering_restriction(base_ks_comp)

  ks_comp_rely_stuttering: THEOREM rely_stuttering_restriction(base_ks_comp)

  ks_comp : (comp_t) = base_ks_comp

  ks_comp_hidd_prop: THEOREM
    hidd(ks_comp)(st1, st2, ag)
    => k_threads(ag) OR view(ks_comp)(st1, st2)

END ks_spec
```

Section 24
# Security Service Usage Policy Server

This section describes the Synergy Security Service Usage Policy Server (SSUPS) component. The role of the SSUPS is to function as a security policy server for the use of cryptography. It associates with each cryptographic situation a list of possible protection and keying mechanisms that are acceptable for encrypting the data. Presumably, the network server would only send data out onto the network if it has been protected consistently with the decisions of the SSUPS.

## 24.1 State

The SSUPS for each node maintains a set $service\_port$ of ports on which it accepts $select\_prot\_family\_op$ requests and a set $handles$ of port names to serve as identifiers for protection families that have already been selected. The expression $handle\_pf(h)$ denotes the protection family associated with handle $h$. The policy in the SSUPS is represented by the function $sit\_pfs$ which maps each $SITUATION$ to a set of protection families. The SSUPS also maintains a set $avail\_port$ of port names available for use as handles.

The SSUPS state consists of the data structures described above as well as its $KERNEL\_SHARED\_STATE$, $kst$, containing $existing\_threads$, $pending\_requests$, $thread\_status$, and $received\_info$. The valid states are defined by $SSUPS\_STATE$. In a valid state,

- the sets $handles$ and $service\_port$ are disjoint from $avail\_port$, and
- $existing\_threads$ must be a subset of $ssups\_threads$.

All the data in $SSUPS\_STATE$ is visible to the SSUPS.

### THEORY *ssups_state*

```
ssups_state : THEORY
  BEGIN

  IMPORTING crypto_shared_state

  ssups_threads: (nonempty?[THREAD])

  ssups_threads_witness: (ssups_threads)

  ssups_threads_nonempty: THEOREM ssups_threads /= emptyset

  SSUPS_STATE_BASE: TYPE =
     [#
       avail_port: setof[NAME],               % my supply of ports

       service_port : setof[NAME],

       sit_pfs : [SITUATION -> setof[PROT_FAMILY]],
```

```
        handles : setof[NAME],

        handle_pf : [(handles) -> PROT_FAMILY],

        kst: KERNEL_SHARED_STATE
        #]

  base : VAR SSUPS_STATE_BASE

  SSUPS_STATE(base): bool =
    disjoint?(avail_port(base), handles(base))
        AND disjoint?(avail_port(base), service_port(base))
        AND subset?(existing_threads(kst(base)), ssups_threads)

  st1, st2: VAR (SSUPS_STATE)

  ssups_view(st1,st2) : bool =
    st1 = st2

  END ssups_state
```

## THEORY *ssups_state_witness*

```
ssups_state_witness: THEORY

BEGIN

  IMPORTING ssups_state


  ssups_state_witness: (SSUPS_STATE) =
    (# avail_port := emptyset[NAME],
        service_port := emptyset[NAME],
        sit_pfs := (LAMBDA (s : SITUATION): emptyset[PROT_FAMILY]),
        handles := emptyset[NAME],
        handle_pf := (LAMBDA (h: (emptyset[NAME])): null_prot_family),
        kst := empty_kst
       #)

  ssups_state_witness_prop : THEOREM
        (EXISTS (s : (SSUPS_STATE)) : TRUE)

END ssups_state_witness
```

## 24.2 Operations

This section describes the subset of SSUPS operations that are relevant to this example.

*Editorial Note:*
This section currently describes only successful processing of requests.

We first define several utility functions:

- *ssups_static* — defines the following state invariants: *service_port*, *sit_pfs* and *existing_threads* do not change.
- *ssups_step* — SSUPS transitions obey *ssups_static* and only make allowed modifications to kernel state.
- $ssups\_receive\_request\_util(thread, ri, op\_id, perm, st_1, st_2)$ — *thread* checks permission *perm* and operation *op_id* on the received information in *ri* and then uses *process_request* to mark *ri* as processed.

## THEORY *ssups_ops_base*

```
ssups_ops_base: THEORY
  BEGIN

  IMPORTING ssups_state

  IMPORTING dtos_kernel_shared_ops

  %%This should probably be in dtos_kernel_shared_ops
  IMPORTING messaging

  st, st1, st2: VAR (SSUPS_STATE)

  %%local state invariants
  ssups_static(st1, st2): bool =
    sit_pfs(st2) = sit_pfs(st1)
        AND service_port(st2) = service_port(st1)
              AND existing_threads(kst(st2)) = existing_threads(kst(st1))

  %a step must obey local invariants and only make allowed
  % mods to kernel state.
  ssups_step(st1, st2): bool =
    ssups_static(st1, st2)
        AND effects_on_kernel_state(kst(st1), kst(st2), ssups_threads)

  thread: VAR THREAD

  prot_family: VAR PROT_FAMILY

  ri: VAR RECEIVED_INFO

  op_id: VAR OP

  perm: VAR PERMISSION

  name, reply_port, to: VAR NAME

  kernel_req: VAR KERNEL_REQ

  msg: VAR USER_MSG


  % UTILITY FUNCTIONS

  % processing a newly received request
  ssups_receive_request_util(thread, ri, op_id, perm, st1, st2): bool =
    receive_request(thread, ri, op_id, perm, kst(st1), kst(st2))
```

```
  END ssups_ops_base
```

At any time when the SSUPS has a thread that is not already waiting for a message operation to be performed, that thread can request to receive a message from a port. The thread initiates this processing by setting its pending request to be a message receive and changing its state to $thread\_waiting$.

## THEORY *ssups_receive_request*

```
ssups_receive_request: THEORY
  BEGIN

  IMPORTING ssups_ops_base

  st1, st2: VAR (SSUPS_STATE)

  thread: VAR (ssups_threads)

  name: VAR NAME

  ssups_receive_request_submit(st1, st2, thread): bool =
    EXISTS name:
      receive_msg(kst(st1), kst(st2), thread, name)

  ssups_receive_request(st1, st2, thread): bool =
   avail_port(st2) = avail_port(st1)
        AND handles(st2) = handles(st1)
        AND handle_pf(st2) = handle_pf(st1)
        AND ssups_receive_request_submit(st1, st2, thread)

  END ssups_receive_request
```

When the SSUPS receives a valid $retrieve\_prot\_family\_op$ request on one of its handles $h$, it responds by sending a $provide\_prot\_family\_op$ message to the reply port containing the protection family associated with $h$.

## THEORY *ssups_retrieve_prot_family*

```
ssups_retrieve_prot_family: THEORY
  BEGIN

  IMPORTING ssups_ops_base


  % VARIABLES

  st1, st2: VAR (SSUPS_STATE)

  thread: VAR THREAD
```

```
prot_family : VAR PROT_FAMILY

ri: VAR RECEIVED_INFO

handle : VAR NAME


ssups_retrieve_prot_family(st1, st2, thread): bool =
  (EXISTS ri, prot_family, handle:

        ssups_receive_request_util(thread, ri, retrieve_prot_family_op,
                                   retrieve_prot_family_perm, st1, st2)
        AND handle = service_port(ri)
        AND handles(st1)(handle)
        AND prot_family = handle_pf(st1)(handle)

        AND existing_threads(kst(st2)) = existing_threads(kst(st1))

        AND avail_port(st2) = avail_port(st1)
        AND handles(st2) = handles(st1)
        AND handle_pf(st2) = handle_pf(st1)

        AND send_msg(kst(st1), kst(st2), thread, reply_name(ri), provide_prot_family_op,
                                             null_name, provide_prot_family_msg(prot_family))

  )

END ssups_retrieve_prot_family
```

When a key server thread receives a valid $select\_prot\_family\_op$ request containing situation $sit$ and protection family $prot\_family$ (where $prot\_family$ is in $sit\_pfs(sit)$) on one of its handles $h$, it

- allocates a handle $h$,
- associates $prot\_family$ with $h$, and
- sends a $provide\_pf\_handle\_op$ message to the reply port containing the handle $h$.

## THEORY *ssups_select_prot_family*

```
ssups_select_prot_family: THEORY
  BEGIN

  IMPORTING ssups_ops_base


  % VARIABLES

  st1, st2: VAR (SSUPS_STATE)

  thread: VAR THREAD

  prot_family : VAR PROT_FAMILY
```

```
sit: VAR SITUATION

ri: VAR RECEIVED_INFO

handle : VAR NAME


ssups_select_prot_family(st1, st2, thread): bool =
  (EXISTS ri, sit, prot_family, handle:

        ssups_receive_request_util(thread, ri, select_prot_family_op,
                                   select_prot_family_perm, st1, st2)
        AND select_prot_family_msg(sit, prot_family) = user_msg(ri)
        AND service_port(st1)(service_port(ri))
        AND sit_pfs(st1)(sit)(prot_family)

        AND existing_threads(kst(st2)) = existing_threads(kst(st1))

        AND avail_port(st1)(handle)
        AND avail_port(st2) = remove(handle, avail_port(st1))
        AND handles(st2) = add(handle, handles(st1))
        AND handle_pf(st2) = handle_pf(st1) WITH [ handle := prot_family ]

        AND send_msg(kst(st1), kst(st2), thread, reply_name(ri), provide_pf_handle_op,
                                             null_name, provide_pf_handle_msg(handle))

  )

END ssups_select_prot_family
```

An SSUPS operation consists of any one of the operations defined above. The $guar$ of the SSUPS consists of those transitions with an SSUPS thread serving as the agent such that the start and final states of the transition satisfy $ssups\_step$ and $ssups\_op$ or look the same with respect to $ssups\_view$.

## THEORY *ssups_ops*

```
ssups_ops: THEORY
  BEGIN

  IMPORTING ssups_receive_request
  IMPORTING ssups_select_prot_family
  IMPORTING ssups_retrieve_prot_family

  st1, st2 : VAR (SSUPS_STATE)
  thread : VAR (ssups_threads)
  ag: VAR THREAD

  ssups_op(st1, st2, thread) : bool =
    ssups_receive_request(st1, st2, thread)
        OR   ssups_select_prot_family(st1, st2, thread)
        OR   ssups_retrieve_prot_family(st1, st2, thread)


  ssups_guar(st1,st2,ag) : bool =
```

```
  ssups_threads(ag) AND
        (ssups_view(st1, st2)
          OR (ssups_step(st1, st2) AND ssups_op(st1, st2, ag)))

 END ssups_ops
```

## 24.3 Environment Assumptions

The environment of the SSUPS is assumed to alter no SSUPS state information other than *kst* and to obey the constraints on changing *kst* that are given in *environment_base* on page 140. The *hidd* of the SSUPS is defined similarly using *hidd_base*.

### THEORY *ssups_rely*

```
ssups_rely : THEORY

BEGIN

IMPORTING dtos_kernel_shared_rely

IMPORTING ssups_state

st1, st2 : VAR (SSUPS_STATE)

ag : VAR THREAD

ssups_environment(st1,st2,ag) : bool =
      environment_base(ag,kst(st1),kst(st2)) and
      st1 with [kst := kst(st2)] = st2

ssups_environment_refl: THEOREM
      ssups_environment(st1,st1,ag)

ssups_hidd(st1,st2,ag) : bool =
      NOT ssups_threads(ag)
            AND hidd_base(ag, kst(st1), kst(st2))
            AND st2 = st1 WITH [ kst := kst(st2) ]

ssups_hidd_prop: THEOREM
      ssups_hidd(st1,st2,ag)
            => k_threads(ag) OR ssups_view(st1,st2)


ssups_rely(st1,st2,ag) : bool =
      not ssups_threads(ag) AND
      ssups_environment(st1,st2,ag)

END ssups_rely
```

## 24.4 Component Specification

We use the set *initial_ssups_states* to denote the valid initial states for the SSUPS. A valid initial state has the following properties:

- There are no handles in use.

- No kernel requests are pending for any SSUPS thread and no messages are waiting to be processed.

The SSUPS is a component having state type $SSUPS\_STATE$, satisfying initial constraint $initial\_ssups\_states$, and executing only the transitions defined in Section 24.2.

## THEORY *ssups_spec*

```
ssups_spec : THEORY
BEGIN

IMPORTING dtos_kernel_shared_state
IMPORTING ssups_ops
IMPORTING ssups_rely
IMPORTING ssups_state_witness
IMPORTING component_aux[(SSUPS_STATE),THREAD]

st, st1, st2 : VAR (SSUPS_STATE)
ag : VAR THREAD
thread : VAR (ssups_threads)

initial_ssups_states(st) : bool =
  handles(st) = emptyset[NAME]
    AND pending_requests(kst(st)) = emptyset[KERNEL_REQ]
    AND (FORALL ag :
        existing_threads(kst(st))(ag) =>
                 ri_status(received_info(kst(st))(ag)) = ri_processed)

ssups_state_witness_initial: THEOREM
  initial_ssups_states(ssups_state_witness)

base_ssups_comp : base_comp_t =
     (# init := initial_ssups_states,
        guar := ssups_guar,
        rely := ssups_rely,
        hidd := ssups_hidd,
        cags := ssups_threads,
        view := ssups_view,
        wfar := emptyset[TRANSITION_CLASS[(SSUPS_STATE), THREAD]],
        sfar := emptyset[TRANSITION_CLASS[(SSUPS_STATE), THREAD]] #)

  ssups_view_eq: THEOREM view_eq(base_ssups_comp)

  ssups_comp_init: THEOREM init_restriction(base_ssups_comp)

  ssups_comp_guar: THEOREM guar_restriction(base_ssups_comp)

  ssups_comp_rely_hidd: THEOREM rely_hidd_restriction(base_ssups_comp)

  ssups_comp_hidd: THEOREM hidd_restriction(base_ssups_comp)

  ssups_comp_rely: THEOREM rely_restriction(base_ssups_comp)

  ssups_comp_cags: THEOREM cags_restriction(base_ssups_comp)

  ssups_comp_guar_stuttering: THEOREM guar_stuttering_restriction(base_ssups_comp)

  ssups_comp_rely_stuttering: THEOREM rely_stuttering_restriction(base_ssups_comp)

  ssups_comp : (comp_t) = base_ssups_comp
```

```
  ssups_comp_hidd_prop: THEOREM
    hidd(ssups_comp)(st1, st2, ag)
    => k_threads(ag) OR view(ssups_comp)(st1, st2)

END ssups_spec
```

60

# Section 25
# Cryptographic Client

This section describes a component acting as a client of the Synergy Cryptographic Subsystem. We will call this the Client component. The Client interacts with the SSUPS to obtain a handle for a selected a protection family. It then provides this handle in a request to the CC to establish a cryptographic context. Once the protection handle is received back from the CC, the client can send protection requests to the handle and receive the cyphertext in the reply messages. Since we are focusing on the Crypto Subsystem itself rather then on application programs that require encryption services, we just specify that the client stores the cyphertext in its state rather than modeling actions such as sending the data across the network or writing it to encrypted media.

## 25.1 State

We model the state of the Client component in the same way as the PT and KS components. That is, we have multiple client threads functioning intuitively as separate Client subcomponents within the actual Client component.

Each individual client maintains the following pieces of state information:

- $reply\_port$ — reply port name supplied in the client's requests to the subsystem,
- $situation$ — situation in which it is operating,
- $requested\_prot\_family$ — protection family that it requests to use,
- $ssups$ — client's name for an SSUPS service port,
- $cc$ — client's name for a CC service port,
- $pf\_handle\_provided$ — a boolean flag indicating whether a protection family handle has been received from the SSUPS (and forwarded to the CC),
- $handle$ — a handle received from CC for use in encrypting data according to the established cryptographic context,
- $clear\_text\_sent$ — the text most recently sent to $handle$ in a protection request. This is $null\_text$ until the first protection request has been sent.
- $reply\_received$ — a boolean flag, true if the cypher text has been received for the most recently sent protection request. Should be true if no protection requests have been sent yet.
- $cypher\_text\_received$ — the most recently received cypher text. Should be $null\_text$ until a reply to the first protection request has been processed.

All of the above state information, defined by the type $CLIENT\_THREAD\_STATE$, is maintained by each element of $client\_threads$. All values of type $CLIENT\_THREAD\_STATE$ are considered valid. A Client state consists of a mapping $thst$ from the $client\_threads$ to their associated $CLIENT\_THREAD\_STATE$ as well as the $KERNEL\_SHARED\_STATE$, $kst$. The latter contains $existing\_threads$, $pending\_requests$, $thread\_status$, and $received\_info$. The valid states for the Client component are defined by $CLIENT\_STATE$. In a valid state, the $existing\_threads$ must be a subset of $client\_threads$.

All the data in $CLIENT\_STATE$ is visible to the client.

## THEORY *client_state*

```
client_state : THEORY
  BEGIN

  IMPORTING crypto_shared_state

  client_threads: (nonempty?[THREAD])

  client_threads_witness: (client_threads)

  client_threads_nonempty: THEOREM client_threads /= emptyset


  % Each client thread can have a situation, selected protection family, crypto
  % handle and active protection request.
  CLIENT_THREAD_STATE_BASE : TYPE =
     [#
        reply_port: NAME,              % where I wait for replies

        situation : SITUATION,   % my situation

        requested_prot_family : PROT_FAMILY,   % pf I requested

        ssups : NAME,                  % my name for an SSUPS service port

        cc : NAME,                     % my name for a crypto controller service port

        pf_handle_provided: bool,                        % have I received (and forwarded)
                                               % a pf port?

        handle: NAME,                          % crypto handle for my pf

        clear_text_sent : TEXT,        % most recent text that I asked to have encrypted
                                       % Should be null_text until first protection request
                                       % is sent.

        reply_received : bool,         % have I received back the cypher text for my
                                       % most recent protection request? Should be
                                       % true if no protection requests have been sent
                                       % yet.

        cypher_text_received: TEXT             % most recently received cypher text
                                               % Should be null_text until reply
                                               % to first protection request is received.

        #]

  thstate : VAR CLIENT_THREAD_STATE_BASE

  CLIENT_THREAD_STATE(thstate) : bool = true
```

```
CLIENT_STATE_BASE : TYPE =
    [# thst : [(client_threads) -> (CLIENT_THREAD_STATE)],

        kst: KERNEL_SHARED_STATE
        #]

base : VAR CLIENT_STATE_BASE

CLIENT_STATE(base): bool =
  subset?(existing_threads(kst(base)), client_threads)

st1, st2: VAR (CLIENT_STATE)

client_view(st1,st2) : bool =
  st1 = st2

END client_state
```

## THEORY *client_state_witness*

```
client_state_witness: THEORY

BEGIN

  IMPORTING client_state

  th: VAR (client_threads)


  client_thread_state_witness: (CLIENT_THREAD_STATE) =
    (# reply_port := epsilon(fullset[NAME]),
       situation := epsilon(fullset[SITUATION]),
       requested_prot_family := epsilon(fullset[PROT_FAMILY]),
       ssups := epsilon(fullset[NAME]),
       cc := epsilon(fullset[NAME]),
       pf_handle_provided := false,
       handle := null_name,
       clear_text_sent := null_text,
       reply_received := true,
       cypher_text_received := null_text
       #)


  client_state_witness: (CLIENT_STATE) =
    (# thst := (LAMBDA th : client_thread_state_witness),
       kst := empty_kst
       #)

  client_state_witness_prop : THEOREM
        (EXISTS (base : (CLIENT_STATE)) : TRUE)

END client_state_witness
```

## 25.2 Operations

This section describes the subset of Client operations that are relevant to this example.

---

*Editorial Note:*
This section currently describes only successful processing of requests.

---

We first define several utility functions:

- *client_static* — defines the following state invariants: *reply_port*, *situation*, *requested_prot_family*, *ssups* and *cc* do not change for any thread and *existing_threads* does not change.
- *client_step* — Client transitions obey *client_static* and make only allowed modifications to kernel state, and no client can change the *thst* of any other client thread.
- $client\_receive\_request\_util(thread, ri, op\_id, perm, st_1, st_2)$ — *thread* checks permission *perm* and operation *op_id* on the received information in *ri* and then uses *process_request* to mark *ri* as processed.

## THEORY *client_ops_base*

---

```
client_ops_base: THEORY
  BEGIN

  IMPORTING client_state

  IMPORTING dtos_kernel_shared_ops

  IMPORTING messaging

  st1, st2: VAR (CLIENT_STATE)

  th, th1, th2 : VAR (client_threads)

  thread: VAR THREAD

  %%local state invariants
  client_static(st1, st2): bool =
    (FORALL th: reply_port(thst(st2)(th)) = reply_port(thst(st1)(th))
          AND situation(thst(st2)(th)) = situation(thst(st1)(th))
          AND requested_prot_family(thst(st2)(th)) = requested_prot_family(thst(st1)(th))
          AND ssups(thst(st2)(th)) = ssups(thst(st1)(th))
          AND cc(thst(st2)(th)) = cc(thst(st1)(th)))
    AND existing_threads(kst(st2)) = existing_threads(kst(st1))

  %a step must obey local invariants and only make allowed
  % mods to kernel state or its own thst.
  client_step(st1, st2, thread): bool =
    client_static(st1, st2)
          AND effects_on_kernel_state(kst(st1), kst(st2), client_threads)
          AND (FORALL th:
                  (NOT (th = thread) IMPLIES
                          thst(st1)(th) = thst(st2)(th)))
```

```
  ri: VAR RECEIVED_INFO

  op_id: VAR OP

  perm: VAR PERMISSION


  % UTILITY FUNCTIONS


  % processing a newly received request
  client_receive_request_util(thread, ri, op_id, perm, st1, st2): bool =
     receive_request(thread, ri, op_id, perm, kst(st1), kst(st2))

  END client_ops_base
```

At any time when the Client has a thread that is not already waiting for a message operation to be performed, that thread can request to receive a message from a port. The thread initiates this processing by setting its pending request to be a message receive and changing its state to $thread\_waiting$.

## THEORY *client_receive_request*

```
client_receive_request: THEORY
  BEGIN

  IMPORTING client_ops_base

  st1, st2: VAR (CLIENT_STATE)

  thread: VAR (client_threads)

  name: VAR NAME

  client_receive_request_submit(st1, st2, thread): bool =
     EXISTS name:
        receive_msg(kst(st1), kst(st2), thread, name)

  client_receive_request(st1, st2, thread): bool =
   thst(st2) = thst(st1)
          AND client_receive_request_submit(st1, st2, thread)

  END client_receive_request
```

If for some thread $th$, $handle(th)$ is not $null\_name$ and $reply\_received(th)$ is true (i.e., a handle has been obtained, and the thread is not waiting for an encryption request to complete), it may send a $protect\_op$ request to the handle containing clear text $text$ and destination $reply\_port(th)$. The $text$ is stored in $clear\_text\_sent(th)$ and $reply\_received(th)$ is set to false.

## THEORY *client_protect*

```
client_protect: THEORY
  BEGIN

  IMPORTING client_ops_base


  % VARIABLES

  st1, st2: VAR (CLIENT_STATE)

  thread: VAR (client_threads)

  text : VAR TEXT

  client_protect(st1, st2, thread): bool =
    (EXISTS text :
          handle(thst(st1)(thread)) /= null_name
          AND reply_received(thst(st1)(thread))

          AND thst(st2)(thread) = thst(st1)(thread)
                  WITH [clear_text_sent := text,
                        reply_received := false]

          AND send_msg(kst(st1), kst(st2), thread, handle(thst(st1)(thread)), protect_op,
                   reply_port(thst(st1)(thread)),
                   protect_msg(text, reply_port(thst(st1)(thread))))
    )

  END client_protect
```

When a client thread $th$ receives a $provide\_crypto\_context\_op$ message containing $crypto\_handle$, it stores the handle in $handle(th)$. This transition is only enabled when $handle(th)$ is $null\_name$ and $clear\_text\_sent(th)$ is $null\_text$. Thus, we only consider clients that set up and use at most one context.

## THEORY *client_provide_crypto_context*

```
client_provide_crypto_context: THEORY
  BEGIN

  IMPORTING client_ops_base


  % VARIABLES

  st1, st2: VAR (CLIENT_STATE)

  thread: VAR (client_threads)

  ri : VAR RECEIVED_INFO

  crypto_handle : VAR NAME

  client_provide_crypto_context(st1, st2, thread): bool =
    (EXISTS ri, crypto_handle :
```

```
        client_receive_request_util(thread, ri, provide_crypto_context_op,
                                    provide_crypto_context_perm, st1, st2)
        AND handle(thst(st1)(thread)) = null_name
        AND clear_text_sent(thst(st1)(thread)) = null_text
        AND provide_crypto_context_msg(crypto_handle) = user_msg(ri)

        AND thst(st2)(thread) = thst(st1)(thread) WITH [handle := crypto_handle]

    )

  END client_provide_crypto_context
```

When a client thread $th$ receives a $provide\_pf\_handle\_op$ message containing $pf\_handle$, it

- sets $pf\_handle\_provided(th)$ to true, and
- sends a $create\_crypto\_context\_op$ request to $cc(th)$ containing its $situation$, the $pf\_handle$ and its $requested\_prot\_family$.

This transition is only enabled when $pf\_handle\_provided(th)$ is false and $clear\_text\_sent(th)$ is $null\_text$.

## THEORY *client_provide_pf_handle*

```
client_provide_pf_handle: THEORY
  BEGIN

  IMPORTING client_ops_base


  % VARIABLES

  st1, st2: VAR (CLIENT_STATE)

  thread: VAR (client_threads)

  ri : VAR RECEIVED_INFO

  pf_handle : VAR NAME

  client_provide_pf_handle(st1, st2, thread): bool =
    (EXISTS ri, pf_handle :

        client_receive_request_util(thread, ri, provide_pf_handle_op,
                                    provide_pf_handle_perm, st1, st2)
        AND pf_handle_provided(thst(st1)(thread)) = false
        AND clear_text_sent(thst(st1)(thread)) = null_text
        AND provide_pf_handle_msg(pf_handle) = user_msg(ri)

        AND thst(st2)(thread) = thst(st1)(thread) WITH [pf_handle_provided := true]

        AND send_msg(kst(st1), kst(st2), thread, cc(thst(st1)(thread)), create_crypto_context_op,
                 reply_port(thst(st1)(thread)),
                 create_crypto_context_msg(situation(thst(st1)(thread)),
                                           pf_handle,
                                           requested_prot_family(thst(st1)(thread))))
```

```
  )

  END client_provide_pf_handle
```

When a client thread $th$ receives a $provide\_protected\_data\_op$ message containing $text$, it

- sets $reply\_received(th)$ to true, and
- sets $cypher\_text\_received$ to $text$.

This transition is only enabled when $reply\_received(th)$ is false.

## THEORY *client_provide_protected_data*

```
client_provide_protected_data: THEORY
  BEGIN

  IMPORTING client_ops_base


  % VARIABLES

  st1, st2: VAR (CLIENT_STATE)

  thread: VAR (client_threads)

  ri : VAR RECEIVED_INFO

  text : VAR TEXT


  client_provide_protected_data(st1, st2, thread): bool =
    (EXISTS ri, text :

          client_receive_request_util(thread, ri, provide_protected_data_op,
                                      provide_protected_data_perm, st1, st2)
          AND NOT reply_received(thst(st1)(thread))
          AND provide_protected_data_msg(text) = user_msg(ri)

          AND thst(st2)(thread) = thst(st1)(thread)
                  WITH [reply_received := true,
                        cypher_text_received := text]

    )

  END client_provide_protected_data
```

At any time a client thread may send a $select\_prot\_family\_op$ request to $ssups(th)$ containing its $situation$ and $requested\_prot\_family$.

## THEORY *client_select_prot_family*

```
client_select_prot_family: THEORY
  BEGIN

  IMPORTING client_ops_base


  % VARIABLES

  st1, st2: VAR (CLIENT_STATE)

  thread: VAR (client_threads)

  client_select_prot_family(st1, st2, thread): bool =
        thst(st2)(thread) = thst(st1)(thread) AND
        send_msg(kst(st1), kst(st2), thread, ssups(thst(st1)(thread)), select_prot_family_op,
                 reply_port(thst(st1)(thread)),
                 select_prot_family_msg(situation(thst(st1)(thread)),
                                        requested_prot_family(thst(st1)(thread))))


  END client_select_prot_family
```

A Client operation consists of any one of the operations defined above. The *guar* of the Client consists of those transitions with a Client thread serving as the agent such that the start and final states of the transition satisfy *client_step* and *client_op* or look the same with respect to *client_view*.

## THEORY *client_ops*

```
client_ops: THEORY
  BEGIN

  IMPORTING client_receive_request
  IMPORTING client_select_prot_family
  IMPORTING client_provide_pf_handle
  IMPORTING client_provide_crypto_context
  IMPORTING client_protect
  IMPORTING client_provide_protected_data

  st1, st2 : VAR (CLIENT_STATE)
  thread : VAR THREAD
  th: VAR (client_threads)

  client_op(st1, st2, th) : bool =
    client_receive_request(st1, st2, th)
        OR client_select_prot_family(st1, st2, th)
        OR client_provide_pf_handle(st1, st2, th)
        OR client_provide_crypto_context(st1, st2, th)
        OR client_protect(st1, st2, th)
        OR client_provide_protected_data(st1, st2, th)


  client_guar(st1,st2,thread) : bool =
    client_threads(thread) AND
    (client_view(st1,st2)
        OR (client_step(st1, st2, thread) AND
             client_op(st1, st2, thread)))
```

```
END client_ops
```

## 25.3 Environment Assumptions

The environment of the Client is assumed to alter no Client state information other than $kst$ and to obey the constraints on changing $kst$ that are given in $environment\_base$ on page 140. The $hidd$ of the Client is defined similarly using $hidd\_base$.

## THEORY *client_rely*

```
client_rely : THEORY

BEGIN

IMPORTING dtos_kernel_shared_rely

IMPORTING client_state

st1, st2 : VAR (CLIENT_STATE)

ag : VAR THREAD

client_environment(st1,st2,ag) : bool =
    environment_base(ag,kst(st1),kst(st2)) and
    st1 with [kst := kst(st2)] = st2

client_environment_refl: THEOREM
    client_environment(st1,st1,ag)

client_hidd(st1,st2,ag) : bool =
    NOT client_threads(ag)
        AND hidd_base(ag, kst(st1), kst(st2))
        AND st2 = st1 with [ kst := kst(st2) ]

client_hidd_prop: THEOREM
    client_hidd(st1,st2,ag)
        => k_threads(ag) OR client_view(st1,st2)


client_rely(st1,st2,ag) : bool =
    not client_threads(ag) AND
    client_environment(st1,st2,ag)

END client_rely
```

## 25.4 Component Specification

We use the set $initial\_client\_states$ to denote the valid initial states for the Client. A valid initial state has the following properties:

- For each thread, its $pf\_handle\_provided$ is false, its $handle$ is $null\_name$, its $clear\_text\_sent$ is $null\_text$ and its $reply\_received$ is true.

- No kernel requests are pending for any Client thread and no messages are waiting to be processed.

A Client is a component having state type $CLIENT\_STATE$, satisfying initial constraint $initial\_client\_states$, and executing only the transitions defined in Section 25.2.

## THEORY *client_spec*

```
client_spec : THEORY
BEGIN

IMPORTING dtos_kernel_shared_state
IMPORTING client_ops
IMPORTING client_rely
IMPORTING client_state_witness
IMPORTING component_aux[(CLIENT_STATE),THREAD]

st, st1, st2 : VAR (CLIENT_STATE)
ag : VAR THREAD
thread : VAR (client_threads)

initial_client_states(st) : bool =
  (FORALL thread:
    pf_handle_provided(thst(st)(thread)) = false
    AND handle(thst(st)(thread)) = null_name
    AND clear_text_sent(thst(st)(thread)) = null_text
    AND reply_received(thst(st)(thread)) = true
    AND pending_requests(kst(st)) = emptyset[KERNEL_REQ]
    AND (FORALL ag :
        existing_threads(kst(st))(ag) =>
                  ri_status(received_info(kst(st))(ag)) = ri_processed))

client_state_witness_initial: THEOREM
  initial_client_states(client_state_witness)

base_client_comp : base_comp_t =
     (# init := initial_client_states,
        guar := client_guar,
        rely := client_rely,
        hidd := client_hidd,
        cags := client_threads,
        view := client_view,
        wfar := emptyset[TRANSITION_CLASS[(CLIENT_STATE), THREAD]],
        sfar := emptyset[TRANSITION_CLASS[(CLIENT_STATE), THREAD]] #)

  client_view_eq: THEOREM view_eq(base_client_comp)

  client_comp_init: THEOREM init_restriction(base_client_comp)

  client_comp_guar: THEOREM guar_restriction(base_client_comp)

  client_comp_rely_hidd: THEOREM rely_hidd_restriction(base_client_comp)

  client_comp_hidd: THEOREM hidd_restriction(base_client_comp)

  client_comp_rely: THEOREM rely_restriction(base_client_comp)

  client_comp_cags: THEOREM cags_restriction(base_client_comp)

  client_comp_guar_stuttering: THEOREM guar_stuttering_restriction(base_client_comp)
```

```
client_comp_rely_stuttering: THEOREM rely_stuttering_restriction(base_client_comp)

client_comp : (comp_t) = base_client_comp

client_comp_hidd_prop: THEOREM
  hidd(client_comp)(st1, st2, ag)
  => k_threads(ag) OR view(client_comp)(st1, st2)

END client_spec
```

---

To demonstrate the use of the framework for reasoning about components and systems we define and prove a simple lemma about the Client component. This lemma will later be "lifted" to a lemma regarding the entire system. The lifted lemma would constitute one small part of the entire proof that the subsystem correctly encrypts data that it receives in protection requests. This overall system property is formalized as the state predicate *correct_encryption_pred*. Informally, it requires that, in any state in which a client thread

- has non-null text in *clear_text_sent* and
- has set *reply_received* to true,

the text in *cypher_text_received* is an encryption of the clear text with respect to the protection family requested by the thread. Note that this property can be entirely stated in terms of the Client state. This makes some sense in that the Crypto Subsystem is essentially a black box from the Client's perspective, yet the Client should, in principle, be able to independently verify that the encryption is correct.

It is easy to prove (theorem *correct_encryption_prop*1) that this predicate is satisfied in the initial state since *clear_text_sent* is required to be *null_text* in the initial state of the Client (i.e., *have_encrypted_text* is not satisfied in the initial state). We do not even need to consider any system components other than Client. However, proving that the predicate is preserved during transitions cannot be done without considering all of the system components. We must show that every *provide_protected_data_op* message sent to the Client will contain the correct data. This requires proving properties of each system component, applying the appropriate composition theorem to lift each of those properties to be a system property, and then showing that the lifted properties imply that *correct_encryption_pred* is preserved by all transitions.

Since our goal in this report is only to demonstrate and test the use of the framework, we will not perform this entire analysis. We will instead focus on one small lemma. We will show that if we assume that all *provide_protected_data_op* messages received by the client contain correct encryptions then *correct_encryption_pred* is always true. This is formalized in theorem *correct_encryption_prop*. This theorem is an easy consequence of *correct_encryption_prop*1 and *correct_encryption_prop_steps*. The latter constitutes the heart of the argument. It essentially shows that whenever the Client component processes a *provide_protected_data_op* message it correctly extracts the cypher text and stores it in *cypher_text_received*.

## THEORY *client_props*

---

```
client_props: THEORY
```

```
BEGIN

  IMPORTING client_spec

  IMPORTING more_preds[(CLIENT_STATE), THREAD ]

  IMPORTING unity

  st, st1, st2 : VAR (CLIENT_STATE)

  pf: VAR PROT_FAMILY

  clear, cypher: VAR TEXT

  p : VAR FSEQ[[ENCRYPT_MECH, KEY]]

  seed : VAR SEED

  key_mech : VAR KEY_MECH

  th: VAR (client_threads)

  t: VAR TEXT

  ri: VAR RECEIVED_INFO

  %% Next two functions probably belong in crypto_shared_state.pvs
  map_protect(p, t): RECURSIVE TEXT =
    IF nonemptyfseq(p) THEN
        map_protect(pop(p),protect_text(PROJ_1(elem(p)(1)), PROJ_2(elem(p)(1)), t))
    ELSE t
    ENDIF
   MEASURE size(p);

  encrypted_with_pf(pf, clear, cypher): bool =
    (EXISTS p:
        cypher = map_protect(p,clear)
        AND size(p) = size(pf)
        AND (FORALL (i: {i: nat | i>0 AND i <= size(pf)}):
              PROJ_1(elem(p)(i)) = encrypt_mech(elem(pf)(i))
              AND (EXISTS seed, key_mech:
                    PROJ_2(elem(p)(i)) = generate_key(key_mech, seed)
                    AND key_mech = key_mech(elem(pf)(i)))))



  have_encrypted_text(st, pf, clear, cypher): bool =
    EXISTS th:
        requested_prot_family(thst(st)(th)) = pf
         AND pf_handle_provided(thst(st)(th))
         AND handle(thst(st)(th)) /= null_name
         AND clear_text_sent(thst(st)(th)) = clear
         AND clear /= null_text
         AND cypher_text_received(thst(st)(th)) = cypher
         AND reply_received(thst(st)(th))



  %%vvv DESIRED PROPERTY vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
  %%
  %% This is the top-level desired property. We could try
  %% to prove it by decomposing it into other properties
  %% which eventually reduce to things you would prove about
  %% a single component (using the composition theorems to
```

```
%% lift each result).

correct_encryption_pred : STATE_PRED =
  (LAMBDA st:
     (FORALL pf, clear, cypher:
       have_encrypted_text(st, pf, clear, cypher)
       => encrypted_with_pf(pf, clear, cypher)))

%% We can prove that it is satisfied in the initial state without
%% considering any component other than the client.

correct_encryption_prop1: THEOREM
       init_satisfies(client_comp, correct_encryption_pred)

%% However, we cannot prove that the system steps satisfy
%% correct_encryption_pred without considering properties of the
%% entire system.
%%
%%^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^


%% To demonstrate property lifting, we prove a lemma that says if
%% the ri of the client always has things that are correct
%% encryptions then encrypted_with_pf is satisfied

correct_ppd_def(st): bool =
  (FORALL th, ri, cypher, pf, clear:
        (NOT reply_received(thst(st)(th))
            AND existing_threads(kst(st))(th)
            AND received_info(kst(st))(th) = ri
            AND op(ri) = provide_protected_data_op
            AND provide_protected_data_msg(cypher) = user_msg(ri)
            AND ri_status(ri) = ri_unprocessed
            AND pf = requested_prot_family(thst(st)(th))
            AND clear = clear_text_sent(thst(st)(th)))
        => encrypted_with_pf(pf, clear, cypher))

correct_ppd_pred: STATE_PRED =
  (LAMBDA st: correct_ppd_def(st))

correct_encryption_prop_steps: THEOREM
  steps_satisfy(client_comp,
                stable_assuming(correct_ppd_pred, correct_encryption_pred))

correct_encryption_prop: THEOREM
  satisfies(client_comp,
               pimplies(alwayss(correct_ppd_pred),
                        alwayss(correct_encryption_pred)))

END client_props
```

# Section 26 Composing the Components

To compose the components we must

- define a common state space,
- define state translators to translate the seven components into the common state space,
- show that the translators satisfy the requirements on translators
- apply the translators obtaining seven new components,
- show that the seven translated components are composable (i.e., that they have a common initial state), and
- apply the composition operator to obtain a specification of the system.

An agent translator is also needed in this process, but it is a simple identity translation since we use the same agent type.

Theory *system_state* defines the common state type along with several functions for dealing with it. The function *thset_func* maps each element of the set of component indices $COMP\_INDEX$ to the *cags* of the associated component. This function is used in axiom *thset_ax* to declare the *cags* sets to be disjoint.

The common state is $SYSTEM\_STATE$. It contains seven fields, one for each component of the system. The function *build_system_state_base* takes as arguments seven states of the respective types and returns a $SYSTEM\_STATE\_BASE$ record containing the states in the appropriate fields. In a valid system state, *stb*, the $KERNEL\_SHARED\_STATE$ within each of the component states is a *kst_substate* of the *ext_st* of the kernel component state $k(stb)$. Axiom *substates_ax* asserts that for every $SYSTEM\_STATE$, *stb*, there exists a $KERNEL\_SHARED\_STATE$, *superstate*, such that each $KERNEL\_SHARED\_STATE$ in *stb* is a substate of *superstate*. This axiom could be proven using axiom *thsets_ax*, but for simplicity we have just asserted it. This axiom is used in proving *ksts_mergable* which states that the value of *build_system_state_base* always contains a mergable set of $KERNEL\_SHARED\_STATE$ values for the seven components. We also assert as an axiom (*k_st_ax*) that for any $KERNEL\_SHARED\_STATE$, *kst*, there exists a $K\_STATE$ with an *ext_st* field equal to *kst*. Proving this would require that we show how to select an *int_st* consistent with any given *ext_st*. The theorem *build_system_state_base_prop* demonstrates that if the $K\_STATE$ supplied as the first argument of *build_system_state_base* has an *ext_st* which is a *kst_merge* of the $KERNEL_SHARED_STATE$s for any seven component states, then the value of *build_system_state_base* is a valid $SYSTEM\_STATE$. The function *build_system_state* captures this in its type declaration. This function is used later in the proofs that the translators satisfy the requirements on translators.

## THEORY *system_state*

*system_state*: **THEORY**

```
BEGIN

IMPORTING k_spec

IMPORTING cc_spec

IMPORTING pt_spec

IMPORTING ks_spec

IMPORTING ssups_spec

IMPORTING client_spec

IMPORTING security_server_spec

IMPORTING kst_merge

IMPORTING more_set_lemmas

IMPORTING disjoint_sets

th: VAR THREAD
thset, thset1, thset2: VAR setof[THREAD]
kst : VAR KERNEL_SHARED_STATE

COMP_INDEX : TYPE+ = {k_ind, cc_ind, pt_ind, ks_ind, ssups_ind, client_ind, ss_ind}

i: VAR COMP_INDEX

thsets_func: [COMP_INDEX -> setof[THREAD]] =
  (LAMBDA i:
    CASES i OF
      k_ind: k_threads,
      cc_ind: cc_threads,
      pt_ind: pt_threads,
      ks_ind: ks_threads,
      ssups_ind: ssups_threads,
      client_ind: client_threads,
      ss_ind: ss_threads
    ENDCASES)

thsets_ax: AXIOM
  pairwise_disjoint(thsets_func)

thsets_prop: LEMMA
  k_threads = thsets_func(k_ind)
  AND cc_threads = thsets_func(cc_ind)
  AND pt_threads = thsets_func(pt_ind)
  AND ks_threads = thsets_func(ks_ind)
  AND ssups_threads = thsets_func(ssups_ind)
  AND client_threads = thsets_func(client_ind)
  AND ss_threads = thsets_func(ss_ind)


%%% Define the composite state

SYSTEM_STATE_BASE:
    TYPE = [# k: (K_STATE),
              cc: (CC_STATE),
              pt: (PT_STATE),
              ks: (KS_STATE),
              ssups: (SSUPS_STATE),
              client: (CLIENT_STATE),
```

```
                    ss: (SS_STATE) #]

stb: VAR SYSTEM_STATE_BASE

k_st: VAR (K_STATE)

cc_st: VAR (CC_STATE)

pt_st: VAR (PT_STATE)

ks_st: VAR (KS_STATE)

ssups_st: VAR (SSUPS_STATE)

client_st: VAR (CLIENT_STATE)

ss_st: VAR (SS_STATE)

kstset : VAR setof[KERNEL_SHARED_STATE]

build_system_state_base(k_st, cc_st, pt_st, ks_st,
                         ssups_st, client_st, ss_st): SYSTEM_STATE_BASE =
  (# k:= k_st,
     cc:= cc_st,
     pt:= pt_st,
     ks:= ks_st,
     ssups:= ssups_st,
     client:= client_st,
     ss:= ss_st #)

all_ksts(stb) : setof[KERNEL_SHARED_STATE]
  = {kst | kst = ext_st(k(stb))
           OR kst = kst(cc(stb))
           OR kst = kst(pt(stb))
           OR kst = kst(ks(stb))
           OR kst = kst(ssups(stb))
           OR kst = kst(client(stb))
           OR kst = kst(ss(stb))}



SYSTEM_STATE(stb): bool =
  (FORALL kst:
     all_ksts(stb)(kst) => kst_substate(kst, ext_st(k(stb))))

%% This axiom could be proven from disjointness of threads for the
%% components.  For simplicity we will just assert that it is true.
substates_ax: AXIOM
  EXISTS (superstate: KERNEL_SHARED_STATE) :
    (FORALL kst:
       all_ksts(stb)(kst) => kst_substate(kst, superstate))

ksts_mergable: THEOREM
  kst_mergable(all_ksts(build_system_state_base(k_st,
                                                cc_st, pt_st,
                                                ks_st, ssups_st,
                                                client_st,
                                                ss_st)))

%% This could be proven by showing how to select int_st(k_st) so
%% that it is consistent with K_STATE requirements and with the kst
%% chosen.  Again, for simplicity we assert this.
k_st_ax: AXIOM
  (EXISTS k_st: ext_st(k_st) = kst)
```

```
build_system_state_base_prop: THEOREM
  SYSTEM_STATE(
      build_system_state_base(
              choose({k_st1 : (K_STATE) | ext_st(k_st1)
                      = kst_merge(all_ksts(build_system_state_base(
                                              k_st, cc_st, pt_st, ks_st,
                                              ssups_st, client_st, ss_st)))}),
              cc_st, pt_st, ks_st, ssups_st, client_st, ss_st))

build_system_state(k_st, cc_st, pt_st, ks_st,
                      ssups_st, client_st, ss_st): (SYSTEM_STATE) =
      build_system_state_base(
              choose({k_st1 : (K_STATE) | ext_st(k_st1)
                      = kst_merge(all_ksts(build_system_state_base(
                                              k_st, cc_st, pt_st, ks_st,
                                              ssups_st, client_st, ss_st)))}),
              cc_st, pt_st, ks_st, ssups_st, client_st, ss_st)

system_state_witness : (SYSTEM_STATE) =
   build_system_state_base(k_state_witness,
              cc_state_witness,
              pt_state_witness,
              ks_state_witness,
              ssups_state_witness,
              client_state_witness,
              ss_state_witness)

system_state_nonempty: THEOREM (EXISTS (x: ((SYSTEM_STATE))): TRUE)

END system_state
```

Theory *system_trans* defines the eight translators (seven state translator and one agent translator) needed in this example and demonstrates that each of them is a legal translator. It then defines the translated components.

## THEORY *system_trans*

```
system_trans: THEORY

  BEGIN

  IMPORTING system_state

  IMPORTING idtran

  IMPORTING tcprops

  st, st1, st2: VAR (SYSTEM_STATE)
  k_st : VAR (K_STATE)
  ccst : VAR (CC_STATE)
  ptst : VAR (PT_STATE)
  ksst : VAR (KS_STATE)
  ssupsst : VAR (SSUPS_STATE)
  clientst : VAR (CLIENT_STATE)
  ssst : VAR (SS_STATE)

%%% Define the translators
```

```
  k2system_sttran: (translator_t[(K_STATE), (SYSTEM_STATE)]) =
    (LAMBDA k_st: {st | k(st) = k_st})

  cc2system_sttran: (translator_t[(CC_STATE), (SYSTEM_STATE)]) =
    (LAMBDA ccst: {st | cc(st) = ccst})

  pt2system_sttran: (translator_t[(PT_STATE), (SYSTEM_STATE)]) =
    (LAMBDA ptst: {st | pt(st) = ptst})

  ks2system_sttran: (translator_t[(KS_STATE), (SYSTEM_STATE)]) =
    (LAMBDA ksst: {st | ks(st) = ksst})

  ssups2system_sttran: (translator_t[(SSUPS_STATE), (SYSTEM_STATE)]) =
    (LAMBDA ssupsst: {st | ssups(st) = ssupsst})

  client2system_sttran: (translator_t[(CLIENT_STATE), (SYSTEM_STATE)]) =
    (LAMBDA clientst: {st | client(st) = clientst})

  ss2system_sttran: (translator_t[(SS_STATE), (SYSTEM_STATE)]) =
    (LAMBDA ssst: {st | ss(st) = ssst})

  system_agtran: (translator_t[THREAD, THREAD]) = idt[THREAD]


%%% Translate the components

  kt: (comp_t[(SYSTEM_STATE), THREAD]) =
    tran_cmp(k_comp, k2system_sttran, system_agtran)

  cct: (comp_t[(SYSTEM_STATE), THREAD]) =
    tran_cmp(cc_comp, cc2system_sttran, system_agtran)

  ptt: (comp_t[(SYSTEM_STATE), THREAD]) =
    tran_cmp(pt_comp, pt2system_sttran, system_agtran)

  kst: (comp_t[(SYSTEM_STATE), THREAD]) =
    tran_cmp(ks_comp, ks2system_sttran, system_agtran)

  ssupst: (comp_t[(SYSTEM_STATE), THREAD]) =
    tran_cmp(ssups_comp, ssups2system_sttran, system_agtran)

  clientt: (comp_t[(SYSTEM_STATE), THREAD]) =
    tran_cmp(client_comp, client2system_sttran, system_agtran)

  sst: (comp_t[(SYSTEM_STATE), THREAD]) =
    tran_cmp(ss_comp, ss2system_sttran, system_agtran)

END system_trans
```

Theory *system* defines the component *system* as the composite of the seven translated components. To show that *system* is in fact a component we only need demonstrate a consistent starting state. This is demonstrated by the theorem *system_agreeable_start*.

Now that the entire system has been defined as the composite of the seven translated systems we want to reason about this composite. The composition framework provides a powerful tool for doing this: the composition theorem. This theorem allows us to "lift" properties of individual components to properties of the entire system. Before we can apply this theorem we must perform a *tolerance analysis* of the seven translated components. That is, we must show that no component violates the environment assumptions of any of its peer components (taking the *hidd* of the peer into consideration).

In the worst case, this introduces $O(n^2)$ proof obligations, each of which may be quite large, depending upon the complexity of the environment assumptions and the number of component operations that must be considered. Fortunately, in this case (and, we think, many others) many of these obligations can be collapsed into a small set of manageable proofs, and the worst case is not a problem. The key factor here is the *connectivity* of the components. That is, how directly do the components interact? In our example, all interactions are mediated through the kernel. If component $A$ can modify state that is visible to component $B$ then either $A$ or $B$ is the kernel. This reduces 30 obligations (the number of pairs of non-kernel components) to the six theorems with names of the form *system_tolerates_*t1*. Each of these can be proven based solely upon the $cags$ and $hidd$ of the components involved. For example, the proof of $system\_tolerates\_cct1$ relies upon the following:

- $hidd(cct)$ does not include any transitions by an agent not in $k\_threads$ that change its visible state.
- No non-kernel component has an agent in $k\_threads$.

Next we consider the six obligations to show that the kernel does not violate the assumptions of any other component. Here we are aided by the fact that for each non-kernel component we used $environment\_base$ and $hidd\_base$ to define the assumptions about the manipulation of the component's $KERNEL\_SHARED\_STATE$. Thus, we start by proving the lemma $kernel\_tolerance\_help$ showing that for any kernel transition $(a_1, a_2, b)$ and any $kst\_substates$, $kst_1$ and $kst_2$ of $a_1$ and $a_2$, respectively, if $hidd\_base$ allows agent $b$ to make a transition from $kst_1$ to $kst_2$ then so does $environment\_base$. This result is then used in proving the six theorems *system_tolerates_*t2*. The six theorems $system\_tolerates\_cct$, $system\_tolerates\_ptt$, $system\_tolerates\_kst$, $system\_tolerates\_ssupst$, $system\_tolerates\_clientt$ and $system\_tolerates\_sst$ combine the respective *system_tolerates_*t1* and *system_tolerates_*t2* results.

This leaves only the six obligations to show that all the non-kernel components satisfy the environment assumptions of the kernel. In this case our specification of kernel makes these obligations trivial. These tolerance obligations essentially have the form

$$guar(A) \cap hidd(k\_comp) \subseteq rely(k\_comp)$$

for each non-kernel component $A$. We have defined $hidd(k\_comp) = rely(k\_comp)$, making this obligation true independent of $A$. See Section 17.3 for a discussion of the advantages and disadvantages of doing this.

## THEORY *system*

```
system: THEORY
  BEGIN

  IMPORTING system_trans

  IMPORTING cmp_thm

  cmp: VAR (comp_t[(SYSTEM_STATE), THREAD])

  ag: VAR THREAD

  tran: VAR [(SYSTEM_STATE), (SYSTEM_STATE), THREAD]
```

```
a1, a2, kst1, kst2: VAR KERNEL_SHARED_STATE

b: VAR THREAD

system_cmps: setof[(comp_t[(SYSTEM_STATE), THREAD])] =
  {cmp | cmp = kt
         OR cmp = cct
         OR cmp = ptt
         OR cmp = kst
         OR cmp = ssupst
         OR cmp = clientt
         OR cmp = sst
  }

nonk_cmps: setof[(comp_t[(SYSTEM_STATE), THREAD])] =
  {cmp | cmp = cct
         OR cmp = ptt
         OR cmp = kst
         OR cmp = ssupst
         OR cmp = clientt
         OR cmp = sst
  }

nonk_cmps_ags: THEOREM
  (nonk_cmps(cmp) AND cags(cmp)(ag)
         => NOT tmap(system_agtran, k_threads)(ag))

system_union: LEMMA
  system_cmps = union(singleton(kt), nonk_cmps)

system_agreeable_start: THEOREM
  agreeable_start(system_cmps)

system_composable: THEOREM
     composable(system_cmps)

system: (comp_t[(SYSTEM_STATE), THREAD]) =
  compose(system_cmps)


%% TOLERANCE ANALYSIS

%% Start with nonk components since we can use tolerates_cags for them

system_tolerates_cct1: THEOREM
  tolerates(singleton(cct), nonk_cmps)

system_tolerates_ptt1: THEOREM
  tolerates(singleton(ptt), nonk_cmps)

system_tolerates_kst1: THEOREM
  tolerates(singleton(kst), nonk_cmps)

system_tolerates_ssupst1: THEOREM
  tolerates(singleton(ssupst), nonk_cmps)

system_tolerates_clientt1: THEOREM
  tolerates(singleton(clientt), nonk_cmps)

system_tolerates_sst1: THEOREM
  tolerates(singleton(sst), nonk_cmps)


%% Now consider the kernel
```

```
%% First show the kernel does the right things for
%% its kst_substates.

kernel_tolerance_help: THEOREM
 LET a1 = ext_st(k(PROJ_1(tran))),
      a2 = ext_st(k(PROJ_2(tran))) IN
    guar(kt)(tran)
      AND PROJ_3(tran) = b
      AND kst_substate(kst1, a1)
      AND kst_substate(kst2, a2)
      AND hidd_base(b, kst1, kst2)
  IMPLIES environment_base(b, kst1, kst2)


system_tolerates_cct2: THEOREM
  tolerates(singleton(cct), singleton(kt))

system_tolerates_ptt2: THEOREM
  tolerates(singleton(ptt), singleton(kt))

system_tolerates_kst2: THEOREM
  tolerates(singleton(kst), singleton(kt))

system_tolerates_ssupst2: THEOREM
  tolerates(singleton(ssupst), singleton(kt))

system_tolerates_clientt2: THEOREM
  tolerates(singleton(clientt), singleton(kt))

system_tolerates_sst2: THEOREM
  tolerates(singleton(sst), singleton(kt))


%% Now use tolerates_union to tie everything together.

system_tolerates_kt: THEOREM
  tolerates(singleton(kt), system_cmps)

system_tolerates_cct: THEOREM
  tolerates(singleton(cct), system_cmps)

system_tolerates_ptt: THEOREM
  tolerates(singleton(ptt), system_cmps)

system_tolerates_kst: THEOREM
  tolerates(singleton(kst), system_cmps)

system_tolerates_ssupst: THEOREM
  tolerates(singleton(ssupst), system_cmps)

system_tolerates_clientt: THEOREM
  tolerates(singleton(clientt), system_cmps)

system_tolerates_sst: THEOREM
  tolerates(singleton(sst), system_cmps)


END system
```

Theory $system\_cmp\_thm$ instantiates the composition theorem, once for each component, to obtain seven theorems that may be used to lift the properties of individual components to properties of the entire system. For example, theorem $system\_cmp\_thm\_k$ states that if the (untranslated) kernel component $k\_comp$ satisfies a property $kp$ and the state and agent translators for the kernel component (i.e., $k2system\_sttran$ and $system\_agtran$) translate $kp$ to a property $p$ on the common state space, then the system satisfies $p$.

## THEORY *system_cmp_thm*

```
system_cmp_thm: THEORY
BEGIN

  IMPORTING system


  kp: VAR prop_t[(K_STATE), THREAD]

  ccp: VAR prop_t[(CC_STATE), THREAD]

  ptp: VAR prop_t[(PT_STATE), THREAD]

  ksp: VAR prop_t[(KS_STATE), THREAD]

  ssupsp: VAR prop_t[(SSUPS_STATE), THREAD]

  clientp: VAR prop_t[(CLIENT_STATE), THREAD]

  ssp: VAR prop_t[(SS_STATE), THREAD]

  p: VAR prop_t[(SYSTEM_STATE), THREAD]

  system_cmp_thm_k: THEOREM
        satisfies(k_comp, kp)
              AND pmap(kp, k2system_sttran, system_agtran) = p
            IMPLIES satisfies(system, p)

  system_cmp_thm_cc: THEOREM
        satisfies(cc_comp, ccp)
              AND pmap(ccp, cc2system_sttran, system_agtran) = p
            IMPLIES satisfies(system, p)

  system_cmp_thm_pt: THEOREM
        satisfies(pt_comp, ptp)
              AND pmap(ptp, pt2system_sttran, system_agtran) = p
            IMPLIES satisfies(system, p)

  system_cmp_thm_ks: THEOREM
        satisfies(ks_comp, ksp)
              AND pmap(ksp, ks2system_sttran, system_agtran) = p
            IMPLIES satisfies(system, p)

  system_cmp_thm_ssups: THEOREM
        satisfies(ssups_comp, ssupsp)
              AND pmap(ssupsp, ssups2system_sttran, system_agtran) = p
            IMPLIES satisfies(system, p)

  system_cmp_thm_client: THEOREM
        satisfies(client_comp, clientp)
              AND pmap(clientp, client2system_sttran, system_agtran) = p
```

```
        IMPLIES satisfies(system, p)

  system_cmp_thm_ss: THEOREM
      satisfies(ss_comp, ssp)
          AND pmap(ssp, ss2system_sttran, system_agtran) = p
        IMPLIES satisfies(system, p)

END system_cmp_thm
```

Theory *system_props* outlines the beginnings of a proof of an "interesting" system property. The property is formalized in *sys_encrypts_correctly* and essentially states that if a client has stored text *cypher* as the cypher text associated with text *clear*, then it is possible for *cypher* to be a correct encryption of *clear* according to the protection family selected by the client. (See *client_props* for declarations of the functions used.) The proof of this property would require much more analysis of the system than we have performed. However, to demonstrate the use of the framework we have shown the following:

- *sys_correct_encryption_pred* is the translation of *correct_encryption_pred* (defined in *client_props*) under the translator *client2system_sttran*.
- The set of *SYSTEM_STATE*s that satisfy *correct_ppd_def* on their *client* field is equal to the translation of *correct_ppd_pred* under *client2system_sttran*.
- *system* satisfies the conditional correctness property. That is, if we consider only system behaviors in which *sys_correct_ppd_pred* is always satisfied then *sys_correct_encryption_pred* is always satisfied.

To complete the proof of *sys_encrypts_correctly_prop* we would have to prove the conjecture *sys_correct_ppd_prop*.

## THEORY *system_props*

```
system_props: THEORY
BEGIN

  IMPORTING system_cmp_thm

  IMPORTING client_props

  IMPORTING ks_props

  IMPORTING more_preds

  IMPORTING tpreds

  st, st1, st2 : VAR (SYSTEM_STATE)

  pf: VAR PROT_FAMILY

  clear, cypher: VAR TEXT

  ri: VAR RECEIVED_INFO

  p : VAR FSEQ[[ENCRYPT_MECH, KEY]]

  seed : VAR SEED
```

```
key_mech : VAR KEY_MECH

th: VAR (client_threads)

t: VAR TEXT

%% This is the top-level desired state predicate.  We could try
%% to prove it by decomposing it into other properties
%% which eventually reduce to things you would prove about
%% a single component (using the composition theorem to
%% lift the result).

sys_correct_encryption_pred : STATE_PRED[(SYSTEM_STATE),THREAD] =
  (LAMBDA st:
     (FORALL pf, clear, cypher:
       have_encrypted_text(client(st), pf, clear, cypher)
       => encrypted_with_pf(pf, clear, cypher)))

sys_correct_encryption_thm: THEOREM
  sys_correct_encryption_pred =
      tmap(client2system_sttran, correct_encryption_pred)

sys_correct_ppd_pred:  STATE_PRED[(SYSTEM_STATE),THREAD] =
  (LAMBDA st: correct_ppd_def(client(st)))

sys_correct_ppd_thm: THEOREM
  sys_correct_ppd_pred =
      tmap(client2system_sttran, correct_ppd_pred)

correct_encryption_prop: THEOREM
  satisfies(system,
                 pimplies(alwayss(sys_correct_ppd_pred),
                          alwayss(sys_correct_encryption_pred)))

sys_correct_ppd_prop: CONJECTURE
  satisfies(system, alwayss(sys_correct_ppd_pred))

sys_encrypts_correctly_prop: THEOREM
  satisfies(system, alwayss(sys_correct_encryption_pred))

END system_props
```

This section has demonstrated how to define a system as a composition of components within the composition framework presented earlier in this report. It has also demonstrated the tolerance analysis that is needed to apply the composition theorem. The theorem has been used to lift a client property to a system property. Thus, all the primary types of analysis required to apply the framework have been demonstrated.

# Section 27 Conclusion

## 27.1 Achievements

This report has described a PVS framework for specifying and analyzing a system as a composition of components. The advantages of this analysis methodology are that it

- reduces reasoning about a system to reasoning about its components,
- allows reuse of assurance evidence, and
- allows “plug-and-play” assurance analysis.

This report has demonstrated the first of these advantages, but we have not yet had an opportunity to demonstrate the second and third. The framework defines a structure for component specifications that includes

- a strong distinction between operations of the component and those of its environment,
- agents (to support security reasoning), and
- fairness conditions.

It also defines an $n$-ary composition operator that returns a component defined as an interleaving of individual component transitions. It has been shown that under appropriate circumstances (i.e., tolerance) this composition operation is equivalent to intersection of behavior sets for the components. An example of non-trivial size has been considered to help explore scalability and practicality questions. The approach is scalable as long as the tolerance proof obligations can be dealt with effectively. The worked example demonstrates that for systems with a structure similar to that of the example, this can be done.

In terms of writing specifications, the framework described here seems quite usable. The operations supported by each component can be specified in a “standard” state-machine manner, and the framework can then be used to combine the individual operations into a component specification.

There are two other contributions made by this study:

- A definition of what it means for a composition operation to be intuitively “right”. This is important since it recognizes that there are a range of possible definitions for composition, all of which allow the Composition Theorem to be proved, but many of which do not correspond to intuition about what it means to compose systems. While the Composition Theorem places an upper bound on the set of behaviors that the composite can perform, “composition is right” places a lower bound on that set of behaviors.
- The identification and analysis of the “*priv* problem” along with a general solution using *hidd*. This has resulted in a general framework that can be used in analyzing and comparing other approaches.

Finally, we have not merely incorporated the work of others as axioms of our composition framework. The framework has been built from basic definitions of concepts such as *component* and *behavior* using a mechanical proof checker (PVS). This increases the confidence in the soundness of the framework.

## 27.2 Comparison to Prior Work

The approach used to accomplish the composition is a hybrid of the approaches advocated by Abadi-Lamport[1] and Shankar[11]. This approach retains the following advantages of the individual approaches:

- Components must be proven to be appropriate for composition before reasoning about the composite.
- The introduction of new, private data structures in other components does not require updates to be made to the environmental assumptions (i.e., *rely*) nor the *guar* of a component. The *hidd* relation of each new component will constrain the manipulation of its state by the existing components and all components subsequently added.
- The framework makes a clear distinction between the initial states, allowed transitions, and allowed environment transitions for each component. In addition, the framework forces the specification of the agents that are permitted to cause each transition. Although this may not be essential for general proofs of system functionality, it is important for an analysis of security properties. We are not only concerned that a transition is correct, but also that it is performed by an agent that is allowed to perform it.
- Most of the reasoning about a composite system can be reduced to reasoning about individual components.

## 27.3 Problems for Further Work

### 27.3.1 Analysis of System Properties

An obvious disadvantage of using the modular specification approach rather than specifying the example as a monolithic entity is that it was necessary to specify how the individual components interacted. The shared components of the state (i.e., the fields of $KERNEL\_SHARED\_STATE$) are used to model the communication protocol between the kernel and the other components. This increases the size of the specifications. The indirectness of component interactions also complicates the analysis of the system. Multiple system transitions (i.e., kernel and security server transitions) occur for each interaction of the components of the Cryptographic Subsystem. Furthermore, we cannot even talk about the port used by two components to communicate without pulling in the kernel specification to resolve names to ports. The information needed in performing proofs is distributed among several components in ways that are not always convenient.

There is of course a trade-off between the accuracy of the model and the amount of effort required to analyze the model. By explicitly modeling the communication between the components, the correspondence between the model and the actual system is more obvious. It is also important to note that the modular specification approach has advantages from a maintenance standpoint. For example, suppose the security server were later replaced by a different security server that satisfied the same properties used in the correctness proof of the overall system.

Then, the analysis of the system could be updated by simply reproving the security server properties. It would not be necessary to reprove properties of the kernel.

We believe this problem can be addressed by specifying the "application-level" components at two levels of abstraction and using refinement analysis to show that the lower level is an implementation of the higher level and therefore satisfies all properties satisfied by the higher level. The high-level specifications would incorporate any properties that arise from the execution of the components on a secured kernel which are necessary for proving the desired properties of the system. The kernel and security server would not be specified as high-level components. Thus, at the high level, the applications police themselves and each other. At the low level the kernel and security server do the policing. We hope to explore this approach in future work.

We also note here that it can be difficult in some cases to separate system properties into component properties. The example in Section 11 demonstrates this. We were unable to find any obvious way to apply the composition theorem in demonstrating the desired system property. Perhaps something could be done with the state information maintained to make this easier.

### 27.3.2 Proof Obligations

There is a potential problem in the framework with the number of proof obligations for the Composition Theorem when composing a large number of components. For $n$ components there are $O(n^2)$ tolerance proof obligations. However, it appears likely that in practice this will not be a problem. First, for particular architectures, we may be able to reduce the complexity due to the structure of interactions between components. In our example, the non-kernel components interact directly with only the kernel. In this case the number of non-trivial obligations is reduced to $O(n)$. For architectures in which the components are more tightly coupled (e.g., all components share and may modify a given region of memory), this reduction in obligations is not so easily obtained. However, it does not seem unreasonable to require this amount of reasoning about such a tightly coupled system. We are essentially trying to prove that each component manipulates the shared memory according to the conventions agreed upon by all the components. If this is not true, the system probably does not work anyway, and we would like our analysis to uncover this flaw. Even in this case, if all components make the same assumptions regarding the manipulation of the shared memory and they all select from the same operations in manipulating that memory, the analysis could largely be reduced to a comparison of the common assumptions and operations.

We should note here that there may be a tradeoff between the tightness of coupling in component specifications and the level of abstraction as discussed in Section 27.3.1. Omitting the kernel mediation most likely increases the coupling of the other specifications. However, depending upon the assumptions made by the high-level components, this might not pose problems.

### 27.3.3 Translators

Finally, we note that the use of translators is rather clumsy. We have found that in almost all cases (in fact, all cases use in this report) the translators are essentially trivial "inverse projection" functions. Nevertheless,

- the translators must be declared,
- we must prove they really are translators, and

- a significant number of proof steps deal with the translators.

Furthermore, when a translator is used as a way to specify a component, the properties of a component must also be translated. For the inverse projection translators, the translation is the obvious one. However, if a translator that is not an inverse projection function is used (perhaps by accident) the properties of the translated component might not be what is expected. On another program, we are experimenting with a way to virtually avoid the need for inverse projection translators.

*Section* 28
# Notes

## 28.1 Acronyms

**AID** Authentication Identifier

**AVC** Access Vector Cache

**CC** Cryptographic Controller

**CMU** Carnegie Mellon University

**DDT** Domain Definition Table

**DTOS** Distributed Trusted Operating System

**IPC** Interprocess Communication

**KS** Key Servers

**MLS** Multi-Level Secure

**OSF** Open Software Foundation

**PT** Protection Tasks

**SID** Security Identifier

**SSUPS** Security Service Usage Policy Server

**TLA** Temporal Logic of Actions

## 28.2 Glossary

**cags** The agents of a component.

**guar** The transitions that a component can perform.

**hidd** A set of transitions specifying constraints on the interface the component provides to other components.

**init** The set of allowed initial states for a component.

**rely** The assumptions of a component about the transitions that its environment will perform.

**sfar** The set of transition classes for which “strong” fairness assumptions are required

**view** A component’s view of a system is the portion of the system state that is observable by the agents of the component.

**wfar** The set of transition classes for which “weak” fairness assumptions are required

*Appendix* *A*
# Bibliography

[1] Martín Abadi and Leslie Lamport. Conjoining specifications. Technical Report 118, Digital Equipment Corporation, Systems Research Center, December 1993.

[2] K. M. Chandy and J. Misra. *Parallel Program Design — A Foundation*. Addison Wesley, 1988.

[3] Judy Crow, Sam Owre, Natarajan Shankar, and Mandayam Srivas. A tutorial introduction to PVS. Presented at WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida, April 1995. Available from the WEB page WWW://www.csl.sri.com/sri-csl-fm.html.

[4] Todd Fine. A framework for composition. In *Proceedings of the Eleventh Annual Conference on Computer Assurance*, pages 199–212, June 1996.

[5] Keith Loepere. *Mach 3 Kernel Interfaces*. Open Software Foundation and Carnegie Mellon University, November 1992.

[6] Keith Loepere. *OSF Mach Kernel Principles*. Open Software Foundation and Carnegie Mellon University, May 1993.

[7] S. Owre, N. Shankar, and J.M. Rushby. *The PVS Specification Language*. Computer Science Laboratory, SRI International, Menlo Park, CA 94025.

[8] R23 crypto subsystem WEB pages.

[9] O. Sami Saydjari, S. Jeffrey Turner, D. Elmo Peele, John F. Farrell, Peter A. Loscocco, William Kutz, and Gregory L. Bock. Synergy: A distributed, microkernel-based security architecture. Technical report, INFOSEC Research and Technology, R231, November 1993.

[10] Secure Computing Corporation. DTOS Kernel Interfaces Document. DTOS CDRL A003, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, January 1997.

[11] N. Shankar. A lazy approach to compositional verification. Technical Report TSL-93-08, SRI International, December 1993.

*Appendix* B
# Additional PVS Theories

The PVS theories *more_set_lemmas* and *disjoint_sets* provide additional simple mathematical definitions and theorems that are not available in the PVS prelude but were deemed useful in performing proofs.

## THEORY *more_set_lemmas*

```
more_set_lemmas [X: TYPE]    : THEORY

BEGIN

 a,b,s,t : VAR setof[X]

 x : VAR X

 nonempty_union : LEMMA
   nonempty?(a) AND nonempty?(b) IMPLIES nonempty?(union(a,b))

 %% This is useful when you are working with the "choose" function
 %%  so that you can set the domain restriction up to be automatically matched.
 emptyset_not_nonempty? : LEMMA
    a = emptyset IFF NOT nonempty?(a)

 emptyset_no_members : LEMMA
    a = emptyset IFF (FORALL (x: X): NOT member(x, a))

 singleton_epsilon: LEMMA
    (EXISTS (x: X): TRUE) => epsilon(singleton(x)) = x

 singleton_not_emptyset: LEMMA
    singleton(x) /= emptyset

 subset_singleton: LEMMA
    a(x) => subset?(singleton(x), a)

END more_set_lemmas
```

## THEORY *disjoint_sets*

```
disjoint_sets [X: TYPE, IDX: TYPE+]    : THEORY

BEGIN

 a,b,s,t : VAR setof[X]

 x: VAR X
```

```
n : VAR nat

i,j : VAR IDX

f: VAR [IDX -> setof[X]]

subsets_disjoint : LEMMA
        subset?(a,s) and subset?(b,t) and disjoint?(s,t)
         IMPLIES disjoint?(a,b)

disjoint?_commutative : LEMMA
        disjoint?(a,b) IMPLIES disjoint?(b,a)

pairwise_disjoint: setof[[IDX -> setof[X]]] =
 {seq: [IDX -> setof[X]] |
   (FORALL i, j, x:
     seq(i)(x) AND seq(j)(x) => i = j)}

pairwise_disjoint_prop: LEMMA
   pairwise_disjoint(f)
        AND f(i)(x) AND f(j)(x)
    => i = j

END disjoint_sets
```

The theories *finite_sequence* and *fseq_functions* define a type *FSEQ* of finite sequences for use in the Crypto Subsystem example.

## THEORY *finite_sequence*

```
finite_sequence[X : NONEMPTY_TYPE] : THEORY
BEGIN

n : VAR nat

FSEQ : TYPE = [# size : nat, elem : [(LAMBDA n: n > 0 and n <= size)->X] #]

null_seq : FSEQ
null_seq_def : AXIOM size(null_seq) = 0

nonemptyfseq(seq : FSEQ) : bool = (size(seq) > 0)

nseq : VAR (nonemptyfseq)
x: VAR  nat

pop(nseq) : FSEQ =
        (# size := size(nseq) - 1,
           elem := (LAMBDA (x: posnat | x <= size(nseq) - 1) :
                           (elem(nseq))(x+1)) #)

tack_on(e : X, s: FSEQ) : FSEQ
  = (# size := size(s) + 1,
        elem := (LAMBDA (n: posnat | n <= size(s) + 1) :
                        IF n <= size(s) THEN elem(s)(n)
                        ELSE e
                        ENDIF) #)

END finite_sequence
```

# THEORY *fseq_functions*

```
fseq_functions[t1,t2: NONEMPTY_TYPE] : THEORY
  BEGIN

  IMPORTING finite_sequence[t1]
  IMPORTING finite_sequence[t2]

  n : VAR nat

  map(f: [t1 -> t2], s: FSEQ[t1]) : FSEQ[t2]
    = (# size := size(s),
         elem := (LAMBDA (n: posnat | n <= size(s)) : f(elem(s)(n))) #)

  END fseq_functions
```