Part Number 83-0902036A002 Rev A
Version Date 25 June 1997

DTOS GENERALIZED SECURITY POLICY SPECIFICATION

CONTRACT NO. MDA904-93-C-4209
CDRL SEQUENCE NO. A019

Prepared for:
Maryland Procurement Office

Prepared by:

Secure Computing Corporation
2675 Long Lake Road
Roseville, Minnesota 55113

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

Date ______________________   Date ______________________

CDRL

DTOS GENERALIZED SECURITY POLICY SPECIFICATION

Secure Computing Corporation

Abstract

This report forms the basis for an analysis of the generality of the security policies that DTOS can support.

Part Number	83-0902036A002 Rev A
Created	25 June 1997
Revised	25 June 1997
Done for	Maryland Procurement Office
Distribution	SCC Internal
CM	`/home/cmt/rev/dtos/docs/genpolicy/RCS/report.vdd,v 1.7 25 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 Generalized Security Policy Specification surveys security policies, provides a framework for their inclusion in a microkernel design, and provides an analysis of three example policies as part of the Distributed Trusted Operating System (DTOS) program, contract MDA904-93-C-4209.

1.2 System Overview

The DTOS design 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 using appropriately developed security servers, the DTOS kernel can support a wide range of policies, including MLS (Multi-Level Security), Identity Based Access Control (IBAC), and type enforcement. A security server that allows all accesses causes the DTOS kernel to behave essentially the same as the CMU Mach 3.0 kernel, which, although uninteresting from a security standpoint, demonstrates the compatibility of DTOS with Mach 3.0. The first security server planned for development is one that enforces a combination of MLS and type enforcement.

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**, lists other documents that are relevant to this report.
- Section 3, **Generalized Security Policy Overview**, provides motivation for investigating the degree to which DTOS supports various security policies.
- Section 4, **Security Policy Survey**, is a survey of security policies that have been proposed in the literature.
- Section 5, **Generic Framework**, is a framework for the possible interactions between an object manager that enforces a policy and a Security Server that makes policy decisions. It is used as the basis for specifying the DTOS microkernel and each example security server.
- Section 6, **Security Policy Lattice**, defines a set of policy characteristics from which a lattice of security policy types is derived. A number of policies are classified in the lattice based upon the characteristics they hold. The DTOS kernel is also classified based upon the characteristics it can support.

- Section 7, **DTOS Microkernel**, defines the DTOS microkernel as an instance of the generic manager framework.
- Section 8, **MLS/TE Policy**, defines an instance of the generic security server framework that implements MLS with Type Enforcement.
- Section 9, **Clark-Wilson Policy**, defines an instance of the generic security server framework that implements the Clark Wilson integrity policy.
- Section 10, **ORCON Policy**, defines an instance of the generic security server framework that implements the ORCON confidentiality policy.
- Section 11, **Conclusion**, summarizes the results of this work, including the benefits of the formalism and the limitations on implementing security policies in DTOS.
- Section 12, **Notes**, contains a discussion of open issues, an acronym list, and a glossary.
- Appendix A, **Bibliography**, gives citations for each referenced document.

Section 2
Applicable Documents

- DTOS Formal Security Policy Model (FSPM) [27, 28].
- DTOS Formal Top Level Specification (FTLS) [29].
- DTOS Composability Study [24].
- DTOS Kernel and Security Server Software Design Document [30].
- The Z Notation: A Reference Manual [32].

Section 3
Generalized Security Policy Overview

DTOS was designed for a wide variety of uses, both military and civilian. Each of these uses has its own security requirements, and hence DTOS must be capable of supporting a wide range of security policies. The Generalized Security Policy Specification investigates how well that objective is met. It also helps identify ways in which the design might be modified to support additional policies, allowing a decision to be made as to whether these additions are important enough to warrant any increase in overhead to enforce them. As this document evolved its purpose was further generalized to study the policy flexibility of not only the DTOS microkernel but of all systems employing an architecture in which an object manager enforces a security policy, and a separate security server makes policy decisions. Thus, this report will hopefully be of interest not only to those familiar with DTOS, but to anyone interested in policy flexibility in an architecture with separation of policy decision and enforcement. The report identifies characteristics of object managers and their interfaces with security servers that limit policy flexibility. This information may be of value to secure operating systems implementors and to those developing policies for secure systems.[1] No particular knowledge of DTOS is required to read this document.

A major goal of the DTOS project is to mitigate some of the identified risks for the successful implementation of a secure system built from a policy-independent object manager and a security server that defines the policy. One such risk is that the design may rule out certain security policies. Of particular concern are systems in which the security policy must change. The risk is that security enforcement may not respond adequately to policy changes. Enforcement might continue to be based on a previous policy, or, even worse, a combination of the old policy and the new policy. This risk includes both the incremental changes present in a dynamic policy as well as a wholesale change of policy. The purposes of this study are

- to explore the degree to which the DTOS architecture mitigates this risk,
- to identify the range of policies it can support, and
- to learn what would be required to support additional policies.

We approach these questions as follows. We first identify a variety of example security policies by performing a survey of security policies from the computer security literature including dynamic policies that change over time. We then construct a framework that models the interaction between an object manager that enforces policy decisions and a security server that makes policy decisions. The DTOS microkernel is specified as an instance of the generic object manager. For each of three selected security policies we specify an instance of the generic security server that implements the decision making necessary to achieve that policy. For each policy a composability analysis [24] is performed to determine whether the combination of the security server with the DTOS microkernel implements the policy. Any weaknesses in the ability of DTOS to support these policies are identified. Since the framework for specifying object managers and security servers is very general, a wide variety of other manager/security server combinations could be specified and analyzed within the framework.

[1] The sections that may be of the most interest outside the context of DTOS are Section 4 *Security Policy Survey*, Section 6 *Security Policy Lattice*, and Section 11 *Conclusion*.

The steps described above demonstrate how to define a security server for various policies and help establish a lower-bound on policy flexibility. To better study the limits of policy flexibility in DTOS, we have developed a lattice of security policies where each node identifies a set of system characteristics required to support policies at that node. A policy is classified at the lattice node that indicates all the characteristics that are required to support it. An object manager is classified at the node indicating all the characteristics it can support. We have classified the DTOS microkernel and a variety of policies in this lattice. Any policy classified at a node that is not dominated by the node at which the DTOS microkernel is classified probably cannot be supported.[2] A policy classified at a dominated node can be supported assuming that it only requires characteristics that have been used in constructing the lattice. This provides a slightly more general approach to the determination of the policy flexibility of the DTOS microkernel and any other manager.

[2] It is difficult to say with certainty that a particular policy *cannot* be supported since there may be many possible ways to implement the policy on the system. One would need to argue that no implementation can exist.

Section 4

Security Policy Survey

A computing system used in a variety of commercial and military environments must have a security policy general enough to handle the needs of each of those environments. We have surveyed many of the policies discussed in the security literature, including those for both military and commercial systems, from which three have been selected for use in testing the generality of our proposed security architecture and determining the range of security policies that can be supported. Policies surveyed include those based on lattices of levels, those defined in terms of some user identity (including roles and groups), those defined in terms of execution environments (including Type Enforcement and capability systems), and those that maintain well-formed transactions (integrity models).

A *security policy* is "the set of laws, rules, and practices that regulate how an organization manages, protects, and distributes sensitive information" [21]. For a computer system, the security policy must define what information is to be protected, the accesses that the various processes in the system are permitted to make to that information, and how these permissions may be modified. The system must also have a protection mechanism that enforces the policy. In this report our primary focus will be on *access control policies* which are stated in terms of the accesses that processes may make to the information on the system.[3]

The accesses permitted by a security policy can generally be described using an *access control matrix* with a row for each principal that requests accesses, and a column for each object to which access is controlled [13]. Each entry in the matrix is the set of the accesses that are permitted from a principal to an object. Policies differ in how a process is mapped to a principal, the objects that are protected, the accesses in the matrix, and how the matrix can be modified. Each row in the matrix is a list of (object, permitted access set) pairs and is called an execution *environment* [15]. Similarly, each column is a list of (principal, permitted access set) pairs and is called an *Access Control List* (ACL) [13]. The set of objects depends on the system and may include such things as memory, files, message buffers, and processes.

4.1 MLS

A MultiLevel Secure (MLS) policy is defined using a lattice of levels [10]. Level a is said to *dominate*, or be greater than, level b if a is higher than or the same as b in the lattice. In many cases, a level consists of two parts: an element of an ordered sequence of sensitivities (i.e. unclassified, confidential, secret, top secret) and a set of compartments identifying the subject matter of the information; however, a level need not have this structure. Each subject is assigned a level representing its level of trust and each object is assigned a level representing the sensitivity of the information that it contains. In the matrix model, each row represents a level from the lattice; the environment of a subject is the row that represents its level. Each access is classified as a *read* and/or a *write*.

The Bell-LaPadula version of MLS policies [3] allows a process to perform a *read* access only if its level dominates that of the object, and a *write* access only if its level is dominated by that of the object. Thus, *read* is included in those matrix entries for which the level of the row

[3] Section 4.9 briefly describes another family of policies, the information flow policies.

dominates the level of the column, and *write* is included in those matrix entries for which the level of the column dominates that of the row. More restrictive policies can be used for special purposes. For example to achieve non-bypassability of a filter, we could allow a *read* access only if the process and the object have the same level.

With an MLS policy, a process's environment can change if its level changes, if the level of some object changes, if an object is created or destroyed, or if the lattice of levels changes. Some policies prohibit changes to the level of processes and objects; however, level changes can possibly be simulated by creating a duplicate process or object at the new level. Other policies allow the level of an object to reflect the current sensitivity of its contents, or to be modified by the security administrator. For example, in a High-Water-Mark policy a high-level process is allowed to write to an object at a lower level, but the level of the object is then raised to that of the writer.

4.2 Integrity

One class of security policies is concerned with data quality, or integrity, rather than controlling its disclosure. Issues related to integrity include: who created or modified the data, what code was used to do so, what is the integrity of the inputs that were used, and how has the data been tested. For example, the integrity of a compiled program depends on the integrity of the source code (how experienced are the programmers, was it formally verified), the integrity of the compiler, and how thoroughly it has been tested. Data from sources known to be reliable have greater integrity than rumors. Integrity policies proposed by Biba [4] and by Clark and Wilson [8] have been especially well studied.

The Biba policy is based on the premise that the integrity of the output from an execution can be no better than the integrity of the inputs and of the code. A Biba integrity policy is equivalent to Bell-LaPadula with high members of the lattice representing poor integrity instead of high sensitivity. Thus, unreliable data with integrity levels high in the lattice cannot be used by a reliable computation with an integrity level low in the lattice. Likewise, an unreliable computation cannot produce reliable data. Related to this is a Low-Water-Mark policy [7] in which the level of an object of lowered when it is written to that of the writer.

Another form of integrity policy was defined by Clark and Wilson [8]. These policies guarantee that a protected object, known as a Constrained Data Item (CDI), can only be modified by a *well-formed transaction*, known as a Transformation Procedure (TP). "The concept of the well-formed transaction is that a user should not manipulate data arbitrarily, but only in constrained ways that preserve or ensure the integrity of the data." [8, p. 186] Each of these TPs corresponds to an access and the set of TPs that may be applied to a particular CDI defines its type. Another goal of some integrity policies is *separation of duty* in which different subparts of an operation are executed by different processes. Separation of duty is equivalent to the n-person policies discussed in [33] with each principal performing a different duty. In Clark-Wilson policies, separation of duty is implemented by allowing each process to invoke only some of the TPs. An example of this is discussed in [20] for the processing of an invoice in a purchasing department. The TPs for this example are:

1. Record the arrival of an invoice.
2. Verify that the goods have been received.
3. Authorize payment.

A 3-person policy would partition the processes into three groups, such as data entry clerks, purchasing officers, and supervisors. Only a data entry clerk could record the arrival, only a purchasing officer could verify receipt, and only a supervisor could authorize payment.

A special form of Clark-Wilson policy is the dynamic assignment of duties to principals [20]. These policies, called dynamic n-person policies, allow a principal to execute at most one of several TPs on a CDI. In the purchasing department example, any of the processes could record the arrival of an invoice. However, if a supervisor did so, it could not then also authorize payment; another supervisor would have to do this.

4.3 Type Enforcement

Another form of access control policy is Type Enforcement [5, 22]. Allowed accesses are specified with a Domain Definition Table (DDT), which is a coarse version of the access control matrix in which objects that have equal access privileges are clustered into groups called *types* and the principals that have equal access privileges are clustered into groups called *domains*. Because of the coarseness of specification, type enforcement is generally not used to control access to particular objects, but rather to constrain the flow of information between groups of objects (the types). A typical example of its use is for a guard between a group of sensitive objects and the rest of the system. The only domain in the DDT that permits reading from the sensitive object type and writing to objects of other types is the one in which the guard executes. Thus, information may only move from sensitive objects to the rest of the system by passing through a guard. Changes to the DDT potentially impact on many objects and therefore are usually reserved for a very few trusted domains.

Type Enforcement may also be used to implement role-based access control and least privilege which are discussed in the next section. Furthermore, although much of the work in these areas has been within the context of discretionary control, Type Enforcement can implement them in mandatory control.

4.4 IBAC

An Identity-Based Access Control (IBAC) policy [34] defines the allowed accesses to an object according to the identity of the individual making the access. Each row in the matrix represents the accesses allowed to processes operating on behalf of some particular individual, and the row is named by that individual's identity. A column in the matrix (i.e., an ACL) associates with each identity a set of allowed accesses for the corresponding object. A process is permitted an access to an object if the requested access is included in the set of allowed accesses associated with the identity of the process by the ACL of the object. All processes that share an identity have the same permissions (they all have the same environment). Changing the access controls can be controlled by placing each ACL in an object (such as a directory) with its own ACL. Another way to control changes is to include the ACL as part of the object and to make changing the ACL one of the controlled accesses.

Variations to the IBAC policies include groups, roles, negative accesses, and delegation [1]. In the simplest case, a group is a set of individuals and accesses are authorized for groups rather than individuals. A process is associated with some collection of the groups that contain its responsible individual, possibly including a group whose only element is that individual, and its set of allowed accesses is the union of the sets of accesses for each of the groups to which it is associated. The accesses allowed to a process can be restricted by not associating it with all of the groups containing the individual. A principal is therefore a set of groups that share a common member (under this definition, each group can be thought of as the principal whose only member is that group). An environment in the matrix for a principal is the union of the environments for each of the principal's member groups. Thus, the set of accesses for a process is the union of the accesses for each of the groups associated with that process. For example, assume that there are groups:

$$\begin{aligned} W &= \{b, c\} \\ X &= \{a, b, c\} \\ Y &= \{a\} \\ Z &= \{a, b\} \end{aligned}$$

Assume that an object's ACL indicates that $\{W\}$ has no permissions, $\{X\}$ has *execute* permission, $\{Y\}$ has *read* permission, and $\{Z\}$ has *write* permission. One process of individual *a* might be mapped to principal $\{X, Y\}$ and therefore has *execute* and *read* permission, while another process of *a* might be further restricted to only *execute* permission by mapping it to principal $\{X\}$. The advantage of using groups to define the IBAC policy is that the policy can be changed just by changing membership in the groups. Thus, *b* might be given *read* permission without changing any ACLs by adding it to group $\{Y\}$. Groups are objects of the system so that these changes can be controlled by the security policy. A more complex case is to allow a group of groups. The allowed accesses for an interior group (one contained in other groups) is the union of the accesses of each group in which it is contained.

When the principal to which a process is mapped contains only some of the groups that contain the individual associated with the process, then the environment of the process might contain only some of the permissions to which the responsible individual is entitled. This preserves the principle of "least privilege" by allowing the process to operate with only those accesses that it needs. Such a principal is referred to as a *role*. Assumption of a role by a process can be controlled by treating roles as system objects with accesses *assume*, *create_process_in*, and *delegate* (see below) [23].

Another variation of IBAC is to allow *negative accesses* to an object by some groups. The set of accesses permitted to a process is the union of the positive accesses for each of its associated groups, minus the union of the negative accesses for those groups. Usually, permissions granted (denied) a group override permissions denied (granted) a containing group (specificity takes precedence over generality).

Delegation is the ability for a process to allow another to act on its behalf. For example, a client on one node of a distributed system that needs to use a service on another node might delegate its permissions for the service to a network server that will make requests on its behalf. Sometimes, the delegated permissions may even be greater than the original ones, as in the case of a virtual memory manager. A client of the memory manager may only have *execute* permission for a file, but the memory manager will need to read the file in order for the client to execute it. This facility is provided using principals '**B for A**', where **B** and **A** are principals. The ability to become this principal is restricted to **B** and requires the permission of **A**. Also, delegated permissions should expire after some time limit, possibly defined when the delegation occurs.

Abadi et al. [1] have created an access control calculus that includes groups, roles, and delegation. It also allows for operations that require permission from multiple principals. The calculus controls commands of the form *A says s*, where **A** is a principal and *s* is a statement, possibly a request for an operation to be performed on an object. The calculus has the following elements:

$A \Rightarrow B$**:** principal **A** is stronger than principal **B** (if **B** is allowed to say *s*, so is **A**).

$A \wedge B$**:** principal **A** and principal **B** together make a statement (this can be generalized to n principals to represent an n-person policy).

$A \mid B$**:** principal **A** quoting principal **B** (**A** says **B** says s).

4.5 ORCON

In an IBAC policy, processes that are able to read information from an object can usually make that information available to other processes at their discretion. This discretionary aspect of an IBAC policy can be eliminated by using an Originator Controlled (ORCON) policy [2]. With an ORCON policy [14, 16], the allowed accesses for an object are used to determine the allowed accesses for any object derived from it (shared memory segments must be treated as objects in this policy). In the simplest version, each process has a Propagated Access Control List (PACL). Whenever a process reads an object, the intersection of the allowed accesses for the object and the process's PACL form a new PACL. The allowed accesses for any object written by a process must be a subset of its PACL. For example, assume that

- a process b reads from two objects m and n,
- only a and b have *read* permission to m, and
- only b and c have *read* permission to n.

The only process that can have *read* permission to any object that b subsequently writes is b itself. Alternatively, a process's PACL or an object's ACL can be replaced by an ACL and a set of pointers to the inherited ACLs, allowing an originating process the ability to change the allowed accesses. The allowed accesses for a process are again computed by finding the intersection of the allowed accesses from the referenced ACLs.

4.6 Capabilities

Instead of representing the access matrix as a set of ACLs (columns of the matrix), a set of environments (rows of the matrix) can be used. Each element in an environment is called a *capability* and consists of an object identifier and a set of accesses (referred to as *rights*) [9, 15]. An access to an object is allowed if the process's environment contains a capability with that object and the desired access. Included in the accesses are controls on the use and transfer of the capability. For example, a capability may be 'use-once', which causes it to be removed from the environment when it is used. Also, a capability may be 'no-transfer', which prohibits it from being copied to another environment, or 'no-copy', which allows it to be moved to a new environment only if it is simultaneously removed from the old environment. Some of these controls may be set when an object is created, for example setting a particular capability for an object to be 'no-copy' so that only one subject may hold the capability at a time. Other controls can be set before one process passes a copy to another process.

Capability systems can be enhanced with *rights amplification*, which provides a means of implementing protected subsystems [9, 15]. For example, assume that a file system process provides support for a virtual memory. To get a program file, the file system process requires that both *open* and *read* permissions be held by the client. Various users can be given capabilities for a file with only *read* permission. The users can only make use of these capabilities indirectly by first passing them to the virtual memory manager, which amplifies the capability to add *open* permission (permission to amplify requires possession of a special capability). Rights amplification also provides for "courier processes" that can be given a capability that only has *courier* permission. This permission does not allow the courier process that holds it to make any accesses to the object. However, when the capability is delivered to a process that is allowed to amplify its rights, the recipient can gain the needed permissions.

4.7 Chinese Wall

A Chinese Wall policy constrains accesses based on the history of previous accesses [6]. It was motivated by the need to restrict insider information, such as using knowledge about one firm's activities in the analysis of a competitor's best course of action. It can also be used for cases in which an aggregate of information is more sensitive than individual pieces [19]. The objects of interest are partitioned into data sets and the data sets are partitioned into conflict of interest classes. For example, all objects containing information about a single firm would form a data set. All data sets associated with petroleum companies would form a conflict of interest class, and all data sets associated with computer firms would form another (a conglomerate spanning several industries would presumably belong to multiple conflict of interest classes). When a process accesses an object, it (and any other process with which it cooperates) loses access to any objects in the same conflict of interest class but different data set as the accessed object. Thus, once a process reads confidential information about Unisys, it cannot read confidential information about IBM, but it could read additional information about Unisys or information about Texaco. Also, a process may not write to an object after it has read from an object in a different data set. Thus, it cannot copy information about IBM into a Texaco file where it can be read by a process that already has accessed information about Unisys. The policy does permit "trusted" processes that may read information from one data set, sanitize it, and write it into another data set.

4.8 Combinations of Policies

The security requirements for many systems require a combination of policies. For example, a military system might require both an MLS and an IBAC policy. A simple combination can be made by taking the intersection of the various policies: an access is permitted by the combined policy if it is permitted by each of the constituent policies. Sometimes, however, the combination is more complex. For example, MLS and type enforcement can be combined such that a *write* is permitted if either it is permitted by both the MLS and type enforcement policies (as in the simple combination), or *trusted write* is permitted by the type enforcement policy (a *trusted write* is only allowed to those processes with the authority to declassify information or to extract nonsensitive from sensitive information) [5, 22]. This is an example of a policy in which permission in one subpolicy overrides lack of permission in another subpolicy.

Another form of combination is to use different policies for different objects. Control of a group of sensitive objects might use a Chinese Wall policy, control of a shared virtual memory might use capabilities, and control of a file system might use an IBAC policy. These policies might interact, such as storing the capabilities for the virtual memory in the file system. Thus, the initial linking between a process and the virtual memory would depend on the identity of the process, but then the accesses would be controlled by capabilities.

4.9 Information Flow

The policies given above have been stated in terms of allowed accesses. An alternative is to state them as noninterference requirements [12]. One group of subjects is *noninterfering* with another group if what the first group does has no effect on what the second group can see. For example, an MLS policy can be expressed such that subjects at one security level must be noninterfering with subjects at another level if the second level does not dominate the first level. Note that if having an effect includes more than being able to write to an object that the other group can read, this MLS policy is stronger than the Bell-LaPadula policy in Section 4.1 and prohibits certain *covert channels* [21]. The advantage to a noninterference formulation is that the policy can be stated formally and a model of the system can be proved to satisfy the policy. Several versions, differing in what "having an effect on what can be seen" formally means, have been given [11, 17, 18]. There seems to be general agreement, however, that the formalization must be *composable* in that if two systems are both noninterfering, then the larger system formed by composing them also is noninterfering [17].

Section 5

Generic Framework

We now consider what sorts of policies can be expressed and enforced within the DTOS architecture. We do this by first presenting a general system framework consisting of a manager and a security server. A manager is the only subject able to directly access some collection of objects that it manages. It receives a sequence of requests from various client subjects to perform actions on its objects and must decide, based on its current state, possibly augmented by policy information received from a security server, whether or not to carry out each request. The security server is responsible for making policy decisions, and the manager is responsible for enforcing those decisions. This section presents a framework for such a system. In Section 7 we will describe the DTOS microkernel as an instance of the generic manager. Later sections will specify security servers for several security policies using as a basis the generic security server described here.

We note here that the framework described in this section is quite general and could be applied not only to the specification of a variety of security servers but also a variety of object managers. To achieve this generality, substantial nondeterminism has been left in the framework. This nondeterminism allows a given manager or security server to be specified by supplying the detailed information required to narrow the operations to those actually allowed by the manager or security server. Section 7 provides an example of how to do this for a manager, and Sections 8–10 provide three examples of how to do this for a security server.

The structure of the generic framework is depicted in Figure 1. The manager receives requests from other subjects, including the Security Server, and it sends security requests to the Security Server. The Security Server sends to the manager responses to the Security Server requests. The generic types M_REQ, SS_REQ and $RESP$ denote the manager requests, Security Server requests and Security Server responses, respectively. The manager and Security Server each have internal data that records their processing state. The Security Server's data includes

- **policy** — a description of the policy governing the Security Server's policy decisions
- **active computations** — a list of requests that the Security Server has received and is still processing

The manager's data includes

- **control policy** — the security requirements that the manager associates with each manager request
- **retained permissions** — the security requirements that are satisfied by responses that the manager has previously received from the Security Server
- **active requests** — a list of requests that the manager has received and is still processing
- **permission status of active requests** — the status of each active request with regard to the checking of the associated security requirements
- **independently deniable requests** — those requests that may be denied by the manager without additional consultation of the Security Server

The interpretation and use of this data is explained in greater detail below.

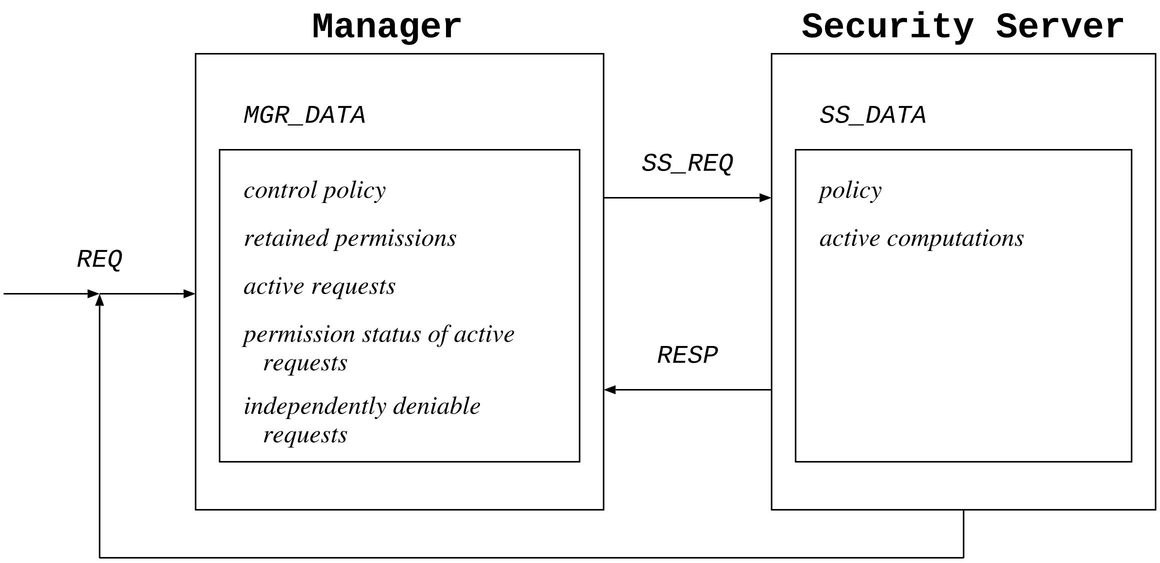

Figure 1: Generic Framework

5.1 Shared State

The bag $pending\_ss\_requests$ denotes the collection of security computation requests that have been sent to the Security Server but not yet received. This component is shared between the manager and the Security Server. The manager can only add elements to the bag while the Security Server can only remove elements. This is formalized in the operation specifications for the manager and Security Server (see Sections 5.2.2 and 5.3.2).

$PendingSsRequests[SS\_REQ]$

$pending\_ss\_requests : \text{bag}\, SS\_REQ$

Similarly, the bag $pending\_responses$ denotes the collection of responses that have been sent by the Security Server to the manager but not yet received.

$PendingResponses[RESP]$

$pending\_responses : \text{bag}\, RESP$

The bag $pending\_requests$ denotes the collection of manager requests that have been sent to the Manager but not yet received. This is shared state information since the Security Server may make manager requests.

$PendingRequests[M\_REQ]$

$pending\_requests : \text{bag}\, M\_REQ$

From a value of type $RESP$, the manager can determine the SS_REQ that prompted the $RESP$ to be sent and also whether the Security Server has indicated that the request is allowed or denied. The operations specified in Sections 5.2.2 and 5.3.2 assume that the generic type SS_REQ coalesces three distinct types of request:

- a permission request — the answer must be "yes" or "no"
- an information request — the answer must be something other than "yes" or "no" (`SSI_context_to_mid` in DTOS is an example),
- a notification — provides information to the security server; no answer is required.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad SsRequestTypes[SS\_REQ] \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
permission\_requests : \mathbb{P}\, SS\_REQ \\
information\_requests : \mathbb{P}\, SS\_REQ \\
notifications : \mathbb{P}\, SS\_REQ \\
\hline
\mathsf{disjoint}\ \langle notifications, permission\_requests, information\_requests \rangle \\
\hline
\end{array}
$$

The first two types of request anticipate a response from the security server. The third does not although it is possible that an acknowledgement might be sent by the Security Server. We allow the possibility that the Security Server might in a single response provide the answers for other requests in addition to the one that is made by the manager. As an example, the DTOS Security Server sends access vectors containing sets of granted and denied permissions to the kernel. The generic type ANS denotes the set of possible answers to SS_REQs. We distinguish two special values of type ANS, ans_yes and ans_no, used to communicate the granting and denial of permissions, respectively. These two special answers may apply only to permission requests. The expression $answers(ss\_response)$ denotes the answers sent in the response $ss\_response$. A pair (ss_req, ans) is in $answers(ss\_response)$ if $ss\_response$ contains an answer ans for security server request ss_req. The functions $grants$, $denies$ and $holds\_information$ partition the answers provided in a response. The expression $grants(ss\_response)$ denotes the set of SS_REQs for which the Security Server provides an answer of ans_yes within response $ss\_response$. The expression $denies(ss\_response)$ denotes the set of SS_REQs for which the Security Server provides an answer of ans_no within response $ss\_response$. The expression $interpret\_response(ss\_response)$ returns the pair (ss_req, ans) containing the request to which the response is a reply and the answer for that request. This pair must be included in $answers(ss\_response)$. The expression $holds\_information(ss\_response)$ denotes the non-permission information that is contained in $ss\_response$. For example, in response to the DTOS Security Server request `SSI_context_to_mid` the Security Server would send a MID (mandatory security identifier) in the $ss\_response$. This MID would be cast as an element of the type ANS. The pair (ss_req, ans) is in $holds\_information(ss\_response)$ exactly when $ss\_response$ contains information ans associated with information request ss_req.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad Answer[ANS] \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
ans\_yes, ans\_no : ANS \\
\hline
ans\_yes \neq ans\_no \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textit{InterpretResponse}[SS\_REQ, RESP, ANS] \\
\hline
\textit{SsRequestTypes}[SS\_REQ] \\
\textit{Answer}[ANS] \\
answers : RESP \longrightarrow (SS\_REQ \mathrel{+\!\!\!\rightarrow} ANS) \\
interpret\_response : RESP \longrightarrow (SS\_REQ \times ANS) \\
grants : RESP \longrightarrow (\mathbb{P}\, SS\_REQ) \\
denies : RESP \longrightarrow (\mathbb{P}\, SS\_REQ) \\
holds\_information : RESP \longrightarrow (SS\_REQ \mathrel{+\!\!\!\rightarrow} ANS) \\
\hline
(\forall\, ss\_response : RESP \\
\bullet\ answers(ss\_response) \neq \varnothing \\
\quad \land\ interpret\_response(ss\_response) \in answers(ss\_response) \\
\quad \land\ grants(ss\_response) \\
\qquad = \{\, ss\_req : SS\_REQ \mid (ss\_req, ans\_yes) \in answers(ss\_response) \,\} \\
\quad \land\ denies(ss\_response) \\
\qquad = \{\, ss\_req : SS\_REQ \mid (ss\_req, ans\_no) \in answers(ss\_response) \,\} \\
\quad \land\ holds\_information(ss\_response) = \{\, ss\_req : SS\_REQ;\ ans : ANS \\
\qquad \mid (ss\_req, ans) \in answers(ss\_response) \land ans \notin \{ans\_yes, ans\_no\} \\
\qquad \bullet\ (ss\_req, ans) \,\} \\
\quad \land\ grants(ss\_response) \subseteq permission\_requests \\
\quad \land\ denies(ss\_response) \subseteq permission\_requests \\
\quad \land\ \mathrm{dom}(holds\_information(ss\_response)) \subseteq information\_requests)
\end{array}
$$

The Security Server may volunteer in a manager request answers to security server requests. (Note that these answers are not necessarily provided in response to any actual security server request.) The expression $volunteers\_answers(req)$ denotes the answers sent in the request req. A pair (ss_req, ans) is in $volunteers\_answers(req)$ if req contains an answer ans for security server request ss_req. The functions $voluntarily\_grants$, $voluntarily\_denies$ and $volunteers\_information$ partition the answers provided in a request. The expression $voluntarily\_grants(req)$ denotes the set of all SS_REQs for which an answer of ans_yes is volunteered in the request req. The expression $voluntarily\_denies(req)$ denotes the set of all SS_REQs for which an answer of ans_no is volunteered in the request req. The expression $volunteers\_information(req)$ denotes the non-permission information that is volunteered in the request req. The pair (ss_req, ans) is in $volunteers\_information(req)$ exactly when req contains information ans associated with information request ss_req. Most managers would execute a request req in which $volunteers\_answers(req) \neq \varnothing$ only if req is received from the security server. However, we do not require that it be impossible for other clients to make such a request.[4]

[4] One example of a manager allowing such a request is a system where clients can make requests to have there own permissions decreased. Such a request constitutes a voluntary denial of permission.

$$
\begin{array}{|l}
\hline
\mathit{InterpretMgrRequest}[M\_REQ, SS\_REQ, ANS] \\
\mathit{SsRequestTypes}[SS\_REQ] \\
\mathit{Answer}[ANS] \\
volunteers\_answers : M\_REQ \longrightarrow (SS\_REQ \nrightarrow ANS) \\
voluntarily\_grants : M\_REQ \longrightarrow \mathbb{P}\, SS\_REQ \\
voluntarily\_denies : M\_REQ \longrightarrow \mathbb{P}\, SS\_REQ \\
volunteers\_information : M\_REQ \longrightarrow (SS\_REQ \nrightarrow ANS) \\
\hline
(\forall\, req : M\_REQ \\
\bullet\ voluntarily\_grants(req) = \{\, ss\_req : SS\_REQ \\
\qquad\qquad \mid (ss\_req, ans\_yes) \in volunteers\_answers(req) \,\} \\
\quad \land\ voluntarily\_denies(req) = \{\, ss\_req : SS\_REQ \\
\qquad\qquad \mid (ss\_req, ans\_no) \in volunteers\_answers(req) \,\} \\
\quad \land\ volunteers\_information(req) = \{\, ss\_req : SS\_REQ;\ ans : ANS \\
\qquad\qquad \mid (ss\_req, ans) \in volunteers\_answers(req) \land ans \notin \{ans\_yes, ans\_no\} \\
\qquad\qquad \bullet\ (ss\_req, ans) \,\} \\
\quad \land\ voluntarily\_grants(req) \subseteq permission\_requests \\
\quad \land\ voluntarily\_denies(req) \subseteq permission\_requests \\
\quad \land\ \mathrm{dom}(volunteers\_information(req)) \subseteq information\_requests) \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\mathit{SharedInterpretation}[M\_REQ, SS\_REQ, RESP, ANS] \\
\quad \mathrel{\widehat{=}} \mathit{InterpretResponse}[SS\_REQ, RESP, ANS] \\
\qquad \land\ \mathit{InterpretMgrRequest}[M\_REQ, SS\_REQ, ANS]
\end{array}
$$

In summary, the state information shared between the manager and the security server consists of all the above.

$$
\begin{array}{|l}
\hline
\mathit{SharedState}[M\_REQ, SS\_REQ, RESP, ANS] \\
\mathit{PendingSsRequests}[SS\_REQ] \\
\mathit{PendingResponses}[RESP] \\
\mathit{PendingRequests}[M\_REQ] \\
\mathit{SharedInterpretation}[M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
\end{array}
$$

5.2 Manager

The manager accepts a set of requests from clients and determines whether each request can be serviced by consulting the Security Server, answers that it has retained from previous consultations with the Security Server, and its own control policy.

5.2.1 State

Each manager is responsible for managing a collection of data. We use the generic type M_DATA to denote the type of the data managed by the manager. The value mgr_data of this type denotes the current contents of the data maintained by the manager.

$$
\begin{array}{|l}
\hline
\mathit{MgrData}[M\_DATA] \\
mgr\_data : M\_DATA \\
\hline
\end{array}
$$

All data maintained by the manager is contained in mgr_data. In specifying pieces of this data we define extraction functions that map mgr_data to the given piece of data. The extraction functions have a name of the form "$extract\_*$" where "*" is the name of the extracted information. For example, the manager assigns a unique request number to each pending request. The expression $active\_request(req\_num)$ denotes the request identified by req_num. The function $active\_request$ is extracted from mgr_data by the mapping $extract\_active\_request$. Generally, we will work with the extracted information in our specification, but it will be seen that each component of the state in the framework is derivable from mgr_data via some extraction function. This formalization allows us to require that a manager maintain certain types of state information and to constrain the ways in which that information is manipulated without specifying precisely how the manager represents the information and performs the manipulation. For example, we will require that the manager have a concept of what security server answers have been *retained*. This information must be contained in mgr_data for the manager. If a given manager does not retain any answers, this may be specified by stating that the appropriate $extract\_*$ function returns an empty set of retained answers for all mgr_data values. For a manager that does retain answers, the $extract\_*$ function may be constrained to model the answers that are retained. Our typical practice will be to constrain the extracted information itself rather than the extraction function. Examples of such definitions can be seen in the section that define the DTOS kernel and the example security servers.

$[REQ\_NUMBER]$

$MgrRequests[M\_DATA, M\_REQ]$
$MgrData[M\_DATA]$
$active\_request : REQ\_NUMBER \nrightarrow M\_REQ$
$extract\_active\_request : M\_DATA \longrightarrow REQ\_NUMBER \nrightarrow M\_REQ$

$active\_request = extract\_active\_request(mgr\_data)$

In general, a manager might need to make a set of computation requests of the Security Server to determine whether a given manager request is permitted. The expression $required(req)$ indicates the set of such computation requests that the manager requires the Security Server to perform before it will accept the request. This set is derived from mgr_policy which gives the set of all SS_REQ (for both permissions and information) that the manager believes are pertinent to the request (i.e., the control policy of the manager) and $retained$ which, for each M_REQ, gives the set of all SS_REQ for which the manager has retained an answer in its state from previous Security Server responses and requests that volunteer answers. Note that we allow the possibility that an answer may be retained with respect to some M_REQ and not with respect to others. This is interesting in the case where a permission can get cached in multiple places in the manager with each place controlling a certain set of requests. The permission might later be flushed from one such cache but remain in others. An example of this in DTOS is the caching of permissions in the protection bits of a page. A manager may also retain for each M_REQ a set of SS_REQs for which a denial of permission has been retained. This is modeled by the function $retained\_denial$.

The functions $retained$ and $retained\_denial$ are both derived from $retained\_answers$. The expression $retained\_answers(req)$ denotes the set of retained answers received in previous responses and requests from the Security Server. These retained answers can be ans_yes, ans_no or any information value of type ANS. The expression $retained(req)$ provides the SS_REQ for which a non-ans_no answer has been retained for request req. The expression $retained\_denial(req)$ provides those for which ans_no has been retained for req. Since $retained$, $retained\_denial$ and

$required$ are derived entirely from $retained\_answers$ and mgr_policy we do not define extraction functions for them.

The relation $retained\_rel$ contains a pair (req, ss_req) if and only if $ss\_req \in retained(req)$. Similarly, $retained\_denial\_rel$ contains a pair (req, ss_req) if and only if $ss\_req \in retained\_denial(req)$.

$$
\begin{array}{|l}
\hline
MgrPolicy[M\_DATA, M\_REQ, SS\_REQ, ANS] \\
MgrData[M\_DATA] \\
SsRequestTypes[SS\_REQ] \\
Answer[ANS] \\
mgr\_policy : M\_REQ \longrightarrow \mathbb{P}\, SS\_REQ \\
extract\_mgr\_policy : M\_DATA \longrightarrow M\_REQ \longrightarrow \mathbb{P}\, SS\_REQ \\
retained\_answers : M\_REQ \longrightarrow (SS\_REQ \nrightarrow ANS) \\
retained\_answers\_rel : M\_REQ \leftrightarrow (SS\_REQ \times ANS) \\
extract\_retained\_answers : M\_DATA \longrightarrow M\_REQ \longrightarrow (SS\_REQ \nrightarrow ANS) \\
retained : M\_REQ \longrightarrow \mathbb{P}\, SS\_REQ \\
retained\_rel : M\_REQ \leftrightarrow SS\_REQ \\
retained\_denial : M\_REQ \longrightarrow \mathbb{P}\, SS\_REQ \\
retained\_denial\_rel : M\_REQ \leftrightarrow SS\_REQ \\
required : M\_REQ \longrightarrow \mathbb{P}\, SS\_REQ \\
\hline
mgr\_policy = extract\_mgr\_policy(mgr\_data) \\
retained\_answers = extract\_retained\_answers(mgr\_data) \\
\\
\operatorname{ran} mgr\_policy \subseteq \mathbb{P}(permission\_requests \cup information\_requests) \\
\operatorname{ran} retained\_denial\_rel \subseteq permission\_requests \\
\operatorname{ran} retained\_rel \subseteq permission\_requests \cup information\_requests \\
\\
\forall\, req : M\_REQ \\
\bullet\ retained(req) = \{ss\_req : SS\_REQ;\ ans : ANS \\
\qquad\qquad \mid (ss\_req, ans) \in retained\_answers(req) \land ans \neq ans\_no \\
\qquad\qquad \bullet\ ss\_req\,\} \\
\quad \land\ retained\_denial(req) \\
\qquad\qquad = \{ss\_req : SS\_REQ \mid (ss\_req, ans\_no) \in retained\_answers(req)\,\} \\
\quad \land\ required(req) = mgr\_policy(req) \setminus retained(req) \\
\quad \land\ retained\_answers\_rel(\!|\{req\}|\!) = retained\_answers(req) \\
\quad \land\ retained\_rel(\!|\{req\}|\!) = retained(req) \\
\quad \land\ retained\_denial\_rel(\!|\{req\}|\!) = retained\_denial(req) \\
\hline
\end{array}
$$

If $required(req)$ is equal to the empty set, then the Manager is able to determine on its own whether the request is permissible. In some cases, the Manager can determine on its own that a request is not permissible even though $required(req) \neq \varnothing$. The set $denied\_requests$ denotes the set of requests that the Manager can reject on its own without further consultation with the Security Server. Note that as the Manager state changes $denied\_requests$ can change. Thus, it is possible that after some amount of processing the Manager would decide on its own that the request should be denied even though it has not received an ans_no response from the Security Server.

$$
\begin{array}{|l}
\multicolumn{1}{l}{MgrDeniedRequests[M\_DATA, M\_REQ]} \\
MgrData[M\_DATA] \\
denied\_requests : \mathbb{P}\, M\_REQ \\
extract\_denied\_requests : M\_DATA \longrightarrow \mathbb{P}\, M\_REQ \\
\hline
denied\_requests = extract\_denied\_requests(mgr\_data)
\end{array}
$$

The expression $sent(req\_num)$ denotes the set of security computation requests that the Manager's records show it has previously sent to the Security Server to determine whether the request indicated by req_num is permissible. The relation $sent\_rel$ contains a pair (req_num, ss_req) if and only if $ss\_req \in sent(req\_num)$.

$$
\begin{array}{|l}
\multicolumn{1}{l}{MgrSent[M\_DATA, SS\_REQ]} \\
MgrData[M\_DATA] \\
sent : REQ\_NUMBER \longrightarrow \mathbb{P}\, SS\_REQ \\
sent\_rel : REQ\_NUMBER \leftrightarrow SS\_REQ \\
extract\_sent : M\_DATA \longrightarrow REQ\_NUMBER \longrightarrow \mathbb{P}\, SS\_REQ \\
\hline
sent = extract\_sent(mgr\_data) \\
\forall\, req\_num : REQ\_NUMBER \\
\bullet\ sent\_rel(\!|\{req\_num\}|\!) = sent(req\_num)
\end{array}
$$

The expression $obtained(req\_num)$ denotes the set of security computation requests for which the Manager's records show the Security Server has responded with an answer other then ans_no for the request indicated by req_num. This may include both permission and information requests. This can be thought of as those computation requests that have "succeeded", where success is interpreted as ans_yes for permission requests and any response for an information request. The relation $obtained\_rel$ contains a pair (req_num, ss_req) if and only if $ss\_req \in obtained(req\_num)$.

$$
\begin{array}{|l}
\multicolumn{1}{l}{MgrObtained[M\_DATA, SS\_REQ]} \\
MgrData[M\_DATA] \\
obtained : REQ\_NUMBER \longrightarrow \mathbb{P}\, SS\_REQ \\
obtained\_rel : REQ\_NUMBER \leftrightarrow SS\_REQ \\
extract\_obtained : M\_DATA \longrightarrow REQ\_NUMBER \longrightarrow \mathbb{P}\, SS\_REQ \\
\hline
obtained = extract\_obtained(mgr\_data) \\
\forall\, req\_num : REQ\_NUMBER \\
\bullet\ obtained\_rel(\!|\{req\_num\}|\!) = obtained(req\_num)
\end{array}
$$

The expression $allowed(req\_num)$ denotes the current status of the security processing performed by the Manager for the request indicated by req_num. The value returned is either $status\_yes$ or $status\_unknown$ depending on whether the request denoted by req_num has been approved or is still being checked. We do not have a value of $status\_no$ since requests for which a permission is denied are terminated rather than being marked unallowed.

$$STATUS ::= status\_yes \mid status\_unknown$$

$$
\begin{array}{|l}
\hline
\textit{MgrAllowed}[M\_DATA] \\
\textit{MgrData}[M\_DATA] \\
allowed : REQ\_NUMBER \rightarrowtail STATUS \\
extract\_allowed : M\_DATA \longrightarrow REQ\_NUMBER \rightarrowtail STATUS \\
\hline
allowed = extract\_allowed(mgr\_data) \\
\hline
\end{array}
$$

The bag *responses* denotes the responses from the Security Server that the Manager currently has buffered. The elements of this bag are denoted by the generic type $RESP$.

$$
\begin{array}{|l}
\hline
\textit{MgrResponses}[M\_DATA, RESP] \\
\textit{MgrData}[M\_DATA] \\
responses : \mathrm{bag}\ RESP \\
extract\_responses : M\_DATA \longrightarrow \mathrm{bag}\ RESP \\
\hline
responses = extract\_responses(mgr\_data) \\
\hline
\end{array}
$$

A Manager contains internally all of the state components described in this section. We also require the following:

- Every active request has an *allowed* status.
- If for some request, R, the manager policy calls for an SS_REQ, S, and a denial of S is currently retained for R, then the manager is *capable* of denying the request independently. This does not mean that the manager will make no Security Server computation requests.

$$
\begin{array}{|l}
\hline
\textit{MgrInternals}[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\textit{MgrData}[M\_DATA] \\
\textit{MgrRequests}[M\_DATA, M\_REQ] \\
\textit{MgrPolicy}[M\_DATA, M\_REQ, SS\_REQ, ANS] \\
\textit{MgrDeniedRequests}[M\_DATA, M\_REQ] \\
\textit{MgrSent}[M\_DATA, SS\_REQ] \\
\textit{MgrAllowed}[M\_DATA] \\
\textit{MgrResponses}[M\_DATA, RESP] \\
\textit{MgrObtained}[M\_DATA, SS\_REQ] \\
\hline
\mathrm{dom}\ allowed = \mathrm{dom}\ active\_request \\
\mathrm{dom}\ sent\_rel \subseteq \mathrm{dom}\ active\_request \\
\mathrm{dom}\ obtained\_rel \subseteq \mathrm{dom}\ active\_request \\
\{req : M\_REQ \mid mgr\_policy(req) \cap retained\_denial(req) \neq \emptyset\} \subseteq denied\_requests \\
\hline
\end{array}
$$

The state of a Manager consists of all of its internal state plus the shared state information.

$$
\begin{array}{|l}
\hline
\textit{MgrState}[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\textit{MgrInternals}[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\textit{SharedState}[M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
\end{array}
$$

An execution step by the Manager may change any of the Manager's state information except for the shared state information from *SharedInterpretation*. We do not allow this to change in order to ensure that when the Manager receives a response or a request from the Security Server it interprets any permissions embedded in the response or request in the same way that the Security Server did when it formulated the response or request in an earlier system step.

$$\begin{array}{|l}
\hline
\textit{MgrStep}[M_DATA, M_REQ, SS_REQ, RESP, ANS] \\
\Delta\ \textit{MgrState}[M_DATA, M_REQ, SS_REQ, RESP, ANS] \\
\Xi\ \textit{SharedInterpretation}[M_REQ, SS_REQ, RESP, ANS] \\
\hline
\end{array}$$

5.2.2 Operations

The possible actions performed by the manager are described below. We note that none of these actions constrains mgr_policy', $extract\_mgr\_policy'$, $denied\_requests'$ or $extract\_denied\_requests'$. As a result, the manager's control policy and the set of requests that it decides on its own to reject (for whatever reason) can change at any time. This reflects the fact that the state of the manager can change during any operation, and the state changes that occur might affect the control policy or the denied requests. For example, in the DTOS kernel almost any processing step could potentially produce a resource shortage that would cause many requests to be added to the set $denied\_requests$. It is assumed that the specification of a particular manager will constrain the changes to mgr_policy and $denied\_requests$ in whatever way is appropriate for that manager. All constraints upon $retained\_answers\_rel'$, $sent\_rel'$ and $obtained\_rel'$ are defined in terms of the subset relation $\subseteq$ rather than equality. A similar property applies to constraints on the bag $responses'$. This means that these sets are allowed to shrink at any time. That is, the manager can freely remove the following from its internal data:

- the retention of any permissions that have been granted or volunteered by the Security Server,
- retained denial of permissions,
- any record of past Security Server requests,
- any record of Security Server affirmative responses, and
- any unprocessed Security Server responses.

These subset constraints imply that, while the manager is allowed to discard its records of past events, it cannot record events that did not happen. For example, a permission may not be retained unless it was actually granted in a response or request.

5.2.2.1 Receive Request

In response to receiving a request $req?$, the manager:

- assigns it an unused request number $req\_num?$,
- records the binding between $req\_num?$ and $req?$ in $active\_request$, and
- sets $allowed(req\_num?)$ to $status\_unknown$.

In summary, the request is added to the set of active requests and the manager data is set to indicate that no security processing has yet been performed on the request.

$$
\begin{array}{|l}
\hline
MgrReceiveRequest[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
req? : M\_REQ \\
req\_num? : REQ\_NUMBER \\
MgrStep[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
req? \sqsubseteq pending\_requests \\
req\_num? \notin \mathrm{dom}\ active\_request \\
\\
pending\_ss\_requests' = pending\_ss\_requests \\
pending\_responses' = pending\_responses \\
pending\_requests' = pending\_requests \uplus [\![req?]\!] \\
active\_request' = active\_request \oplus \{ req\_num? \mapsto req? \} \\
allowed' = allowed \oplus \{ req\_num? \mapsto status\_unknown \} \\
obtained\_rel' \subseteq obtained\_rel \\
sent\_rel' \subseteq sent\_rel \\
responses' \sqsubseteq responses \\
retained\_answers\_rel' \subseteq retained\_answers\_rel \\
\hline
\end{array}
$$

5.2.2.2 Send Request to Security Server A manager may send requests (including permission, information and notification requests) to the Security Server. This always involves placing a ss_req in $pending\_ss\_requests$. We model this common behavior together with other invariants in the schema $MgrSendRequestAux$.

$$
\begin{array}{|l}
\hline
MgrSendRequestAux[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
ss\_req? : SS\_REQ \\
MgrStep[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
pending\_ss\_requests' = pending\_ss\_requests \uplus [\![ss\_req?]\!] \\
pending\_responses' = pending\_responses \\
pending\_requests' = pending\_requests \\
active\_request' = active\_request \\
obtained\_rel' \subseteq obtained\_rel \\
allowed' = allowed \\
responses' \sqsubseteq responses \\
retained\_answers\_rel' \subseteq retained\_answers\_rel \\
\hline
\end{array}
$$

Permission and information requests are modeled by the same transition, $MgrRequestComputation$. If the status of the request indicated by $req\_num?$ is $status\_unknown$ and there is a SS_REQ that

- is in $required$ for this request,
- has not already been sent according to the manager's records, and
- has no denial retained with respect to the manager request,

then the manager can send this SS_REQ computation request to the Security Server.[5] The fact

[5]Note that our framework does not include any liveness (i.e., scheduling) requirements. Furthermore, its nondeterminism allows many possible orderings for its operations. Thus, we cannot really say here that the manager *will* take a particular action such as sending a computation request. We can only say that in particular situations, this transition is allowed. From a standpoint of access control, this is probably sufficient. If we wished to analyze systems for denial of service, we might need to consider stronger specifications. Of course, there is nothing to prevent an instantiation of this framework as the specification of a particular manager from being entirely deterministic. We note that liveness properties have been considered some in the DTOS Composability Study [24].

that it has been sent may be recorded in $sent\_rel$.

$$
\begin{array}{l}
\hline
MgrRequestComputation[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\quad req\_num? : REQ\_NUMBER \\
\quad req? : M\_REQ \\
\quad ss\_req? : SS\_REQ \\
\quad MgrStep[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\quad MgrSendRequestAux[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
\quad (req\_num?, req?) \in active\_request \\
\quad (req\_num?, status\_unknown) \in allowed \\
\quad ss\_req? \in (required(req?) \setminus sent(req\_num?)) \setminus retained\_denial(req?) \\
\quad ss\_req? \in permission\_requests \cup information\_requests \\
\\
\quad sent\_rel' \subseteq sent\_rel \cup \{req\_num? \mapsto ss\_req?\} \\
\hline
\end{array}
$$

Notification requests are modeled by the schema $MgrSendNotification$. We do not record that a notification has been sent since the manager does not require a response.

$$
\begin{array}{l}
\hline
MgrSendNotification[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\quad ss\_req? : SS\_REQ \\
\quad MgrStep[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\quad MgrSendRequestAux[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
\quad ss\_req? \in notifications \\
\\
\quad sent\_rel' \subseteq sent\_rel \\
\hline
\end{array}
$$

5.2.2.3 Receive Response

The manager can receive a response by moving an element of $pending\_responses$ into $responses$.

$$
\begin{array}{l}
\hline
MgrReceiveResponse[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\quad ss\_response? : RESP \\
\quad MgrStep[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
\quad ss\_response? \mathrel{⋿} pending\_responses \\
\\
\quad pending\_responses' = pending\_responses \mathbin{⩁} [\![ss\_response?]\!] \\
\quad responses' \sqsubseteq responses \uplus [\![ss\_response?]\!] \\
\quad pending\_ss\_requests' = pending\_ss\_requests \\
\quad pending\_requests' = pending\_requests \\
\quad active\_request' = active\_request \\
\quad sent\_rel' \subseteq sent\_rel \\
\quad obtained\_rel' \subseteq obtained\_rel \\
\quad allowed' = allowed \\
\quad retained\_answers\_rel' \subseteq retained\_answers\_rel \\
\hline
\end{array}
$$

5.2.2.4 Process Response

When the manager receives a response from the Security Server, it must determine the active request to which the response pertains and the answer. The security computation being checked must be in the set of computations recorded as sent for the request and not in the set of computations recorded as obtained for the request. The $allowed$ status must be $status\_unknown$.

The actions of the manager depend upon whether the response is affirmative (i.e., permission is granted or information provided) or negative (i.e., permission is denied). We first define an auxiliary schema $MgrProcessResponseAux$ that states the behavior that these two cases have in common.

In addition to the requested permission or information, the response might contain other answers. The manager might choose to retain any or all of these. We define the expression $Max\_retain(old\_retained\_rel, ss\_req\_set)$ to be the retained answer set containing all the answers in $old\_retained$ plus the retention of each answer indicated by ans_fun for each request req.

$$
\begin{array}{|l}
[M\_REQ, SS\_REQ, ANS] \\
\hline
Max\_retain : (M\_REQ \leftrightarrow (SS\_REQ \times ANS)) \times (SS\_REQ \rightarrowtail\!\!\!\!\!\!+\; ANS) \\
\quad \longrightarrow (M\_REQ \leftrightarrow (SS\_REQ \times ANS)) \\
\hline
\forall\, old\_retained\_rel : M\_REQ \leftrightarrow (SS\_REQ \times ANS); \\
\quad ans\_fun : (SS\_REQ \rightarrowtail\!\!\!\!\!\!+\; ANS) \\
\bullet\; Max\_retain(old\_retained\_rel, ans\_fun) \\
\quad = old\_retained\_rel \cup (M\_REQ \times ans\_fun) \\
\hline
\end{array}
$$

As part of processing a response, any active request may be terminated and its permission status cleared. No request may obtain the $status\_yes$ value for the $allowed$ function during the processing of a response (see Section 5.2.2.6).

$$
\begin{array}{|l}
MgrProcessResponseAux[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
req\_num? : REQ\_NUMBER \\
req? : M\_REQ \\
ss\_req? : SS\_REQ \\
ss\_response? : RESP \\
MgrStep[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\Xi\, SharedState[M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
(req\_num?, req?) \in active\_request \\
(req\_num?, status\_unknown) \in allowed \\
ss\_req? \in sent(req\_num?) \setminus obtained(req\_num?) \\
ss\_response? \sqsubseteq responses \\
\\
responses' \sqsubseteq responses \uplus [\![ss\_response?]\!] \\
retained\_answers\_rel' \subseteq Max\_retain(retained\_answers\_rel, answers(ss\_response?)) \\
sent\_rel' \subseteq sent\_rel \\
active\_request' \subseteq active\_request \\
\forall\, req\_num : REQ\_NUMBER \\
\bullet\; allowed'(req\_num) = status\_yes \Rightarrow allowed(req\_num) = status\_yes \\
\hline
\end{array}
$$

If the answer is ans_no and there is a request for which the computation has been sent but not yet obtained, then the request can be terminated. When a request is terminated, it is removed from the domain of $active\_request$ (and hence from the domain of $allowed$). The permission status of the request is also cleared. We allow the possibility that additional requests might be terminated (and their permission status cleared) in the same transition. This makes it possible for the manager to immediately apply the answers in a response to a variety of requests. Since several manager state transitions result in the termination of a request we specify an auxiliary schema $MgrTerminateRequestAux$ describing this transition.

$$
\begin{array}{l}
\textit{MgrTerminateRequestAux}[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
MgrStep[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
req\_num? : REQ\_NUMBER \\
\hline
active\_request' \subseteq \{ req\_num? \} \lhd active\_request \\
sent\_rel' \subseteq sent\_rel \\
obtained\_rel' \subseteq obtained\_rel \\
\forall req\_num : REQ\_NUMBER \\
\bullet\ allowed'(req\_num) = status\_yes \Rightarrow allowed(req\_num) = status\_yes
\end{array}
$$

$$
\begin{array}{l}
\textit{MgrNegativeResponse}[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
MgrProcessResponseAux[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
MgrTerminateRequestAux[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
(ss\_response?, (ss\_req?, ans\_no)) \in interpret\_response
\end{array}
$$

If the answer in a response is not ans_no and there is a request for which the computation has been sent but not yet obtained, then the manager can record that a non-negative answer has been obtained for the computation for that request. In the case of an information request, the information will be taken into account (and might be retained), but for the purpose of tracking the security processing of the request we need only record that the information has been obtained.

$$
\begin{array}{l}
\textit{MgrAffirmativeResponse}[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
MgrProcessResponseAux[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
(\exists\, ans : ANS \\
\mid ans \neq ans\_no \\
\bullet\ (ss\_response?, (ss\_req?, ans)) \in interpret\_response) \\
\\
obtained\_rel' \subseteq obtained\_rel \cup \{ req\_num? \mapsto ss\_req? \}
\end{array}
$$

5.2.2.5 Deny Request If the manager determines that a pending request should not be permitted, then the request can be terminated.

$$
\begin{array}{l}
\textit{MgrDenyRequest}[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
req\_num? : REQ\_NUMBER \\
req? : M\_REQ \\
MgrStep[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\Xi\, SharedState[M\_REQ, SS\_REQ, RESP, ANS] \\
MgrTerminateRequestAux[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
(req\_num?, req?) \in active\_request \\
req? \in denied\_requests \\
\\
responses' \sqsubseteq responses \\
retained\_answers\_rel' \subseteq retained\_answers\_rel
\end{array}
$$

5.2.2.6 Accept Request

If all of the security computations in $required(req\_num?)$ are also in $obtained(req\_num?)$, then the Security Server has approved all of the computations and supplied any necessary information. In this case $allowed(req\_num?)$ can be set to $status\_yes$.

$$
\begin{array}{|l}
\hline
MgrAcceptRequest[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
req\_num? : REQ\_NUMBER \\
req? : M\_REQ \\
MgrStep[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\Xi\, SharedState[M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
(req\_num?, req?) \in active\_request \\
(req\_num?, status\_unknown) \in allowed \\
required(req?) \subseteq obtained(req\_num?) \\
\\
allowed' = allowed \oplus \{ req\_num? \mapsto status\_yes \} \\
active\_request' = active\_request \\
sent\_rel' \subseteq sent\_rel \\
obtained\_rel' \subseteq obtained\_rel \\
responses' \sqsubseteq responses \\
retained\_answers\_rel' \subseteq retained\_answers\_rel \\
\hline
\end{array}
$$

5.2.2.7 Process Request

For the most part, the manager can process any request that has been approved and may change its state during the processing as it sees fit. The only restrictions are:

- The request identified by $req\_num?$ can only be processed if $allowed(req\_num?)$ is $status\_yes$.
- The request $req?$ may not be in $denied\_requests$.
- The manager may only retain answers that are already retained or are volunteered in the request.[6]

After the request is processed, it is terminated. Thus, the operation $MgrProcessRequest$ denotes the completion of a request in a single transition after all security processing for the request has been completed. We also allow the following to happen:

- additional requests may be terminated,
- permission status information may be discarded,
- retained answers may be discarded,
- requests that are marked $status\_yes$ may be marked $status\_unknown$.

This allows cases where the processing of a request from the Security Server causes an immediate withdrawal of permissions that have been previously granted (whether or not they were retained) and this affects the permission status of other active requests.

[6] Normally, answers will only be volunteered in requests made by the Security Server.

$$
\begin{array}{|l}
\multicolumn{1}{l}{MgrProcessRequest[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS]} \\
req\_num? : REQ\_NUMBER \\
req? : M\_REQ \\
MgrStep[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\Xi\, SharedState[M\_REQ, SS\_REQ, RESP, ANS] \\
MgrTerminateRequestAux[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
(req\_num?, req?) \in active\_request \\
(req\_num?, status\_yes) \in allowed \\
req? \notin denied\_requests \\
\\
retained\_answers\_rel' \\
\quad \subseteq Max\_retain(retained\_answers\_rel, volunteers\_answers(req?)) \\
responses' \sqsubseteq responses \\
\hline
\end{array}
$$

5.2.3 Manager as a Component

Following the concepts in the DTOS Composability Study [24], we define a manager as a type of component. This means defining the following four fields for the component:

guar a set of transitions that the manager may perform,

rely a set of transitions that the manager can tolerate from other system components

init a set of allowed initial states for the manager, and

view an equivalence relation denoting the state information that is visible to the manager.

In defining this information and reasoning with it, we will be slightly less formal than in the Composability Study, but the composability framework is the same. We will leave unspecified the agents of each component. In all cases it should be assumed that there is a nonempty set of agents for each of the manager and security server and these two sets are disjoint. We note here that we are following Draft 2 of the Composability Study[7] with the exception that we ignore the *priv* field of a component since, although it was carried over from Draft 1 of the Composability Study, it really serves no useful purpose in Draft 2. When we define a particular instance of a generic manager, we will further constrain the component's fields.

[7] The differences between Draft 2 and the final draft (number 3) are not very important for the work done here. The final draft extends the *compose* operation to compose an arbitrary number of components as opposed to the pairwise operation in Draft 2. We never compose more than two components, a manager and a security server, in this report. The final draft also uses a different method of indicating what parts of the system state are left alone by each component when the components are composed. Both Draft 2 and this report use *respect relations* which are supplied as arguments of the compose operation. The final draft achieves the same end by defining an additional piece of information *hidd* for each system component. It can be shown that the *hidd* approach is a special case of the respect relation approach. The advantage of the *hidd* approach is that it is somewhat easier to manage in situations where lots of components are being analyzed and composed in different combinations. Again, this is not a concern in this report.

The **guar** for a manager allows any of the transitions described in Section 5.2.2. It is modeled by mgr_guar. We assume there is a single manager agent that can perform these transitions.

$$
\begin{array}{l}
MgrGuarStep[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\widehat{=}\ MgrReceiveRequest[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\quad \lor\ MgrRequestComputation[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\quad \lor\ MgrSendNotification[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\quad \lor\ MgrReceiveResponse[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\quad \lor\ MgrNegativeResponse[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\quad \lor\ MgrAffirmativeResponse[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\quad \lor\ MgrDenyRequest[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\quad \lor\ MgrAcceptRequest[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\quad \lor\ MgrProcessRequest[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS]
\end{array}
$$

$$
\begin{array}{|l}
\hline
MgrGuar[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\quad mgr\_guar : \mathbb{P}\ MgrGuarStep[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
\quad mgr\_guar = MgrGuarStep[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
\end{array}
$$

A manager makes the following assumptions about transitions performed by other components of the system:

- No manager-internal data is modified.
- The data in $SharedInterpretation$ is not modified.
- Nothing is removed from $pending\_responses$ and $pending\_requests$.

This is modeled by mgr_rely.

Editorial Note:
We probably should have assumptions about who can add things to the pending lists.

$$
\begin{array}{|l}
\hline
MgrRely[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\quad mgr\_rely : \mathbb{P}\ \Delta\ MgrState[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
\quad mgr\_rely = \{\ \Delta\ MgrState[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\qquad \mid\ \Xi\ MgrInternals[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\qquad \land\ \Xi\ SharedInterpretation[M\_REQ, SS\_REQ, RESP, ANS] \\
\qquad \land\ pending\_responses \sqsubseteq pending\_responses' \\
\qquad \land\ pending\_requests \sqsubseteq pending\_requests'\ \} \\
\hline
\end{array}
$$

The set of allowed initial states is modeled by mgr_init. We require the following sets to be empty: $pending\_responses$, $pending\_requests$, $active\_request$, $sent$, $obtained$, $allowed$ and $responses$. Note that we allow the possibility that $pending\_ss\_requests$ might be non-empty at system start-up. This means that the start-up process is allowed to automatically queue up one or more Security Server requests as a way of getting security information from the Security Server to the manager.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\ MgrInit[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS]\ \_\_\_\_\_\_\_\_} \\
mgr\_init : \mathbb{P}\ MgrState[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
\forall st : mgr\_init \\
\bullet\ st.pending\_responses = \varnothing \\
\quad \wedge\ st.pending\_requests = \varnothing \\
\quad \wedge\ st.active\_request = \varnothing \\
\quad \wedge\ st.sent = \varnothing \\
\quad \wedge\ st.obtained = \varnothing \\
\quad \wedge\ st.allowed = \varnothing \\
\quad \wedge\ st.responses = \varnothing \\
\hline
\end{array}
$$

All information in $MgrState$ is visible to the manager.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\ MgrView[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS]\ \_\_\_\_\_\_\_\_} \\
mgr\_view : MgrState[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\quad \leftrightarrow MgrState[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
\forall st_1, st_2 : MgrState[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\bullet\ (st_1, st_2) \in mgr\_view \Leftrightarrow st_1 = st_2 \\
\hline
\end{array}
$$

$$
\begin{array}{l}
MgrComponent[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\quad \mathrel{\widehat{=}} MgrGuar[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\qquad \wedge\ MgrRely[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\qquad \wedge\ MgrInit[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\qquad \wedge\ MgrView[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS]
\end{array}
$$

5.2.4 Properties

We can prove[8] some useful properties based entirely upon the specification of a generic manager. These properties can then be used as lemmas when analyzing a particular manager.

Lemma 5.1 *A request $req?$ with request number $req\_num?$ can be processed in some state only if in some preceding state*

$$mgr\_policy(req?) \setminus retained(req?) \subseteq obtained(req\_num?).$$

Proof: $MgrProcessRequest$ has the precondition $(req\_num?, status\_yes) \in allowed$. The relation $allowed$ is empty in the initial state, so we examine the manager transitions to determine which ones can make the precondition true. The following table indicates the change to the value of $allowed$ for each of the manager transitions:

$MgrReceiveRequest$	$allowed' = allowed \oplus \{ req\_num? \mapsto status\_unknown \}$
$MgrRequestComputation$ $MgrSendNotification$ $MgrReceiveResponse$	$allowed' = allowed$
$MgrNegativeResponse$ $MgrAffirmativeResponse$ $MgrDenyRequest$ $MgrProcessRequest$	$\forall req\_num : REQ\_NUMBER$ $\bullet\ allowed'(req\_num) = status\_yes$ $\quad \Rightarrow allowed(req\_num) = status\_yes$
$MgrAcceptRequest$	$allowed' = allowed \oplus \{ req\_num? \mapsto status\_yes \}$

[8]The proofs in this and later sections are rather informal. For many of the stated lemmas and theorems, a formal proof would require induction on the length of a prefix of a system trace.

This analysis indicates that the only transition that can add $(req\_num?, status\_yes)$ to $allowed$ is $MgrAcceptRequest$ which requires $required(req?) \subseteq obtained(req\_num?)$. Applying the definition of $required$ completes the proof. □

Lemma 5.2 *A Security Server request $ss\_req?$ is in the value of $obtained(req\_num?)$ in some state only if in some preceding state there is an $ss\_response? \sqsubseteq responses$ such that $(ss\_response?, (ss\_req?, ans)) \in interpret\_response$ where ans is not ans_no.*

Proof: The relation $obtained\_rel$ is initially empty. Only $MgrAffirmativeResponse$ can add an element to the value of $obtained(req\_num?)$, and it does so only under the specified conditions. □

Lemma 5.3 *The bag $responses$ contains an element $ss\_response?$ in some state only if in some preceding state $ss\_response? \sqsubseteq pending\_responses$.*

Proof: The bag $responses$ is initially empty. Only $MgrReceiveResponse$ can add an element to $responses$, and it does so only if the element is in $pending\_responses$. □

Lemma 5.4 *A Security Server request $ss\_req?$ is in the value of $obtained(req\_num?)$ in some state only if in some preceding state there is an $ss\_response? \sqsubseteq pending\_responses$ such that $(ss\_response?, (ss\_req?, ans)) \in interpret\_response$ where ans is not ans_no.*

Proof: This follows from the preceding two lemmas together with the requirements in $MgrStep$ and $MgrRely$ that $interpret\_response$ and ans_no are invariant in all transitions. □

Lemma 5.5 *A Security Server request $ss\_req?$ is in the value of $retained(req?)$ in some state only if it is in $retained(req?)$ in the initial state or there is some preceding state in which*

1. *$ss\_response? \sqsubseteq pending\_responses$ where $(ss\_req?, ans) \in answers(ss\_response?)$, or*
2. *$(req\_num_1, req_1) \in active\_request$, $(req\_num_1, status\_yes) \in allowed$, $req_1 \notin denied\_requests$ and $(ss\_req?, ans) \in volunteers\_answers(req_1)$*

where ans is not ans_no.

Proof: Only $MgrProcessResponseAux$ and $MgrProcessRequest$ may add a pair, (ss_req, ans), to the value of $retained(req?)$. The former (applying Lemma 5.3 and the invariance of $answers$ and ans_no) requires the conditions in (1). The latter requires the conditions in (2). □

The following theorem states that if a request is performed, then every SS_REQ required, at some earlier time, for that request was either retained in the initial state, has been received in a response from the security server or has been volunteered in a processed manager request.

Theorem 5.6 *A request $req?$ with request number $req\_num?$ can be processed in some state only if there is some preceding state s in which, for every $r \in mgr\_policy(req?)$ in s, either*

1. *$r \in retained(req?)$ in the initial state, or*
2. *there is some state s_1 preceding s and some $ans \neq ans\_no$ such that in s_1*
   (a) *$(r, ans) \in answers(ss\_response?)$ for some $ss\_response? \sqsubseteq pending\_responses$, or*

(b) for some request req_1 *and request number* req_num_1,

$$(r, ans) \in volunteers\_answers(req_1),$$
$$req_1 \notin denied\_requests,$$
$$(req\_num_1, req_1) \in active\_request, \text{ and}$$
$$(req\_num_1, status\_yes) \in allowed.$$

Proof: We first note that

$$\forall\, ss\_response : interpret\_response(ss\_response) \in answers(ss\_response).$$

The desired result then follows from Lemmas 5.1, 5.4 and 5.5. Note that the hypotheses of Lemma 5.4 are subsumed by those of 5.5. □

The following corollary states a sufficient condition for a manager to be obeying the mgr_policy in effect at the time when a request is executed rather than the one that was in effect when the permission checking for the request was concluded.

Corollary 5.7 *If* mgr_policy *is monotone non-increasing, that is, in all transitions*

$$\forall\, req : mgr\_policy'(req) \subseteq mgr\_policy(req),$$

then Theorem 5.6 also applies with s *taken to be the state in which* $req?$ *is executed.*

5.3 Security Server

5.3.1 State

The Security Server is responsible for servicing requests for computation that are received from clients. The generic types SS_REQ and $RESP$ are used to denote, respectively, requests for computations and responses to computations. The generic type SS_DATA is used to denote states of the Security Server. The value ss_data denotes the current state of the Security Server.

$$\begin{array}{|l}
\hline
SsData[SS_DATA] \\
\hline
ss_data : SS_DATA \\
\hline
\end{array}$$

The set $policy\_allows$ denotes those security computations returning ans_yes under the policy currently implemented by the Security Server. The expression $ss\_information(ss\_req)$ denotes the information to be returned for an information request ss_req.

$$\begin{array}{|l}
\hline
SsPolicyAllows[SS_DATA, SS_REQ, ANS] \\
\hline
SsData[SS_DATA] \\
policy_allows : \mathbb{P}\, SS_REQ \\
extract_policy_allows : SS_DATA \longrightarrow \mathbb{P}\, SS_REQ \\
ss_information : SS_REQ \twoheadrightarrow ANS \\
extract_ss_information : SS_DATA \longrightarrow (SS_REQ \twoheadrightarrow ANS) \\
\hline
policy_allows = extract_policy_allows(ss_data) \\
ss_information = extract_ss_information(ss_data) \\
\hline
\end{array}$$

Upon receiving a security computation request, the Security Server assigns a unique identifier. The expression $active\_computations(comp\_num)$ denotes the security computation identified by $comp\_num$.

$[COMP\_NUMBER]$

$$
\begin{array}{|l}
\hline
SsActiveComputations[SS\_DATA, SS\_REQ] \\
SsData[SS\_DATA] \\
active\_computations : COMP\_NUMBER \pfun SS\_REQ \\
extract\_active\_computations : SS\_DATA \longrightarrow COMP\_NUMBER \pfun SS\_REQ \\
\hline
active\_computations = extract\_active\_computations(ss\_data) \\
\hline
\end{array}
$$

A Security Server contains internally all of the state components described in this section.

$$
\begin{array}{|l}
\hline
SsInternals[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS] \\
SsData[SS\_DATA] \\
SsPolicyAllows[SS\_DATA, SS\_REQ, ANS] \\
SsActiveComputations[SS\_DATA, SS\_REQ] \\
\hline
\end{array}
$$

The state of a Security Server consists of all of its internal state plus the shared state information. We require the set of SS_REQs allowed by the policy to be a subset of $permission\_requests$. The function $ss\_information$ must be defined for every element of $information\_requests$, but we do allow the possibility that ANS might contain a special value indicating “no information”.

$$
\begin{array}{|l}
\hline
SsState[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS] \\
SsInternals[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS] \\
SharedState[M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
policy\_allows \subseteq permission\_requests \\
\mathrm{dom}\, ss\_information = information\_requests \\
\hline
\end{array}
$$

An execution step by the Security Server may change any of the Security Server’s state information except for the shared state information from $SharedInterpretation$. We do not allow this to change in order to ensure that when the Manager receives a response or a request from the Security Server it interprets any permissions embedded in the response or request in the same way that the Security Server did when it formulated the response or request in an earlier system step.

$$
\begin{array}{|l}
\hline
SsStep[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS] \\
\Delta\, SsState[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS] \\
\Xi\, SharedInterpretation[M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
\end{array}
$$

5.3.2 Operations

The Security Server can perform at least the operations described below. Note that none of these operations constrains $policy\_allows'$, or $extract\_policy\_allows'$. That means the policy is allowed to change during any Security Server operation. The specification for a particular Security Server must state any constraints upon change of the policy.

5.3.2.1 Receive Computation Request In response to receiving a security computation request $ss\_req?$, the Security Server:

- assigns it an unused computation number $comp\_num?$, and
- records the binding between $comp\_num?$ and $ss\_req?$ in $active\_computations$.

$$
\begin{array}{l}
SsReceiveRequest[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS] \\
\hline
comp\_num? : COMP\_NUMBER \\
ss\_req? : SS\_REQ \\
SsStep[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS] \\
\hline
ss\_req? \sqsubseteq pending\_ss\_requests \\
comp\_num? \notin \operatorname{dom} active\_computations \\
\\
pending\_ss\_requests' = pending\_ss\_requests \uminus [\![ss\_req?]\!] \\
active\_computations' = active\_computations \oplus \{ comp\_num? \mapsto ss\_req? \} \\
pending\_responses' = pending\_responses \\
pending\_requests' = pending\_requests
\end{array}
$$

5.3.2.2 Send "No" Response

The Security Server checks $ss\_req?$ against $policy\_allows$ to see if it is allowed. If not, it creates a response containing ans_no, and sends it to the Manager. Note that additional permissions may be sent in the response but only if they are allowed by the policy. Also, information may be sent, but only if it is consistent with $ss\_information$. We do not constrain the expression $denies(ss\_response?)$ since it might be desirable to design a Security Server that sometimes denies permissions that are grantable. For example, a Security Server might only grant permissions that are directly requested in $ss\_req?$, forcing the manager to explicitly request the permissions it needs.

We note that, because of the constraints stated in $InterpretResponse$, the value of ans in $SsSendResponseAux$ constrains the type of the request $ss\_req?$. If $ans \in \{ans\_no, ans\_yes\}$, then $ss\_req? \in permission\_requests$. Otherwise, $ss\_req? \in information\_requests$.

$$
\begin{array}{l}
SsSendResponseAux[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS] \\
\hline
comp\_num? : COMP\_NUMBER \\
ss\_req? : SS\_REQ \\
ss\_response? : RESP \\
ans : ANS \\
SsStep[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS] \\
\hline
(comp\_num?, ss\_req?) \in active\_computations \\
(ss\_response?, (ss\_req?, ans)) \in interpret\_response \\
grants(ss\_response?) \subseteq policy\_allows \\
holds\_information(ss\_response?) \subseteq ss\_information \\
ss\_req? \in permission\_requests \cup information\_requests \\
\\
active\_computations' = \{ comp\_num? \} \ntriangleleft active\_computations \\
pending\_responses' = pending\_responses \uplus [\![ss\_response?]\!] \\
pending\_ss\_requests' = pending\_ss\_requests \\
pending\_requests' = pending\_requests
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{SsSendNegativeResponse}[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS] \\
comp\_num? : COMP\_NUMBER \\
ss\_req? : SS\_REQ \\
ss\_response? : RESP \\
SsStep[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS] \\
\hline
ss\_req? \in permission\_requests \setminus policy\_allows \\
(\mathbf{let}\ ans == ans\_no \\
\bullet\ SsSendResponseAux[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS]) \\
\hline
\end{array}
$$

5.3.2.3 Send "Yes" Response

If $ss\_req?$ is allowed by $policy\_allows$ or if it is an information request, a response containing ans_yes or the requested information is sent to the Manager. Additional permissions and information may be sent but only when consistent with the Security Server state.

$$
\begin{array}{|l}
\hline
\textit{SsSendAffirmativeResponse}[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS] \\
comp\_num? : COMP\_NUMBER \\
ss\_req? : SS\_REQ \\
ss\_response? : RESP \\
SsStep[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS] \\
\hline
ss\_req? \in policy\_allows \cup information\_requests \\
(\exists\, ans : ANS \\
\mid ans \neq ans\_no \\
\bullet\ SsSendResponseAux[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS]) \\
\hline
\end{array}
$$

5.3.2.4 Make Manager Request

The Security Server may make requests of the manager. The request may contain volunteered permissions as long as those permissions are consistent with the policy. As with extra denials sent in responses, we place no constraints on volunteered denials.

$$
\begin{array}{|l}
\hline
\textit{SsMgrRequest}[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS] \\
req? : M\_REQ \\
SsStep[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS] \\
\hline
voluntarily\_grants(req?) \subseteq policy\_allows \\
volunteers\_information(req?) \subseteq ss\_information \\
\\
active\_computations' = active\_computations \\
pending\_responses' = pending\_responses \\
pending\_ss\_requests' = pending\_ss\_requests \\
pending\_requests' = pending\_requests \uplus [\![req?]\!] \\
\hline
\end{array}
$$

5.3.2.5 Change State

The Security Server may change state without communicating with the Manager in any way. This allows the possibility of changes to the policy in response to factors such as the time of day. We require ss_data to change so that we can later distinguish instances of $SsInternalTransition$ from those of $MgrProcessRequest$ (see Section 5.5). We note that this transition is extremely flexible and can model many types of events for a Security Server in an implemented system including

- an intermediate step in Security Server processing,
- a change to the internal tables defining the policy and
- a switch to an entirely new Security Server (perhaps saving the state of the old Security Server so that the system may switch back later).

$$
\begin{array}{|l}
\hline
\textit{SsInternalTransition}[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS] \\
\textit{SsStep}[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS] \\
\Xi\ \textit{SharedState}[M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
ss\_data' \neq ss\_data \\
active\_computations' \subseteq active\_computations \\
\hline
\end{array}
$$

5.3.3 Security Server as a Component

As with the generic manager, we define a generic security server as a type of component.

The **guar** for a security server allows any of the transitions described in Section 5.3.2. It is modeled by ss_guar. We assume there is a single security server agent that can perform these transitions.

$$
\begin{aligned}
& \textit{SsGuarStep}[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS] \\
& \widehat{=}\ \textit{SsReceiveRequest}[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS] \\
& \qquad \vee\ \textit{SsSendNegativeResponse}[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS] \\
& \qquad \vee\ \textit{SsSendAffirmativeResponse}[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS] \\
& \qquad \vee\ \textit{SsMgrRequest}[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS] \\
& \qquad \vee\ \textit{SsInternalTransition}[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS]
\end{aligned}
$$

$$
\begin{array}{|l}
\hline
\textit{SsGuar}[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS] \\
ss\_guar : \mathbb{P}\ \textit{SsGuarStep}[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS] \\
\hline
ss\_guar = \textit{SsGuarStep}[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS] \\
\hline
\end{array}
$$

A security server makes the following assumptions about transitions performed by other components of the system:

- No security server internal data is modified.
- The data in *SharedInterpretation* is not modified.
- No requests are removed from $pending\_ss\_requests$.
- Nothing is added to $pending\_responses$.

This is modeled by ss_rely.

Editorial Note:
We probably also need assumptions about *who* can add and delete things from the pending lists.

$$
\begin{array}{l}
\textbf{SsRely}[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS] \\
\hline
ss\_rely : \mathbb{P}\,\Delta\, SsState[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS] \\
\hline
ss\_rely = \{\, \Delta\, SsState[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS] \\
\quad \mid \Xi\, SsInternals[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS] \\
\quad \land \Xi\, SharedInterpretation[M\_REQ, SS\_REQ, RESP, ANS] \\
\quad \land pending\_ss\_requests \sqsubseteq pending\_ss\_requests' \\
\quad \land pending\_responses' \sqsubseteq pending\_responses \,\}
\end{array}
$$

The set of allowed initial states is modeled by ss_init. We require only that $pending\_responses$, $pending\_requests$ and $active\_computations$ be empty.

$$
\begin{array}{l}
\textbf{SsInit}[SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
ss\_init : \mathbb{P}\, SsState[SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
\forall\, st : ss\_init \\
\bullet\; st.pending\_responses = \varnothing \\
\quad \land st.pending\_requests = \varnothing \\
\quad \land st.active\_computations = \varnothing
\end{array}
$$

All information in $SsState$ is visible to the security server.

$$
\begin{array}{l}
\textbf{SsView}[SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
ss\_view : SsState[SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\quad \leftrightarrow SsState[SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
\forall\, st_1, st_2 : SsState[SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\bullet\; (st_1, st_2) \in ss\_view \Leftrightarrow st_1 = st_2
\end{array}
$$

$$
\begin{array}{l}
SsComponent[SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\quad \widehat{=}\; SsGuar[SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\qquad \land SsRely[SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\qquad \land SsInit[SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\qquad \land SsView[SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS]
\end{array}
$$

5.3.4 Properties

We can prove some useful properties based entirely upon the specification of a generic security server. These properties can then be used as lemmas when analyzing a particular security server.

Theorem 5.8 *If a security server transition adds $ss\_response?$ to $pending\_responses$, and $(ss\_req, ans\_yes) \in answers(ss\_response?)$, then $ss\_req \in policy\_allows$ in the initial state of the transition.*

Proof: Only transitions $SsSendAffirmativeResponse$ and $SsSendNegativeResponse$ can add an $ss\_response?$ to $pending\_responses$. Both of them require

$$grants(ss\_response?) \subseteq policy\_allows.$$

Since

$$grants(ss\_response?) = \{\, ss\_req : SS\_REQ \mid (ss\_req, ans\_yes) \in answers(ss\_response?) \,\},$$

$ss\_req \in grants(ss\_response?)$, and we are done. □

Theorem 5.9 *If a security server transition adds $ss\_response?$ to $pending\_responses$, and $(ss\_req, ans) \in answers(ss\_response?)$ where $ans \notin \{ans\_yes, ans\_no\}$, then $(ss\_req, ans) \in ss\_information$ in the initial state of the transition.*

Proof: Only transitions $SsSendAffirmativeResponse$ and $SsSendNegativeResponse$ can add an $ss\_response?$ to $pending\_responses$. Both of them require

$$holds\_information(ss\_response?) \subseteq ss\_information.$$

Since

$$\begin{aligned} holds\_information(ss\_response?) = \{ & ss\_req : SS\_REQ;\ ans : ANS \\ & \mid (ss\_req, ans) \in answers(ss\_response) \wedge ans \notin \{ans\_yes, ans\_no\} \\ & \bullet (ss\_req, ans)\}, \end{aligned}$$

$(ss\_req, ans) \in holds\_information(ss\_response?)$, and we are done. □

The proofs of the following two theorems are analogous to the preceding two, and we state them without proof.

Theorem 5.10 *If a security server transition adds $req?$ to $pending\_requests$, and $(ss\_req, ans\_yes) \in volunteers\_answers(req?)$, then $ss\_req \in policy\_allows$ in the initial state of the transition.*

Theorem 5.11 *If a security server transition adds $req?$ to $pending\_requests$, and*

$$(ss\_req, ans) \in volunteers\_answers(req?)$$

where $ans \notin \{ans\_yes, ans\_no\}$, then $(ss\_req, ans) \in ss\_information$ in the initial state of the transition.

5.4 Composing the Generic Manager and Security Server

In this section we compose the generic manager and security server and analyze the properties of the composite system. We will, in this section, use the term *system* to refer to the composite of the generic manager and security server. Some important properties can be proven to hold based merely upon the generic manager and security server specifications. A *property* is defined as a set of system behaviors, and a *behavior* is a infinite sequence of states[9] that represents one execution history of a system. To say that a component (including a composite) satisfies a given property means that every behavior it allows is an element of the property. Since any specific manager or security server allows a subset of the behaviors of its generic counterpart, these properties will also hold for the composition of specific managers and security servers.

The composite state is $SystemState$ which contains all the fields of $MgrState$ and $SsState$. Note that the fields of $SharedState$ occur only once in $SystemState$ which means that the manager and security server use consistent values for these state fields.

[9] In the DTOS framework, a behavior also contains an infinite sequence of agents that are responsible for the state transitions in the corresponding state sequence. For simplicity, we will frequently ignore the agents when talking informally about behaviors and properties.

$$\begin{array}{|l}
\hline
\textit{SystemState}[M_DATA, SS_DATA, M_REQ, SS_REQ, RESP, ANS] \\
\textit{MgrState}[M_DATA, M_REQ, SS_REQ, RESP, ANS] \\
\textit{SsState}[M_REQ, SS_DATA, SS_REQ, RESP, ANS] \\
\hline
\end{array}$$

We map each manager state to the set of all *SystemState* that have the same value for each of the fields of *MgrState*. Similarly, we map each security server state to the set of all *SystemState* that have the same value for each of the fields of *SsState*.

The set of allowed initial states for the composition of two components is the intersection of the two sets (after mapping them into the composite state). This set of states is modeled by $system\_init$. Since $system\_init$ is nonempty, the manager and security server are composable.

$$\begin{array}{|l}
\hline
\textit{SystemInit}[M_DATA, SS_DATA, M_REQ, SS_REQ, RESP, ANS] \\
system_init : \mathbb{P}\ \textit{SystemState}[M_DATA, SS_DATA, M_REQ, SS_REQ, RESP, ANS] \\
\hline
\forall\, st : system_init \\
\bullet\ st.pending_responses = \emptyset \\
\quad \wedge\ st.pending_requests = \emptyset \\
\quad \wedge\ st.active_request = \emptyset \\
\quad \wedge\ st.sent = \emptyset \\
\quad \wedge\ st.obtained = \emptyset \\
\quad \wedge\ st.allowed = \emptyset \\
\quad \wedge\ st.responses = \emptyset \\
\quad \wedge\ st.active_computations = \emptyset \\
\hline
\end{array}$$

The composition theory requires that we define two *respect* relations when composing two components. These relations make explicit our assumptions about the ways in which the two components can affect each other's state. In particular, they indicate the data in the composite state that should not change during each component's transitions. One use of these relations is to specify that neither component modifies data that is considered private to its peer. Another use is to specify that neither component manipulates the interface that its peer has with a third component. (See the DTOS Composability Study [24] for more information in respect relations.)

In composing the generic manager and security server we will use respect relations that require each component to leave alone its peer's internal data.

$$\begin{array}{|l}
\hline
\textit{MgrSsRespect}[M_DATA, SS_DATA, M_REQ, SS_REQ, RESP, ANS] \\
mgr_rsp_ss : \mathbb{P}\,\Delta\ \textit{SystemState}[M_DATA, SS_DATA, M_REQ, SS_REQ, RESP, ANS] \\
ss_rsp_mgr : \mathbb{P}\,\Delta\ \textit{SystemState}[M_DATA, SS_DATA, M_REQ, SS_REQ, RESP, ANS] \\
\hline
mgr_rsp_ss = \{\ \Delta\ \textit{SystemState}[M_DATA, SS_DATA, M_REQ, SS_REQ, RESP, ANS] \\
\qquad \mid \Xi\ \textit{SsInternals}[M_REQ, SS_DATA, SS_REQ, RESP, ANS]\ \} \\
ss_rsp_mgr = \{\ \Delta\ \textit{SystemState}[M_DATA, SS_DATA, M_REQ, SS_REQ, RESP, ANS] \\
\qquad \mid \Xi\ \textit{MgrInternals}[M_DATA, M_REQ, SS_REQ, RESP, ANS]\ \} \\
\hline
\end{array}$$

The composability theory defines the **guar** of the composite to be

$$(mgr\_guar \cap mgr\_rsp\_ss) \cup (ss\_guar \cap ss\_rsp\_mgr).$$

This is modeled by mgr_ss_guar.

$$
\begin{aligned}
& SystemStep[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
& \mathrel{\widehat=} (\Xi\, SsInternals[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS] \\
& \qquad \land MgrGuarStep[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS]) \\
& \lor (\Xi\, MgrInternals[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
& \qquad \land SsGuarStep[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS])
\end{aligned}
$$

$MgrSsGuar[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS]$

$$mgr\_ss\_guar : \mathbb{P}\, SystemStep[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS]$$

$$mgr\_ss\_guar = SystemStep[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS]$$

The **rely** of the composite is the intersection of the two rely relations.

$MgrSsRely[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS]$

$$mgr\_ss\_rely : \mathbb{P}\, \Delta\, SystemState[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS]$$

$$
\begin{aligned}
& mgr\_ss\_rely = \{\ \Delta\, SystemState[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
& \qquad \mid \Xi\, MgrInternals[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
& \qquad \land \Xi\, SsInternals[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS] \\
& \qquad \land \Xi\, SharedInterpretation[M\_REQ, SS\_REQ, RESP, ANS] \\
& \qquad \land pending\_ss\_requests \sqsubseteq pending\_ss\_requests' \\
& \qquad \land pending\_responses' = pending\_responses\ \}
\end{aligned}
$$

For the composite of the generic manager and security server every field of the composite state is visible. However, as an aid to analysis, it may sometimes be useful to have fields of the composite state that are not visible to the composite. These fields can be used to encapsulate information about behaviors in which a given state occurs. For example, a field could be introduced that records whether a particular M_REQ has been performed. This information might not be kept explicitly by either the manager or security server but might still be used in defining properties that are desired of the composite. To leave this option open we will not specify a mgr_ss_view here.

The Composition Theorem states that if every transition of each component (when limited by its respect relation) is allowed by its peer component, then any property of one of the components is a property of the composite. To show that the theorem applies in this case, we must show that

$$mgr\_guar \cap mgr\_rsp\_ss \subseteq ss\_guar \cup ss\_rely \cup ss\_view,$$

and

$$ss\_guar \cap ss\_rsp\_mgr \subseteq mgr\_guar \cup mgr\_rely \cup mgr\_view.$$

We first show that $mgr\_guar \cap mgr\_rsp\_ss \subseteq ss\_rely$. Recall that ss_rely is defined to be

$$
\begin{aligned}
& ss\_rely = \{\ \Delta\, SsState[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS] \\
& \qquad \mid \Xi\, SsInternals[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS] \\
& \qquad \land \Xi\, SharedInterpretation[M\_REQ, SS\_REQ, RESP, ANS] \\
& \qquad \land pending\_ss\_requests \sqsubseteq pending\_ss\_requests' \\
& \qquad \land pending\_responses' \sqsubseteq pending\_responses\ \}.
\end{aligned}
$$

Since mgr_rsp_ss implies $\Xi\, SsInternals$, and $MgrStep$ implies $\Xi\, SharedInterpretation$, we only need to consider the $pending\_ss\_requests$ and $pending\_responses$ properties. Consideration of the individual manager operations shows that both of these properties are satisfied. There is a similar proof that $ss\_guar \cap ss\_rsp\_mgr \subseteq mgr\_rely$.

We can now apply the Composition Theorem to conclude that all of the lemmas, theorems and corollaries in Sections 5.2.4 and 5.3.4 apply to the composite as well. This supports the proof of the following theorem that relates the processing of requests to the contents of $policy\_allows$ and to the initial state under certain hypotheses about the manager and security server. The hypotheses of the Consistency Theorem are not the weakest possible. However, they are true of a large number of policies, and they lead to a theorem that is relatively easy to prove. We will discuss possible generalizations below.

While the first hypothesis deals solely with the manager and the second solely with the security server, the third hypothesis involves both. There are probably many ways in which the validity of the third hypothesis can be ensured in particular systems. For example, we could associate a set of SS_REQ with the ability to volunteer answers and constrain the initial state and $policy\_allows$ so that only the security server has this permission. This might allow any client to submit a request that volunteers answers, but the manager would only process those where the client is the security server. In a second approach the manager would receive such requests only on a special communication channel (e.g., a port in Mach) and the system would be set up to guarantee that only the security server can submit requests on this port.

The first hypothesis can be violated in ways that are not terribly obvious. For example, assume a manager defines $mgr\_policy(req?)$ based upon security labels attached to the subjects and objects involved in $req?$. If the manager allows these labels to change, this may cause new SS_REQs to be added to $mgr\_policy(req?)$. Such a manager would not have a monotone non-increasing mgr_policy. If the label can change between the time at which all permission checking is completed (i.e., when the computation is marked as $allowed$) and the time when the request is actually performed, then this manager can fail to be consistent with $policy\_allows$.

It might be possible to define an extended consistency theorem that allows relabeling under appropriate circumstances, but it is not clear what this theorem should look like. In particular, one must be careful about the interaction with the second hypothesis of the Consistency Theorem constraining removal of permissions from $policy\_allows$. It might be possible to design the security server so that permission to change a security label is granted only if appropriate permissions are allowed for the new label. However, we then need to consider these permissions as being implicitly granted and be sure that they are never removed from $policy\_allows$. It is clear that non-tranquillity of security labels complicates the definition of the security server and the analysis of the system. In this report we will instead prohibit relabeling in those cases where we wish to show consistency of the manager with $policy\_allows$.

Theorem 5.12 (Consistency Theorem) *Let Sys be a system with the following properties:*

1. *mgr_policy is monotone non-increasing,*

2. *no SS_REQ that has been either granted or volunteered by the security server is ever subsequently removed from $policy\_allows$, and*

3. *if a request that volunteers answers is processable (i.e., $(req\_num_1, req_1) \in active\_request$, $(req\_num?, status\_yes) \in allowed$ and $req? \notin denied\_requests$) in any state u, then there is a prior state in which the request was added to $pending\_requests$ during a security server transition.*

In Sys, if the manager processes a request $req?$ in some state s, then for each $r \in mgr\_policy(req?) \cap permission\_requests$ in s, either

- *$r \in retained(req?)$ in the system initial state, or*
- *$r \in policy\_allows$ in state s.*

Proof: Corollary 5.7 implies

1. $r \in retained(req?)$ in the initial state, or
2. there is some state s_1 preceding s and some $ans \neq ans\_no$ such that in s_1
   (a) $(r, ans) \in answers(ss\_response?)$ for some $ss\_response? \sqsubseteq pending\_responses$, or
   (b) for some request req_1 and request number req_num_1,

   $(r, ans) \in volunteers\_answers(req_1)$,
   $req_1 \notin denied\_requests$,
   $(req\_num_1, req_1) \in active\_request$, and
   $(req\_num_1, status\_yes) \in allowed$.

If $r \in retained(req?)$ in the initial state, we are done. Otherwise, consider Case 2. Since $r \in permission\_requests$, we know $ans \in \{ans\_yes, ans\_no\}$. Thus, Case 2 simplifies to

2. there is some state s_1 preceding s and some $ans \neq ans\_no$ such that in s_1
   (a) $(r, ans\_yes) \in answers(ss\_response?)$ for some $ss\_response? \sqsubseteq pending\_responses$, or
   (b) for some request req_1 and request number req_num_1,

   $(r, ans\_yes) \in volunteers\_answers(req_1)$,
   $req_1 \notin denied\_requests$,
   $(req\_num_1, req_1) \in active\_request$, and
   $(req\_num_1, status\_yes) \in allowed$.

Assume Case 2(a) is true. The initial state constraints require $pending\_responses = \emptyset$ in the system initial state. The **rely** for the composite states that $pending\_responses$ is not changed by the environment of the composite. Inspection of the manager operations shows that the manager does not add items to $pending\_responses$. Thus, $ss\_response?$ must have been added to $pending\_responses$ by some prior security server transition t with starting state s_t. Theorem 5.8 implies that $r \in policy\_allows$ in s_t. Since r is granted in transition t, Hypothesis 2 implies that it is still in $policy\_allows$ in state s.

Now consider Case 2(b). Since req_1 volunteers an answer and is processable in state s_1, Hypothesis 3 implies req_1 was added to $pending\_requests$ by some security server transition t with starting state s_2 prior to state s_1. Theorem 5.10 implies that $r \in policy\_allows$ in s_2. Since r is volunteered in transition t, Hypothesis 2 implies that it is still in $policy\_allows$ in state s. □

Corollary 5.13 (Complete Consistency) *Let Sys be a system that satisfies the hypotheses of the Consistency Theorem and in which, for some request $req?$, $policy\_allows$ in all reachable states contains all elements from $retained(req?)$ in the initial state. In Sys, if the manager processes $req?$ in some state s, then for each $r \in mgr\_policy(req?) \cap permission\_requests$ in s, $r \in policy\_allows$ in state s.*

The Consistency Theorem seems at first glance to be a generally useful theorem that would be satisfied by most manager/security server combinations. It does encapsulate a good deal of analysis. However, a closer examination reveals that it applies only to a certain class of policies. In fact, its conclusion is false for the optimal implementation of some dynamic policies such as Clark-Wilson and Dynamic N-Person.

First, the policy must be non-retractive (see Section 6) if the theorem is to be applied. Consistency between the manager's processing steps and $policy\_allows$ can also be achieved for retractive policies, but it is more difficult. We either need to provide a mechanism to make the manager immediately sensitive to changes in $policy\_allows$ or limit the circumstances in which $policy\_allows$ may change. We must show that if an SS_REQ is removed from $policy\_allows$, then, from that point on (or until the SS_REQ is reinserted into $policy\_allows$), no request is processed that needs this permission according to mgr_policy.

One possible way to implement this is that, if the security server needs to remove an SS_REQ that it has previously granted or volunteered, it must first request that the manager do the following:

- Remove all retentions of the SS_REQ.
- Determine all active requests for which the SS_REQ has been utilized and either mark them as failed or wait until they have been processed.
- Notify the security server that all traces of the SS_REQ have been removed from its state.

Only at this point can the security server change $policy\_allows$. If further actions of the security server were dependent upon this change to $policy\_allows$, it might be necessary for the security server to block while all this happens. Note that Step 2 means that checking a permission only once during the processing of a request may not be sufficient. Furthermore, if the security server does block, it might not be possible for the manager to complete all the requests identified in Step 2. Some of these requests might require additional security computations. To avoid deadlock, the manager must terminate these requests. It might also be possible to design manager requests so that they must be *committed* in the transaction processing sense. That is, the manager's persistent state would not be modified until all preliminary processing had occurred. The commit step would be atomic, and it would include confirmation that the required SS_REQs are still held.

Second, if a security policy has permissions that may only be granted once, this is also a situation in which the Consistency Theorem's conclusions are generally false. (The ideal implementation of Clark-Wilson will prevent the creation of multiple tasks with the same context. It therefore is an example of such a policy.) The permission will be removed from $policy\_allows$ as soon as it is granted. It will not be there at the time when the permission is used in the manager. This is not really a case of permission retraction since the kernel may still be allowed to use the permission after it has been removed from $policy\_allows$ (although the permission would probably be marked non-cachable and used only once). We should point out that it might be possible to patch this problem by designing the security server to block and wait for a notification that the permission has been used. At this point it could modify $policy\_allows$. However, such behavior is not necessary to implement a Clark-Wilson security server, only to get it to comply with the Consistency Theorem. Since it would complicate the implementation, there seems little value to this approach.

We could also consider weaker types of consistency. Two possible examples are

Check Completion Consistency - If a request is marked as allowed ($status\_yes$) in state s, then for every SS_REQ r required for the request by mgr_policy in s, $r \in policy\_allows$.

Unless all permission checking is atomic, it is not clear that this could be achieved for retractive policies any more easily than our earlier definition.

Invocation Consistency - If a request that was invoked in state s is eventually processed, then for every SS_REQ r required for the request by mgr_policy in s, $r \in policy\_allows$ in state s.

Although further investigation is necessary, an Invocation Consistency Theorem could probably be proven. The hypotheses would probably include that mgr_policy is monotone non-*de*creasing and $policy\_allows$ is monotone non-increasing.

A more fundamental question is whether such consistency properties would be of any value. Consistency is only of interest if it supports the definition and analysis of a security server for some interesting security policy. Most policies are concerned with events such as reading and writing data to files, not with the completion of permission checking or even the invocation of a request. It is usually not a problem for an unallowed request to be invoked, as long as it is not executed. To take a specific example, the ORCON policy is not supported by Invocation Consistency. Assume A is not allowed to read certain data that B has read and consider the following sequence of events:

1. A invokes a read request on file f and is granted permission.

2. B invokes a write request on f and is granted permission. This causes read permission for A to f to be removed from $policy\_allows$, but A has already received this permission and no subsequent confirmation is done.

3. B writes data to f that A is not allowed to read.

4. A reads f.

Although this sequence of events does not violate Invocation Consistency, it *does* violate the ORCON policy. Similar problems could occur with environment-sensitive policies (see Section 6). For example, if a policy requires that all write access to f is be revoked at 5:00 PM, it might not be good enough to say that any write operation that is invoked prior to 5:00 PM can complete after that time. We might normally assume that any such operation would conclude shortly after 5, and that there is not really any problem. However, it might be possible for an attacker to delay completion of the invoked operation until much later (e.g., by manipulating scheduling, system load, etc.), and this could pose a significant violation of the high-level policy.

In conclusion, it seems likely that few managers will provide sufficient support for retractive policies unless they are specifically designed with that goal in mind. Furthermore, in this case the design of the manager would probably be greatly influenced by the desire to support retractive policies.

5.5 Complete System

We use composability theory to analyze the behavior of the combination of any given manager and security server. However, in Section 6 we discuss some general characteristics of policies, and we classify a number of policies in a policy lattice that is derived from those characteristics. The characteristics are also formally defined. To prepare for this, in this section we define several concepts and functions relating to the combination of the manager and security server that will be useful.

We define the functions $Action\_init\_state$ and $Action\_final\_state$ to return the initial and final state, respectively, for any step.

$$
\begin{array}{l}
[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
Action\_init\_state : \\
\quad SystemStep[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\quad \longrightarrow SystemState[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
Action\_final\_state : \\
\quad SystemStep[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\quad \longrightarrow SystemState[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
\forall SystemStep[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\bullet\ Action\_init\_state(\theta SystemStep) = \theta SystemState \\
\quad \land Action\_final\_state(\theta SystemStep) = \theta SystemState'
\end{array}
$$

In the following we focus on sequences of operations occurring in the system. Each individual operation is a subset of $SystemStep$ representing the set of state transitions that can be performed by that operation. We define the set OP as the set of all subsets of $SystemStep$ and OP_SEQ as the set of all sequences of elements of OP. We also define $STATE\_SET$ as the set of all subsets of $SystemState$.

$$
\begin{array}{l}
OP[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\quad == \mathbb{P}\ SystemStep[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
OP\_SEQ[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\quad == \mathrm{seq}\ OP[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
STATE\_SET[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\quad == \mathbb{P}\ SystemState[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS]
\end{array}
$$

The operations of a system include those outlined in Sections 5.2 and 5.3. The set $system\_ops$ contains each of these operations. Note that each operation must have the same signature, so we use the operation definition schemas as restrictions of the general $SystemStep$ schema. We allow operations in this set to overlap. In fact one operation may be a specialization (i.e., subset) of another. For example, each operation that processes a manager request will be a specialization of the $MgrProcessRequest$ operation.

$$
\begin{array}{l}
\textit{SystemOperations}[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
system\_ops : \mathbb{P}\ OP \\
\hline
\{\{SystemStep[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\qquad \mid MgrReceiveRequest[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS]\}, \\
\quad \{SystemStep[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\qquad \mid MgrRequestComputation[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS]\}, \\
\quad \{SystemStep[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\qquad \mid MgrSendNotification[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS]\}, \\
\quad \{SystemStep[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\qquad \mid MgrReceiveResponse[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS]\}, \\
\quad \{SystemStep[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\qquad \mid MgrNegativeResponse[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS]\}, \\
\quad \{SystemStep[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\qquad \mid MgrAffirmativeResponse[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS]\}, \\
\quad \{SystemStep[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\qquad \mid MgrDenyRequest[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS]\}, \\
\quad \{SystemStep[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\qquad \mid MgrAcceptRequest[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS]\}, \\
\quad \{SystemStep[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\qquad \mid MgrProcessRequest[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS]\}, \\
\quad \{SystemStep[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\qquad \mid SsReceiveRequest[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS]\}, \\
\quad \{SystemStep[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\qquad \mid SsSendNegativeResponse[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS]\}, \\
\quad \{SystemStep[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\qquad \mid SsSendAffirmativeResponse[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS]\}, \\
\quad \{SystemStep[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\qquad \mid SsMgrRequest[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS]\}, \\
\quad \{SystemStep[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\qquad \mid SsInternalTransition[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS]\}\} \\
\subseteq system\_ops
\end{array}
$$

We define a system to be the system state, the initial state and the operations that may be performed.

$$
\begin{array}{l}
System[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\quad \widehat{=}\ SystemState[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\qquad \land\ SystemInit[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\qquad \land\ SystemOperations[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS]
\end{array}
$$

The expression $Apply\_op(op, state\_set)$ denotes the set of all states s such that the system could reach state s by executing operation op in some state in the set $state\_set$. Note that op might not be applicable to a given state in $state\_set$ (i.e., its preconditions are not met), or it might produce multiple possible final states (i.e., it is nondeterministic). Thus, $Apply\_op(op, state\_set)$ need not have the same number of elements as $state\_set$.

$$
\begin{array}{l}
[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
Apply\_op : OP[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\quad \times STATE\_SET[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\quad \longrightarrow STATE\_SET[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
\forall state\_set : STATE\_SET[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS]; \\
\quad op : OP[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\bullet Apply\_op(op, state\_set) \\
\quad = \{ new\_state : SystemState[M\_DATA, SS\_DATA, \\
\qquad M\_REQ, SS\_REQ, RESP, ANS] \\
\qquad \mid (\exists old\_state : state\_set; transition : op \\
\qquad\quad \bullet Action\_init\_state(transition) = old\_state \\
\qquad\qquad \wedge Action\_final\_state(transition) = new\_state) \}
\end{array}
$$

The function $Execute$ extends $Apply\_op$ for a sequence of operations.

$$
\begin{array}{l}
[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
Execute : STATE\_SET[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\quad \times OP\_SEQ[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\quad \longrightarrow STATE\_SET[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
\forall state\_set : STATE\_SET[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS]; \\
\quad op\_seq : OP\_SEQ[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\bullet ((op\_seq = \langle\rangle \Rightarrow Execute(state\_set, op\_seq) = state\_set) \\
\quad \wedge (op\_seq \neq \langle\rangle \\
\qquad \Rightarrow Execute(state\_set, op\_seq) \\
\qquad\quad = Execute(Apply\_op(head\ op\_seq, state\_set), tail\ op\_seq)))
\end{array}
$$

An operation sequence is *valid in a state* S if and only if the result of executing the sequence from the initial set of states $\{S\}$ is non-empty. A state S is *reachable* if and only if there exists an operation sequence op_seq such that S is in the set of states resulting from the execution of op_seq beginning in the initial state $system\_init$.

$$
\begin{array}{l}
[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
Valid\_op\_seqs : SystemState[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\quad \longrightarrow \mathbb{P}\ OP\_SEQ[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
Reachable : System[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\quad \longrightarrow STATE\_SET[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
\forall S : SystemState[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS]; \\
\quad op\_seq : OP\_SEQ[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\bullet op\_seq \in Valid\_op\_seqs(S) \Leftrightarrow Execute(\{S\}, op\_seq) \neq \varnothing \\
\\
\forall S : SystemState[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS]; \\
\quad system : System[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\bullet S \in Reachable(system) \\
\quad \Leftrightarrow (\exists op\_seq : \mathrm{seq}\ system.system\_ops \\
\qquad \bullet S \in Execute(system.system\_init, op\_seq))
\end{array}
$$

We will want to talk about the subsequences of an operation sequence. We first define the expression $Sub\_seqs(s)$ to return the set of all subsequences of s. A subsequence of s is any sequence obtained by omitting zero or more elements from s and leaving the remaining elements in the same order. This includes the empty sequence as well as s.[10]

$$
\begin{array}{l}
[X] \\
\hline
Sub\_seqs : \mathrm{seq}\, X \longrightarrow \mathbb{P}(\mathrm{seq}\, X) \\
\hline
\forall\, s, t : \mathrm{seq}\, X \\
\bullet\, (t \in Sub\_seqs(s) \\
\quad \Leftrightarrow (\exists\, indexes : \mathbb{P}\, \mathbb{N} \\
\qquad \bullet\, t = indexes \upharpoonleft s))
\end{array}
$$

We define $Manager\_requests(op\_seq)$ to be the subsequence of op_seq consisting of the steps where the manager is processing a request.[11]

$$
\begin{array}{l}
[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
Manager\_requests : OP\_SEQ[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\quad \longrightarrow OP\_SEQ[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
\forall\, op\_seq : OP\_SEQ[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\bullet\, Manager\_requests(op\_seq) = op\_seq \\
\quad \upharpoonright \{op : OP[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\qquad \mid op \subseteq \{SystemStep[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\qquad\qquad \mid \Xi\, SsInternals[M\_REQ, SS\_DATA, SS\_REQ, RESP, ANS] \\
\qquad\qquad \wedge\, MgrProcessRequest[M\_DATA, M\_REQ, \\
\qquad\qquad\qquad SS\_REQ, RESP, ANS]\}\}
\end{array}
$$

Because of the system state elements that are required to be invariant in $MgrProcessRequest$ and the fact that ss_data is required to change in $SsInternalTransition$ there is no ambiguity in distinguishing the processing of a request from other system steps. No other system transition qualifies as a valid element of $MgrProcessRequest$.

[10] The Z operation $U \upharpoonleft s$ returns a sequence that contains the elements of s with an index in the set U in the order in which they occur in s.

[11] The Z operation $s \upharpoonright V$ returns a sequence that contains the elements of s that are in the set V in the order in which they occur in s.

Section 6

Security Policy Lattice

In this section we develop a lattice of security policies. We begin by defining a list of policy characteristics. This set of characteristics defines a lattice of policies. For any interesting policy, we can place the policy at a node in the lattice by determining the set of policy characteristics that it has.

This lattice is useful in analyzing the capabilities of a manager to support security policies. We first determine the set of policy characteristics that are supported by the manager. This places the manager in the lattice at a node N. The manager can support each policy that is located at a node that is dominated by or equal to N. Furthermore, the placement of policies in the lattice will help to identify policy characteristics that are important to support in a manager for which policy flexibility is a design goal. Finally, if m managers are to be analyzed with respect to p policies, the complexity of doing this with the lattice is $m + p$ as opposed to mp for analyzing each manager-policy combination separately.

6.1 Security Policy Characteristics

In this section we informally describe the dimensions along which we will classify security policies. They are

Input — Many policies vary on the amount of information used in a single policy decision. For example, Type Enforcement considers only two security contexts, a domain and a type. In contrast, Clark-Wilson (in its pure version discussed below) considers an arbitrary number of security contexts contained in an access triple when making a single policy decision. At the Mach microkernel level, it would probably be useful to control port requests based upon a triple containing the client, the target task, and the target port. We will use C to denote the number of contexts used in making a policy decision and will say that $C = 2$ for policies such as Type Enforcement and $C > 2$ for policies like Clark-Wilson.

In addition to security contexts a policy decision might also take into account other data describing the operation for which a subject is requesting permission. For example, a subject might wish to change its scheduling priority. A policy might associate a range of allowed priority values with each subject and include in its decision-making process an examination of the priority requested by the subject. Since the priority is most likely a parameter of the kernel request submitted by the subject, we will say that a policy is *parametric* if it takes this type of information into account in making its decisions.

We note here that with regard to both the number of contexts and the inclusion of parameters we are considering security server permission requests, not manager requests. This policy characteristic deals with the "language" used to request permissions. The processing of a single manager request may involve multiple permission requests to the security server. The number and content of these requests will certainly be affected by the manager request, including the values of the parameters and the number of entities in the request. This, by itself, does not make the policy $C > 2$ nor parametric. If the manager never supplies more than two contexts in permission requests and never includes any parameter information, then the policy decisions made in the security server are based upon just two contexts.

Sensitivity — While policy decisions are obviously influenced by the input (i.e., contexts and parameters) to the policy, many policies are also sensitive to other information. Such sensitivity can cause a policy to change so that it might make different decisions in two cases that have identical input. This dimension indicates whether a policy is sensitive to other information and, if so, the kind of information.

We will call a policy that is insensitive (i.e., decisions are based solely upon the input information) a *static* policy. Static policies can change only as a result of a specific policy modification request by a *privileged* user. MLS policies are typically static. If a policy is not static, we call it *dynamic*. A Chinese Wall policy is dynamic. Initially, a user can read all files in the system. However, if a user u reads a file f, that user may no longer read any files that are in the conflict of interest class for f except for those that are in the same company data set as f. Thus, as a side effect of reading a file, u has lost read access to other files, and the policy has changed automatically. In terms of sensitivity, the Chinese Wall policy is sensitive to the history of file read accesses.

For a dynamic policy, it is important to know which events cause the policy to change — the events to which it is sensitive. We identify several types of sensitivity. A single policy may have multiple sensitivities.

History — Many policies are sensitive to accesses to files (see the Chinese Wall example above). We generalize this concept and say that any policy that is sensitive to the execution of manager requests is *history-sensitive*.

Environment — Other policies are sensitive to properties such as the time of day (e.g., no write access between 5 PM and 9 AM) or operational mode of the system (e.g., training, normal, emergency). We will call such policies *environment-sensitive*.

Discretionary — A policy such as IBAC changes as a result of an explicit request by a user (privileged or unprivileged) to change the portion of the policy that applies to objects owned by that user. We call such policies *discretionary*.

Relinquishment — In addition to making requests for permissions (e.g., read access to a file) processes also logically *relinquish* the permissions that they hold. For example, when a process closes a file, it is relinquishing its access to that file. True relinquishment implies that it is impossible for the process to reopen the file without making a new permission request. When a process terminates, it relinquishes all the permissions that were granted to it. Note that relinquishment and history-sensitivity are related in that permission is often required to perform a relinquishment action such as deallocating a region or terminating a task. They are different in that the permission(s) relinquished are different from the permission that enables the relinquishment. When a task deallocates a region to which it has read permission, it adds deallocation permission to the history and relinquishes read permission to the region.

If a policy changes in response to relinquishment, we will say it is *relinquishment-sensitive*. For example, if we depended upon the security policy to enforce locking of files (e.g., at most one process using each file at a time), the policy would change whenever a process opened a file. When a process closed a file, the policy should change again to allow access to the file for other processes.[12] Although our discussion has focused on file access, relinquishment can in principle apply to any permission.

[12] Locking can in most cases be handled by the file server rather than the Security Server. However, if file locking is related in some way to security policy decisions, then the policy will probably play a role in the control of locking.

The lines between these sensitivity types are somewhat vague and arbitrary. For example, if we defined history-sensitive as "sensitive to any system event" rather than "sensitive to the execution of a manager request", then history-sensitivity would include environment, discretionary and relinquishment sensitivity since changes in the environment, requests to change discretionary policy and relinquishment are all system events. By restricting history-sensitivity to consider only the execution of manager requests we are singling out those policies where the Security Server must observe what the manager is doing. We consider sensitivity to operational mode to be environment sensitivity. However, to implement a change in operational mode might require a manager request, and it probably is initiated by a user and thus has a discretionary flavor. We choose not to consider it discretionary due to its potentially global effect on security decisions, reserving discretionary sensitivity for modifications performed by a user to the parts of the policy dealing with objects owned by the user. We choose not to consider it history sensitive since the primary purpose of a change-of-mode request would be to modify the Security Server's internal data, not that of the manager. Intuitively, it is really a Security Server request rather than a manager request. Relinquishment sensitivity is really just a special case of history sensitivity.

Retraction — Many dynamic policies need the ability to retract permissions that have been previously granted. We call such policies *retractive*. Note that this is a fairly narrow definition of retraction. We only consider retraction of *granted* permissions. If the removed permission is merely grantable and has never actually been granted, this will not make the policy retractive. Consider the Chinese Wall example, and let f_1 be a file in the same conflict of interest class as f such that f and f_1 are in different company data sets. Initially, read access for u to f_1 is grantable but not granted. When u reads f, read access for u to f_1 becomes nongrantable. We know at this point that read access was never granted for u to f_1 since otherwise read access to f could not be granted. The policy does not need to retract a permission that it has never granted, so no retraction is necessary here. Retraction is similar to relinquishment. The difference is that retraction is an action of the policy to forcibly remove permissions from a process while relinquishment is a voluntary action of a process to give up those permissions.

Transitivity — In defining security policies we are often concerned with the ways that multiple subjects can combine to obtain information or affect the system state and output. For example, in an MLS system a subject may write to an object only if the level of the object dominates the level of the subject. Since the dominance relation on levels is transitive, so is the permission to write. That is, if a subject at level A can write to an object at level B, and a subject at level B can write to an object at level C, then a subject at level A can write to an object at level C. An analogous relationship exists for reading. Such policies are called *transitive*. More generally, we will say that in a transitive policy if a subject A can modify a data item d_A (e.g., a file or a piece of system state) and if a subject B can detect the modifications made by A to d_A and can itself modify a data item d_B, then A can also modify d_B.

Transitivity is not a desirable property in all circumstances. For example, consider the problem of output labeling in an MLS environment. Suppose every page that is printed must be marked with its classification level. A page labeling program could be used to do this automatically. The normal way to print a document then would be to send it to the page labeler program to be marked and then pipe the result into the print spooler. A user must be able to send information to the labeler and the labeler must be able to send information to the print spooler. It is important, however, that users not be allowed to send information directly to the print spooler. If this happened, unlabeled pages could be printed. Thus, an *intransitive* policy is needed here. It should be noted that few if any policies are purely transitive without any exceptions. For example, an MLS system will typically have trusted subjects that are able to downgrade information to a lower classification level. This downgrader can be used by other subjects to write downgraded information to levels to which they themselves cannot write.[13]

Editorial Note:
Another characteristic to consider is whether entities must have unique SIDs.

6.2 Classification of Some Well-Known Policies

Now we will consider a number of well-known security policies and classify them with respect to the dimensions identified in Section 6.1. We will also discuss whether the policy can be implemented in the current DTOS prototype. To gain a better understanding of the issues involved, some variants of these security policies that have slightly different characteristics from the originals will also be considered. These variants are called Nonretractive IBAC, Piecemeal Clark-Wilson, Piecemeal Dynamic N-person, Locking ORCON and Static Chinese Wall. The discussion in this section is summarized in Table 1. Section 6.3 will summarize the conclusions regarding the ability of DTOS to support the policies in this section and will give a more general discussion of the issues involved.

6.2.1 Type Enforcement

In Type Enforcement (TE) subjects with identical access privileges are grouped into a *domain* and objects that may be accessed in precisely the same ways by each domain are grouped into a *type*. Access controls then restrict the access of domains to types. The other policies discussed in this report have explicit goals such as preventing downward flow of classified information, ensuring data integrity, and preventing fraud or insider trading. TE, by itself, has no similar goal. Thus, we consider TE to be more a framework through which security policies can be implemented than a policy in its own right. We discuss it here since the DTOS security mechanisms are based upon TE, and any policy that can be achieved through TE can be supported by DTOS.

TE makes decisions based upon the domain and type, so it is a $C = 2$, nonparametric policy. We consider it static, but if mechanisms are provided for changing the domain definition table, TE can be used as a basis for dynamic policies. TE by itself does not require retraction, and it can support intransitive policies.

6.2.2 IBAC

Identity-Based Access Control (IBAC), also known as Discretionary Access Control (DAC), has as its guiding principle the idea that control of access to a file is up to the individual who owns that file. This owner determines the permissions of other users to access the file. The owner may change the permissions at any time. Thus, the policy is sensitive to discretionary controls. It is trivial for a user to set file permissions in a way that makes the policy intransitive. Since an IBAC policy makes most decisions based upon the user associated with a process and the access controls assigned to a file, IBAC is essentially $C = 2$ and nonparametric. There are,

[13] It is also worth noting that trusted pipelines, such as the page labeler, and downgraders seem to be the main, if not only, places where intransitivity is needed in systems that are otherwise MLS.

Policy	Input		Sensitivity							
	$C > 2$	Parametric	Dynamic/Static	Discretionary	History	Environment	Relinquishment	Retractive	Transitive	DTOS Supports
MLS/BLP			S						•	Y
Biba			S						•	Y
Type Enforcement			S							Y
IBAC										
Retractive			D	•				$\bullet_{rwe}$		N_b
Nonretractive			D	•						Y
Clark-Wilson										
Pure	•		S							N_a
Piecemeal			D		$\bullet_{we}$					Y
Dynamic N-Person										
Pure	•		D		$\bullet_{e}$					N_a
Piecemeal			D		$\bullet_{we}$					Y
ORCON										
Pure			D		$\bullet_{rwe}$			$\bullet_{re}$		N_b
Locking			D		$\bullet_{rwe}$		•			N_c
Chinese Wall										
Pure			D		$\bullet_{rw}$					Y
Static	•		S							N_a

For history-sensitive policies the entry includes an annotation indicating the actions that cause the policy to change. The history-sensitive policies studied here are sensitive to reading (r), writing (w) and executing (e). For policies that are retractive we indicate the permissions that need to be retracted using the same annotations as for history-sensitivity. The rightmost column in the table indicates the policies supported by DTOS. A "Y" indicates support while an "N" indicates lack of support. A subscript attached to "N" denotes the reason why DTOS does not support the policy. The meaning of the subscripts is

- N_a - Arbitrary number of SIDs in a request,
- N_b - Retraction,
- N_c - Relinquishment.

Table 1: Characteristics of Security Policies

however, examples where a request in an IBAC system involves checks on more than two entities and where the specific checks required depend upon parameters of the system request. For example, consider unlinking a UNIX file from a directory with the sticky bit set. The permission logic is:

- the user must have write access to the directory, and
- either
  - the user must own the directory, or
  - the user must own the file being unlinked, or
  - the user must be the superuser.

So, the logic must deal with the contexts of three entities: the process, directory and file. Whether this makes the policy $C > 2$ depends upon how the policy is implemented. If the manager breaks the permission checking up into the individual requests indicated in the bulleted items above and sends each as a separate security server request, then the decisions made in the security server are still based upon a pair of contexts. If on the other hand the manager sends a permission request $unlink\_file(file, dir, process)$ to the security server, then the implementation of the policy is $C > 2$. We anticipate that the most common implementation will be the former one involving multiple permission requests,[14] so we will classify IBAC as $C = 2$. The permission checks required are sufficiently independent that no semantic information is lost by doing so.

As an example of parametric control, consider the UNIX `chmod` command. If the `setgid` bit is set in the new file mode, then the user must be a member of the group that owns the file. As with the example involving unlinking a file, this does not necessarily imply that the policy is parametric. Again, we assume the implementation involving multiple permission requests to be the typical one, and we classify IBAC as nonparametric.

Retraction is slightly complicated. A file owner may remove accesses of other users for a file at any time. If another user is reading a file at the time when read access is removed, the access could be disabled immediately (retractive) or it could remain as long as the file is open (nonretractive). In Table 1 we include both versions labeled as Retractive and Nonretractive, respectively. DTOS supports only the nonretractive version.

6.2.3 MLS and Biba

MLS and Biba are both defined in terms of a lattice of levels. In MLS each level represents a classification (e.g., unclassified, secret and top secret) with high classifications at the top of the lattice. In Biba each level represents a given amount of integrity with low integrity at the top of the lattice. The same restrictions on reading and writing apply to both lattices. A process executing at level l may only read a file whose level is dominated by l and may only write to a file whose level dominates l.

Both of these policies are static. Since decisions are based only upon the level of the process (subject) and file (object), $C = 2$ and the policies are not parametric. Both policies are also nonretractive and transitive. DTOS is able to support both policies.

[14] Actually, we expect that in many implementations directory and file ownership will be managed entirely within the file system and identity as the superuser will be managed entirely within the operating system. Thus, only one permission request will be sent.

6.2.4 Clark-Wilson

6.2.4.1 Pure Clark-Wilson The Clark-Wilson policy is defined in terms of access triples of the form $(UserID, TP, (CDI_1, CDI_2, \ldots, CDI_n))$ rather than the more common access pair $(subject, object)$. Although this policy can be implemented through access pairs by requiring subjects to submit a properly constructed sequence of permission requests, it is more consistent with the definition of the policy to think of a subject requesting access to a *set* of CDIs all at once with one security decision made for the entire set. Access to the CDIs here is all or nothing. This type of system interaction is quite different from what typically occurs in a UNIX system where programs request access to a single file at a time. However, it is closely linked to the way in which file access has historically been controlled on many commercial mainframe systems using COBOL. To run a program on such a system, a user must prepare and submit a sequence of instructions in a Job Control Language (JCL). These JCL instructions state the program that is to be run along with the name and desired access mode of each file to be used by the program. Each file is assigned some code (e.g., a unique letter) by the JCL instructions. These codes are used inside the program to refer to all files. No other way to access files is provided to the program. Before the program begins execution, the system tries to obtain access to each of the files requested in the JCL instructions. If access to any file cannot be obtained, the entire execution terminates without performing a single instruction of the COBOL program. We call this interpretation of Clark-Wilson *Pure Clark-Wilson*.

In many respects Pure Clark-Wilson is a very simple policy. It is static in that no decisions on access to a set of CDIs depend upon earlier decisions. A static policy needs no history, and unless it also includes discretionary control it is nonretractive. However, Pure Clark-Wilson assumes a style of interaction between individuals and the system that is at least similar to the JCL example discussed above. It also means that the Security Server must handle requests containing an arbitrary number of SIDs. This is outside the capabilities of DTOS.[15] A Clark-Wilson policy is almost certainly intransitive since reading is unconstrained but each user/TP pair is granted write access to only certain files.

6.2.4.2 Piecemeal Clark-Wilson If we wish to apply the Clark-Wilson policy to a system that follows the UNIX paradigm for file access (i.e., one file at a time), the way that we think about Clark-Wilson must change dramatically. We will call this view of Clark-Wilson the *Piecemeal Clark-Wilson* policy.[16]

The first thing we observe is that Piecemeal Clark-Wilson is not a static policy since obtaining access to a particular CDI can affect future access decisions. For example, assume a transformation procedure TP_1 is certified to manipulate the following sets of CDIs:

$$\{\{CDI_1, CDI_3\}, \{CDI_2, CDI_3\}, \{CDI_2, CDI_4\}\}$$

Initially, TP_1 could obtain $Have\_write$ permission to any of $CDI_1, \ldots, CDI_4$. However, if TP_1 is granted $Have\_write$ permission to CDI_2, then the policy should no longer grant $Have\_write$ permission to CDI_1 since there is no set above that contains both CDI_1 and CDI_2.

Since the policy changes in response to file access we consider it to be history-sensitive with writing and executing as the events that cause policy changes.[17] Retraction is not necessary

[15] A complex request with arbitrarily many SIDs can be broken down into a sequence of requests based upon SID pairs (which is what happens with Piecemeal Clark-Wilson). However, it is important for Pure Clark-Wilson that this sequence of requests be recombined to form the access triple and that access be all or nothing.

[16] This is the version of Clark-Wilson described in Section 9.

[17] We consider Clark-Wilson to be primarily concerned with the writing of CDIs and not the reading of them. Input to TPs is unconstrained by the policy, and the TPs themselves are responsible for making sure that they are given correct

since actions can only affect future access decisions and never invalidate prior decisions. As with the pure version, a Piecemeal Clark-Wilson policy is almost certainly intransitive. Since DTOS can support nonretractive, history-sensitive policies, Piecemeal Clark-Wilson can be implemented on a DTOS system.

We note here that the pure and piecemeal versions of Clark-Wilson exercise equivalent control over the system. It is only the implementation of the policy that changes. The fact that this has a profound effect on the classification of the two policies suggests that distinctions such as static versus dynamic, while appearing to describe a policy in abstract terms, implicitly incorporate details of how the policy might be implemented. We discuss this further in Section 6.4.

6.2.5 Dynamic N-Person

A dynamic N-person policy is similar to a Clark-Wilson policy except that the roles played by the users are not fixed as they are in Clark-Wilson. To see why fixed roles can be undesirable, let us consider the following example. In a Clark-Wilson policy we might allow a purchasing clerk to initiate a purchase order and then require that the purchasing supervisor approve the order. In order to require that two people be involved in this transaction we must prevent the supervisors from initiating purchase orders and prevent the clerks from approving them. This enforces separation of duty. One drawback of this static assignment of roles is that it does not allow any flexibility in responding to special situations such as the absence of all purchasing clerks (e.g., due to illness or vacation). Dynamic N-person policies add this flexibility while still maintaining separation of duty. The purchasing supervisors are allowed to initiate a purchase order, but a single supervisor cannot both initiate and approve any given order. Instead, the order must be approved by another supervisor. Thus, a supervisor is allowed to fill more than one role but can fill at most one role with respect to any given order.

To define such a policy, we must have a concept of a *valid sequence* of TP executions. The policy defines the valid sequences and for each step in the sequence the CDIs that may be manipulated by the step. When a TP process is created, it must either establish a new sequence or continue an existing one. The granting of permission to create the TP process would be sensitive to the valid TP sequences. The granting of CDI accesses could then be decided on the basis of the user, TP and perhaps the sequence.

6.2.5.1 Pure Dynamic N-Person

As with Clark-Wilson we distinguish two versions of dynamic N-person policy. In the pure version, access for a TP process to the CDIs is all-or-nothing just like it is with Pure Clark-Wilson. This policy has $C > 2$. Unlike Pure Clark-Wilson it is dynamic. This comes not from the CDI access, but from the sensitivity to the position of the TP process in the valid sequence. This policy is nonretractive and will usually be intransitive. It cannot be supported by DTOS due to lack of input flexibility.

6.2.5.2 Piecemeal Dynamic N-Person

This version corresponds to Piecemeal Clark-Wilson — CDIs are accessed one at a time, and the access permissions are sensitive to the history of CDI accesses granted to the TP process. It is thus sensitive to the write and execute accesses granted to files. $C = 2$, and the policy is nonretractive and generally intransitive. This policy can be supported by DTOS.

input. Thus, the reading of a CDI does not cause the policy to change. This focus on writing rather than reading is consistent with the author's experience in a business information systems environment. With the exception of sensitive information (e.g., personnel files) programmers were allowed to execute programs that read virtually any files on the system. However, they were not allowed to write any production data files (the CDIs of the system). This allowed them to debug programs using production data while preventing them from destroying information or committing fraud.

6.2.6 ORCON

6.2.6.1 Pure ORCON In the ORCON policy if a process p_1 reads a file f_1 the permissions associated with f_1 place a new upper bound on the permissions associated with any file f_2 to which p_1 subsequently writes. If a process p_2 is reading f_2, the change to the permissions for f_2 is propagated to the files to which p_2 is writing. This constitutes a change of policy so ORCON is dynamic. Since the change is in response to file access, ORCON is history-sensitive with sensitivity to reading, writing and executing. In the above example it is possible that p_2 might no longer have read or execute access to f_2 when its ACL is changed as a result of p_1 writing to it. In this case ORCON must be able to retract read and execute permissions. In DTOS, read, write execute permissions are what we call *migrating* permissions. A migrating permission is one that is retained in the protection bits associated with memory. When one these permission is flushed from the cache, a "flush thread" is spun off to search the page tables flushing the associated protection bits. This thread will eventually remove all the migrated permissions. However, the cache flush request can return before the flush thread finishes its job, and there is currently no way for the security server to determine when the flush thread has completed the operation. Thus, DTOS provides rather poor support for any policy that needs to flush one of these migrating permissions.

Since read permission migrates in the DTOS microkernel, the prototype cannot support the retractions required in ORCON. We also recall the discussion in Section 5.4 regarding the general difficulties in supporting retraction. This suggests that even if we modified DTOS so that the security server could determine when the page table protection bits have been cleared, DTOS would still not provide good support for retractive policies. The security server would also need the ability to abort or restart active kernel requests that make use of the permissions being retracted.

6.2.6.2 Locking ORCON We now consider a modified version of ORCON which we call *Locking ORCON*. Retraction only needs to be done in ORCON if the process p_2 currently has read access to the file f_2 when process p_1 writes to f_2. If p_2 does not have read access when the writing occurs (i.e., read access was either never granted or was relinquished), then read access can simply be denied when p_2 requests access at a later time. If we can guarantee that no file may be open for reading and writing by different processes at the same time, then no retraction is necessary. Locking ORCON consists of standard ORCON together with this new restriction that a process must have a file locked (i.e., no other processes have it open for reading) in order to write to the file.

> *Editorial Note:*
> We suspect that even Locking ORCON cannot practically be supported by DTOS since the Security Server has no reliable way to find out that the file is no longer locked (even if p_2 terminates). Even worse, if the Security Server itself is enforcing the locking as part of the policy, then once a file is read it would be tied up in perpetuity. The SS needs to be able to respond to relinquishment. This also relates to the issue of how the Security Server recognizes the destruction of a process.

6.2.7 Chinese Wall

6.2.7.1 Pure Chinese Wall As described in Section 6.1 the Chinese Wall policy is history-sensitive. Its sensitivity to reading has already been discussed. It is also extremely sensitive to writing. If a user has write access to any file in an unsanitized data set D, then that user may only read files that are either in D or in the sanitized data set. Thus, granting process p write access to a file makes read access ungrantable for p to most of the files in the system.

This observation casts some question on the claim of Brewer and Nash that a Chinese Wall system may be operated with a number of users that is no more than the largest number of data sets in any conflict of interest class. Since for each company data set there would most likely be at least one user with write permission to that data set, and since that user cannot read any other company data set, the number of users must be at least the number of company data sets. This is most likely a much larger number of required users.

The breach of security that Brewer and Nash are trying to prevent is the following:

1. User-A has access to data on Oil Company-A and Bank-A.
2. User-B has access to data on Oil Company-B and Bank-A.
3. User-A writes information on Oil Company-A to a file in the data set for Bank-A.
4. User-B reads that file and now holds information on two oil companies, in violation of the policy (and United Kingdom laws).

The problem is that, although we can trust a financial analyst not to write any information on Oil Company-A to the data set for Bank-A, we cannot necessarily trust a computer program in this way. The program has no understanding of the data, and furthermore, it could have been subverted by User-B to write information without the knowledge of User-A.

The Chinese Wall policy is nonretractive since each access, when allowed, affects only the permissions that are grantable and ungranted. Note that write access is granted only if no objects have been read in any other unsanitized data set. Because of the extreme constraints on writing, the policy is nearly transitive with only one type of intransitive behavior. Assume A can write to the sanitized data set (and therefore to no other data set), and B can write to an unsanitized data set D. Since every user can read the sanitized data set, B can read files written by A. However, A cannot write to D. This policy can be supported by DTOS.

6.2.7.2 Static Chinese Wall

In Section 6.2.4.2 we modified the static Pure Clark-Wilson policy ($C > 2$) to obtain a dynamic version which we called Piecemeal Clark-Wilson ($C = 2$). The restriction of C to 2 allows us to implement the piecemeal version on DTOS. The fact that this could be done suggests a relationship between history-sensitivity and implementation. To explore this relationship further we now attempt to apply the reverse process to Chinese Wall to obtain a static version with $C > 2$.

Any Chinese Wall system progresses toward a static state. If it reaches a point where every user either has write access to some data set or has read access to one data set from each conflict of interest class, then no more policy changes can occur unless a new user is added or a new conflict of interest class is created. Along the way to this static state the policy changes whenever any user is granted permission to write a file or is granted permission for the first time to read a given data set. Our static version collapses this progression by requiring the administrator to specify at the time when a new user is added to the system all of the data sets to which the user is given read and/or write access.

This policy is static in that the only changes are due to explicit requests from the administration. $C > 2$ since we assume the administrator submits the entire set of allowed accesses all at once. The policy is nonretractive since it is static. It is intransitive for the same reasons as Pure Chinese Wall. Admittedly, this policy is less convenient for everyone involved, but it does allow the same proof of compliance with United Kingdom law as is allowed by the pure version. Since the pure version can be supported, this variant is only of theoretical interest.

6.3 Classification of the DTOS Kernel

In this section we consider DTOS with respect to its ability to support policies with the characteristics we have been discussing. We both collect in one place the earlier conclusions regarding policy support and draw some more general conclusions. In supporting security policies DTOS does have several limitations, and we discuss a number of them in this section. We should point out that these are limitations of the DTOS microkernel and the prototype Security Server, not of the general idea of separating policy decisions from policy enforcement. These limitations could typically be skirted by developing a new manager and Security Server that interacted to enforce a given policy.

The first limitation we discuss is that DTOS makes decisions based upon a *pair* of security identifiers (SIDs), one for the subject (active entity) and another for the object (passive entity). Thus, $C = 2$. While this works well for most policies, there are certain cases where the policy is most naturally thought of in terms of more than two SIDs ($C > 2$). For example, in the Clark-Wilson integrity policy, access is typically defined in terms of access triples of the form $(UserID, TP, (CDI_1, CDI_2, \ldots, CDI_n))$. Since we would most likely consider each CDI_i to be an object with its own unique SID, these access triples really include an arbitrary number of SIDs. Note that one could develop a file server and a Security Server that could send lists of SIDs back and forth, so this is not a general limitation of the architecture. It applies only to the DTOS microkernel and the prototype Security Server. Furthermore, DTOS does not support parametrized policies well. The only input information for a policy decision in addition to the SIDs is the requested permission. This could also be resolved by implementing more extensive detailed communication between the manager and Security Server.

A second limitation relates to the ability of DTOS to retract permissions. DTOS has improved in this regard over the course of its development and maintenance. However, it still does not support retraction entirely. In DTOS, the results of permission requests are stored in a cache to improve performance of the system. To allow for the removal of permissions the cache may be flushed. Furthermore, individual permissions stored in the cache may be marked as non-cachable (i.e., they may be used only once). In certain cases granted permissions may migrate out of the cache and into the microkernel. In early versions of DTOS the migrated permissions were not removed when a cache flush was performed. This placed a very serious limitation on retractive policies. When a page fault occurs in DTOS the cache and Security Server are consulted to calculate the permissions of the task to the page. The result of this computation is stored in the page table so that permissions to the page may be efficiently checked on future page accesses. In early releases, a subsequent cache flush had no immediate effect on the page table. The permissions $Have\_read$, $Have\_write$ and $Have\_execute$ had effectively migrated into the microkernel and were not retractable. This migration problem was solved by modifying the microkernel so that when permissions are flushed from the cache they are also flushed from the page table and are recalculated at the next access to the page. This has been implemented by a flush thread that resets the appropriate page table bits. Since the flush request may return to the Security Server *before* this thread finishes its job, there is a time delay in the retraction. With the current implementation of the flush thread, the Security Server has no way of knowing when the flush thread is done. Thus, retraction is still rather difficult to achieve in DTOS. The situation could be improved by

- having the flush thread send a notification to the Security Server when it is done cleaning the page tables, and

- running the flush thread at a very high priority to ensure that it does its job quickly.

However, even with this approach there is a time delay involved in flushing the cache — the flush request must be sent and processed — and this might make it difficult to implement a strict retraction policy. Refer to the discussion of the Consistency Theorem on page 43 for more on this.

Permissions may also migrate to other servers. When a request is made by sending a message to a server S, the access vector of the requesting task to the server's port is included in the message. S may use this access vector in any way it wishes. If it continues to use this vector over a period of time to make service access decisions regarding the client task, then a retraction of permissions will not be reflected in the actions of S. S might also be able to make requests directly to the Security Server, and the same issue of stale, migrated permissions applies in this case. To solve this migration problem servers must be notified when permissions are removed for any subject. The later releases of DTOS do provide for this notification.

DTOS also is limited in its sensitivity. For history-sensitive policies it is important to know what permissions have been used. However, if a Security Server grants a permission, it cannot tell whether the permission is actually used. By use here, we mean that the service controlled by the permission has occurred. For example, the fact that write access for a file was requested and granted does not mean the requesting process actually wrote to the file. This can partially be overcome by having the Security Server assume that any granted permission is in fact used. We call this the *use-of-permission assumption*. When using this assumption in a Security Server, it will typically be necessary to design the server so that it only grants permissions that are actually requested. Otherwise, the server might unnecessarily restrict future policy decisions making the system hard to use. For example, if a task originally requests read access but not write access, it is only given the former even if the latter is also allowed by the policy. If the task later wishes to write, it must make another request, this time for write access. This allows tasks to declare to the Security Server which of their grantable permissions they wish to exploit. We call this Security Server behavior *stinginess*. If applied to all permissions, stinginess could greatly increase the number of interactions between a task and the Security Server and thus slow the system down. To prevent this a DTOS Security Server is allowed to be stingy with some permissions and generous with others. Selective stinginess can in principle be implemented in DTOS with the use of the cache control vector to make denied, stingy permissions non-cachable. This will cause a new query to the Security Server when the denied permission is checked. However, for the migrating permissions the denied permission is checked from the page table rather than the cache and a new security server request is not generated. To fix this, it would be necessary to make the permission checking mechanisms in the page table sensitive to cachability of permissions. A typical policy need only be stingy with some subset of the permissions read, write and execute, but stinginess can in principle apply to any permission.

We must, however, be judicious in applying the use-of-permission assumption. If the sensitivity of the policy implies that the use of a particular permission reduces the set of grantable permissions, then the assumption is safe. If, on the other hand, the use of the permission can cause an ungrantable permission to become grantable, the use-of-permission assumption is dangerous and should be avoided. An implication of this is that DTOS does not support relinquishment well. There is no way for a task to notify the Security Server that it has relinquished a permission. A task would most likely relinquish access by deallocating a region of memory. There is a permission check at the start of this operation, so the Security Server is told when this operation is being attempted. However, the Security Server is never notified that this operation has actually succeeded. The Security Server could make the use-of-permission assumption and adjust the policy to allow access for the file to other processes. However, if the deallocation fails for some reason, the relinquishing task might still be able to access the file. This would violate the policy.

DTOS can support environment-sensitivity through mechanisms such as the **SSI_load_security_policy** interface and the ability to dynamically swap in a security server by setting the security server port (**host_set_special_port**). The current implementation of **SSI_load_security_policy** may not be adequate to support policies that are both history- and environment-sensitive if the history information must survive an environment change either to affect the decisions in the new environment or to be resumed later when the system returns to the original environment. However, this is only a limitation of the prototype Security Server. A different Security Server could reimplement this operation.

The DTOS prototype Security Server could in principle support discretionary policies through the **SSI_load_security_policy** interface, but this would be clumsy and difficult to use. It would be better to implement an additional interface to the Security Server for requesting changes to the discretionary policy.[18] Another option is to not put the discretionary policy in the Security Server at all but to locate it in another server (e.g., the file server).

Finally, DTOS does support intransitive policies since the policy in the Security Server may implement any general relation.

In summary, the current DTOS prototype microkernel has the following properties with respect to support of the policy characteristics:

- $C > 2$ — No support.
- **Parametric** — No support.
- **Discretionary** — Yes, but clumsily.
- **History Sensitivity** — Yes, but only in cases where the use-of-permission assumption is valid.
- **Environment Sensitivity** — Yes.
- **Relinquishment Sensitivity** — No.
- **Retractive** — Not entirely, because of the time delay in flushing the cache and the inability of the security server to determine when migrated permission have been eliminated. (Early releases failed to support retraction of read, write and execute permissions at all.)
- **Transitivity** — Supports both transitive and intransitive policies.

The fact that it is difficult to provide "yes or no" answers on whether DTOS supports these characteristics suggests that it might be worthwhile in further work to attempt a further decomposition of the characteristics. For example, perhaps history-sensitivity could be split into two classes depending on whether the use-of-permission assumption is valid.

6.4 History Sensitivity and Implementation Methods

As we have seen the sensitivity of a policy to the history of file accesses depends in part on the style of interaction between users, processes and the Security Server. Piecemeal Clark-Wilson is history-sensitive because the Security Server cannot grant access to a CDI for a TP without knowing the other CDIs to which the TP has already been granted access. This cannot be determined based upon a single policy input with $C = 2$. However, Pure Clark-Wilson receives

[18] Discretionary policies typically, in their most natural interpretation, require a 1-1 relationship between objects and security identifiers. While the DTOS microkernel does not prohibit this, it provides virtually no assistance to processes wishing to label objects uniquely. We view this as an inconvenience in using DTOS rather than a total lack of support for discretionary policies.

in a single input (with C arbitrarily large) enough information to make a policy decision without sensitivity to any prior file access.

Working in the reverse direction, we modified the Chinese Wall policy, which is typically viewed as history-sensitive, to obtain a static policy (Static Chinese Wall). In this version, an administrator must certify each individual for a particular non-conflicting group of company data sets when the individual's account is set up. While this static version is clearly less flexible than the original, it can still prevent insider information in much the same way as the Chinese Wall policy.

The difference in both cases is in the amount of information received in a single policy input. We call this *chunking*. It is worth looking at various policies and asking whether there is some level of chunking that would make the policy static. We start with the Pure Dynamic N-Person policy. For this policy the history-sensitivity is not in the CDIs which are accessed in a chunk as with Pure Clark-Wilson. The sensitivity is in the valid sequences of TP executions. The determination of whether an individual i is allowed to perform the next step in a TP sequence depends upon the history of the sequence. That is, which individuals have executed the preceding steps. To remove the sensitivity, the entire sequence must be placed in a single chunk. Thus, an entire sequence would have to be requested (or at least declared in some way) all at once. This would include a specification of who could perform each step. This is clearly harder to use than the standard policy.

The prospects for a static variant of ORCON are much worse. Individual process executions are much more intimately linked than they are in a system with the Pure Clark-Wilson policy. Every read and write operation throughout the life of the system can potentially change the policy. It is therefore likely that we could only use one chunk containing the entire life of the system. This is unworkable.

In summary, history-sensitivity is closely linked to the way in which a policy is implemented. For some policies such as Clark-Wilson the interactions between individual policy decisions are localized enough that a static implementation is feasible. For other policies this seems unlikely.

6.5 The Lattice

Figure 2 graphically shows a portion of the lattice of policies defined by the characteristics presented above. The bottom node of the lattice represents those policies that have none of the characteristics and can be supported by a manager that supports none of the characteristics. The nodes immediately dominating the bottom node represent those policies that have exactly one of the characteristics. Each node is labeled with the set of characteristics held by policies classified at that node. Managers can also be placed in the lattice with the following interpretation. A manager is placed at node N if it supports exactly the characteristics indicated by the label of N. We have placed DTOS in the lattice. See Section 6.3 for a discussion of its placement. The nodes in the lattice that indicate classes of policies that can be supported by DTOS are shaded.

It should be noted that we have taken *in*transitivity (rather then transitivity) to be a characteristic that might be supported by a manager or required by a policy. Transitivity provides an assumption about the ways in which permissions may be assigned in the system. If a system is designed with transitive policies in mind, then it is likely that this assumption will be used to gain efficiency or make it easier to define the set of allowed permissions. Transitive relations are more restricted than general relations, and this provides more information that can be used by designers. If a system is designed with intransitive policies in mind, no such assumption is made. It also seems unlikely that the design would incorporate an assumption

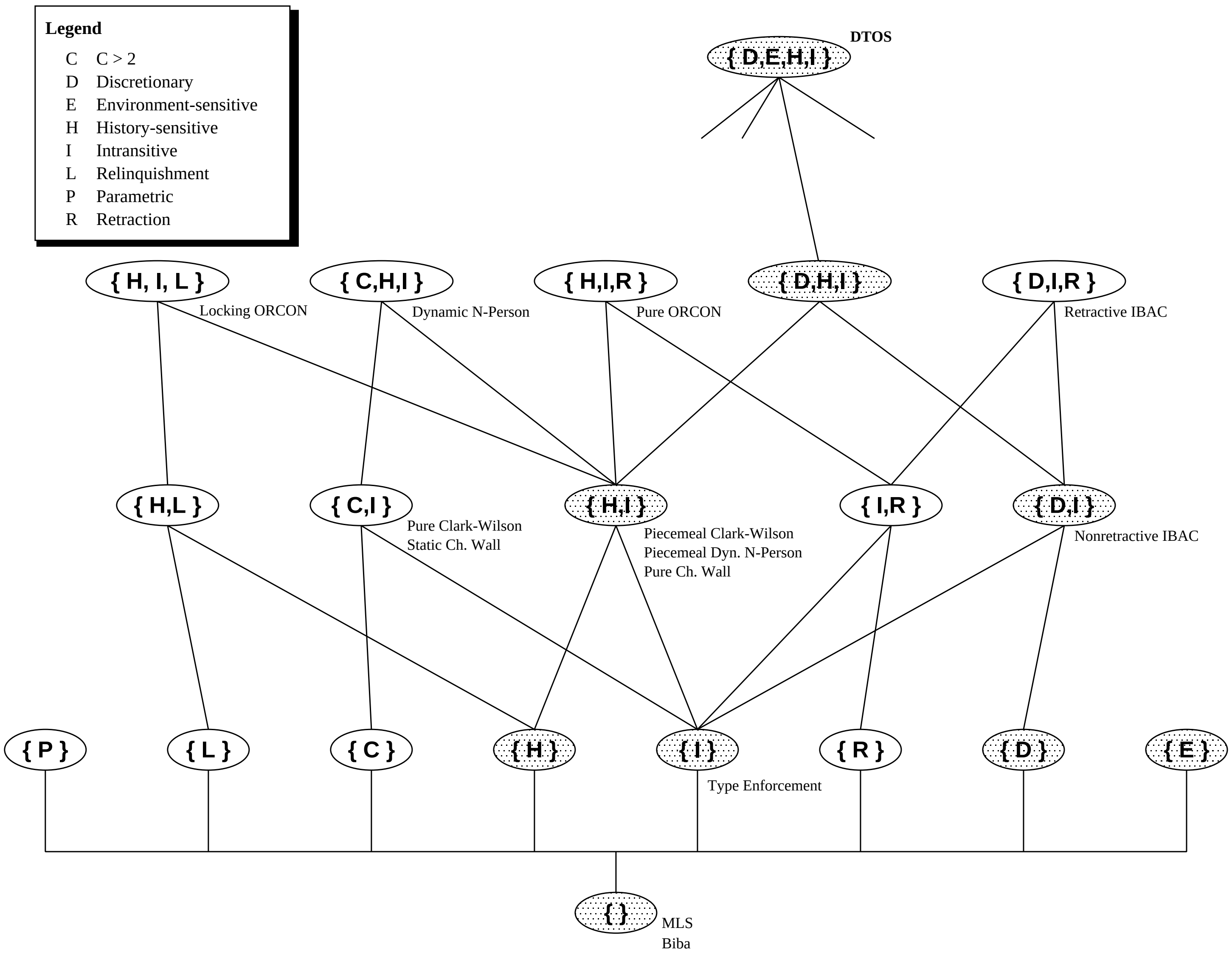

Figure 2: Partial Security Policy Lattice

that no potential policy for the system can be transitive. Thus, while any system that supports intransitive policies will most likely also support transitive ones, the converse is not true.

6.6 Formal Description of Policy Characteristics

In this section we formalize the policy characteristics using the machinery of our formal model of a manager/Security Server structured system. We will define each characteristic as a restriction of the schema $System$.

6.6.1 Input

To formalize the types of input for a security computation request that were defined informally we must interpret elements of SS_REQ as including security contexts and/or parameters.[19] We do this by defining functions $Contexts$ and $Parameters$ from a SS_REQ to sequences of elements from the generic types $CONT$ and PAR, respectively.

$$[SS\_REQ, CONT, PAR]$$
$$Contexts : SS\_REQ \longrightarrow \mathrm{seq}\ CONT$$
$$Parameters : SS\_REQ \longrightarrow \mathrm{seq}\ PAR$$

We can now formally define what it means for a system to be $C > 2$ or parametric.

$$C > 2[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS, CONT, PAR]$$
$$System[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS]$$
$$\exists\, ss\_req : SS\_REQ$$
$$\bullet\ \#(Contexts[SS\_REQ, CONT, PAR](ss\_req)) > 2$$

$$Parametric[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS, CONT, PAR]$$
$$System[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS]$$
$$\exists\, ss\_req : SS\_REQ$$
$$\bullet\ \#(Parameters[SS\_REQ, CONT, PAR](ss\_req)) > 0$$

6.6.2 Sensitivity

In this section we apply the concept of sensitivity to an entire system rather than just a security policy. We will say that a system has a sensitivity if the security policy of its Security Server has that sensitivity.

In defining sensitivities we will need to examine the security policy in effect after a sequence of operations is performed. Thus, we define a function $Final\_policies(init, op\_seq)$ which returns the set of possible policies (allowing nondeterminism) of the Security Server after the execution of an action sequence starting in some initial state.

[19] Note that we do not require that all systems have contexts or parameters in a SS_REQ.

$$
\begin{array}{l}
[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
Final\_policies : (SystemState[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\qquad \times\ OP\_SEQ[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS]) \\
\quad \longrightarrow \mathbb{P}(\mathbb{P}\ SS\_REQ) \\
\hline
\forall\ init : SystemState[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS]; \\
\quad op\_seq : OP\_SEQ[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\bullet\ Final\_policies(init, op\_seq) \\
\quad = \{\ state : Execute(\{init\}, op\_seq) \\
\qquad \bullet\ state.policy\_allows\ \}
\end{array}
$$

Before examining particular sensitivities, it is easy to define those systems that have a dynamic policy. They are simply those in which there is a valid behavior during which the policy changes.

$$
\begin{array}{l}
Dynamic[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
System[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
\exists\ B : \mathrm{seq}\ system\_ops;\ init : system\_init;\ i : \mathbb{N} \\
\bullet\ B \in Valid\_op\_seqs(init) \\
\wedge\ Final\_policies(init, B) \neq \{init.policy\_allows\}
\end{array}
$$

6.6.2.1 History Sensitivity

We first define sensitivity to the processing of a request and then use this to define history sensitivity in general. A system is sensitive to the processing of a request req if there exists a valid operation sequence B of the system in which req is executed at step i such that, for every valid subsequence S of the sequence $B(1), B(2), \ldots, B(i-1), B(i+1), \ldots B(n)$ (where n is the length of B), the policy after executing operation sequence B from some reachable state $init$ differs from the policy after executing S from $init$.

We define sensitivity in terms of operation sequences rather than individual steps since there might be a delay between the execution of the request and the change to the policy. Note that we do not require that all operation sequences containing req produce a different policy when req is removed. There might be some operation sequences in which req has no effect or in which its effect is cancelled by the effects of other requests in the sequence to which the system is also sensitive. We consider subsequences of B in which more than step i has been removed since the sequence obtained by removing only step i from B might not be a valid operation sequence of the system. *All* such subsequences must result in a different final policy since otherwise the system might actually be sensitive to some other request req_1 that also occurs in B (and not to req). In that case we could merely choose to look at a subsequence in which req_1 has been removed, and this subsequence could produce a different policy.

Now we formally define sensitivity to a request.[20]

[20] The Z operation $U \upharpoonright s$ returns a sequence that contains the elements of s with an index in the set U in the order in which they occur in s. In this case we are using it to remove element i from the sequence B.

$$
\begin{array}{|l}
\hline
\textit{SensitiveToRequest}[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\textit{System}[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
req : M\_REQ \\
\hline
(\exists\, B : \mathrm{seq}\, system\_ops;\ i : \mathbb{N}; \\
\quad init : SystemState[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\bullet\ i \in \mathrm{dom}\, B \\
\land\ init \in Reachable(\theta System) \\
\land\ B \in Valid\_op\_seqs(init) \\
\land\ B(i) \subseteq \{SystemStep[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\qquad\qquad \mid MgrProcessRequest[M\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\qquad\qquad\qquad \land\ active\_request(req\_num?) = req\} \\
\land\ (\forall\, S : \mathrm{seq}\, system\_ops \\
\qquad \mid S \in Sub\_seqs(\{j : \mathbb{N} \mid j \neq i\} \upharpoonright B) \\
\qquad\qquad \land\ S \in Valid\_op\_seqs(init) \\
\qquad \bullet\ Final\_policies(init, B) \neq Final\_policies(init, S))) \\
\hline
\end{array}
$$

A more restrictive definition of sensitivity would require

$$\mathsf{disjoint}\ \langle Final\_policies(init, B), Final\_policies(init, S)\rangle.$$

For deterministic operation sequences the two definitions are equivalent. For nondeterministic sequences the disjoint set definition requires that the policies be different whereas the set inequality definition only requires that it be possible for the policies to be different. We have chosen the set inequality definition since it seems more consistent with our decision that the policy need not change in response to all executions of req, only to some execution.

Since this definition is perhaps not very intuitive, we give an example of how it might be applied to a particular implementation of history sensitivity where a “request-succeeded” message is sent to the Security Server when the processing of a request is completed, and when the Security Server receives this message it processes it by changing the policy. Note that we will typically want to choose the operation sequence B to be as short as possible. In this example, we could take it to be

1. manager processes the request R and sends message
2. Security Server receives message
3. Security Server processes the message thereby changing the policy.

We assume that each of these operations is deterministic.

We also get to choose any reachable state $init$ in which B can be validly executed. In this case we select a state with the following properties:

- nothing is in the Security Server request queue,
- the Security Server is idle, and
- the policy contains a permission that should be removed in response to the execution of the request R.

We must prove that this state is reachable. We assume that all three of these properties are true in $system\_init$, so this state is trivially reachable.

Now we consider all valid subsequences of B that do not contain the first operation $B(1)$. Since the Security Server request queue must be empty at the point when $B(2)$ is executed, no valid subsequence begins with the Security Server receiving a message. Similarly, since no messages are received by the Security Server and the Security Server is initially idle, the Security Server cannot perform $B(3)$. Thus, the only subsequence of B that we need to consider is the empty sequence. Since the empty sequence does not change the policy and since the initial state was chosen so that $B(3)$ does change the policy, we have $Final\_policies(init, B) \neq Final\_policies(init, \langle\rangle))$ and the system is sensitive to R.

We can now define a history sensitive system. It is one that is sensitive to some request.

$$HistorySensitive[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS]$$
$$\widehat{=}\ SensitiveToRequest[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \setminus (req)$$

There are some weaknesses in this formalization of history sensitivity. One is that it is too implementation dependent to adequately describe when a *policy* (as opposed to a particular implementation of the policy in a Security Server) is history sensitive. This means that if we are to use our formal definition as a guide to the placement of a policy in the policy lattice, we must apply it to a particular implementation of the policy, not the policy itself. For example, consider the Clark-Wilson policy, and assume that there is a process P and CDIs C_1 and C_2. Furthermore, write access is initially grantable to P for both CDIs, but if P writes to either CDI, then write access becomes ungrantable to P for the other CDI. If we implement this sensitivity by sending a "request succeeded" message from the manager to the Security Server when a CDI is opened for writing and having the Security Server change its policy when the message is received, then Clark-Wilson is history-sensitive according to the formal definition. However, if we implement the sensitivity by having the Security Server change its policy when it grants write permission to a CDI, then Clark-Wilson is *not* history-sensitive according to the formal definition. The processing of a manager request has no effect on the policy.

We could claim that the second implementation of Clark-Wilson is sensitive not to the processing of manager requests, but to the sending of a permission request from the manager to the Security Server. This suggests a broader formalization of history sensitivity as "sensitivity to any manager transition". While this would make both the above implementations of Clark-Wilson history-sensitive, it introduces a different shortcoming. For policies where the performance of a manager request causes permissions to become grantable, it is important to change the policy only when the request has been successfully performed. This policy would be history-sensitive under both definitions, but only with the former definition could we be sure that an implementation was faithful to the policy. A system capable of implementing sensitivity to any manager transition and incapable of implementing sensitivity specifically to processing of a request would be classified as supporting history-sensitivity but would not be able to support this history-sensitive policy.

This leads to a general problem in defining a policy lattice. A policy is classified in the lattice as history-sensitive if it (or its Security Server implementation) reflects *any* aspect of history-sensitivity. If we apply the same existentially quantified definition of classification in the lattice to a manager such as the DTOS microkernel, then even the placement of a policy and a manager in the same node of the lattice does not imply that the manager can support the history-sensitivity required by the policy. One possible solution is to use a universally quantified definition of classification for managers. For example, a manager is classified in the lattice as history-sensitive if it supports all aspects of history-sensitivity. In combination with this we would probably wish to define useful subclasses of history sensitivity. Then, we could classify a manager as, for example, supporting all aspects of sensitivity to Security Server requests but not supporting sensitivity to the processing of manager requests.

This weakness seems less significant if we recall the discussion of Sections 6.2.4.2 and 6.4. As explained there, the entire static/dynamic distinction is heavily influenced by the implementation.

A second weakness in this formalization is that it does not capture our expectation that it will be a necessary property of a Security Server for most history-sensitive policies that the Security Server not make any policy decisions between the occurrence of an event to which the policy is sensitive and the corresponding update to the policy. Otherwise, the Security Server is making decisions based upon out-of-date history information.

6.6.2.2 Environment Sensitivity

Although environment sensitivity makes intuitive sense, it is difficult to formalize as something distinct from history sensitivity within the framework that has been defined. As described earlier environment sensitivity includes sensitivity to things such as the time of day and the operating mode of the system. In most systems, manager processing will be necessary in order to set the operating environment. This means that the change of policy due to a change of mode will be a history sensitivity to any requests used to change the mode. Similarly, to determine the time of day a Security Server will most likely have to request the time from the manager. Any resulting change of policy will represent a history sensitivity to the request for the system time. To formalize environment sensitivity it would be necessary to have a framework that is more tightly constrained than the current one.

6.6.2.3 Discretionary Sensitivity

Similar comments apply to discretionary sensitivity. The key notion here is that each user has the ability to define parts of the policy.

6.6.2.4 Relinquishment Sensitivity

In terms of the Security Server, relinquishment sensitivity is not really any different from history sensitivity. The manager processes a request, and in response, the policy changes. In terms of the manager, the processing of the request to which the policy is sensitive must potentially reduce $retained\_rel$. It is, however, possible that no change occurs in $retained\_rel$ because the relevant retentions are not actually present. For example, they may have been flushed to make room for other retentions. Thus, given the current framework, relinquishment is formally very difficult to distinguish from history-sensitivity.

6.6.3 Retraction

We formalize retractive systems in two parts. The first models those systems where retraction is potentially necessary. This is essentially a Security Server property. It is satisfied if it is possible for the Security Server to grant (either voluntarily or in response to a request) an SS_REQ which is later ungrantable.

$$
\begin{array}{l}
\textit{NeedsRetraction}[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
System[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
\exists\, B : \mathrm{seq}\ system\_ops;\ ss\_req : SS\_REQ; \\
\quad init, s : SystemState[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\bullet\ init \in Reachable(\theta System) \\
\qquad \land\ s \in Execute(\{init\}, B) \\
\qquad \land\ ss\_req \in \bigcup(grants(\!| \mathrm{dom}\ init.pending\_responses |\!) \\
\qquad\qquad\qquad \cup\ voluntarily\_grants(\!| \mathrm{dom}\ init.pending\_requests |\!)) \\
\qquad \land\ ss\_req \notin s.policy\_allows
\end{array}
$$

The second part models systems that support retraction. This property is satisfied if in all states the set of SS_REQ retained for any M_REQ is a subset of those allowed by the current Security Server policy. Note that this property is trivially satisfied by a system that never retains any SS_REQs.

$$
\begin{array}{l}
SupportsRetraction[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
System[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
\mathrm{ran}\ retained\_rel \subseteq policy\_allows
\end{array}
$$

A system is retractive if retraction is potentially necessary and it is supported.

$$
\begin{array}{l}
Retractive[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\quad \widehat{=}\ NeedsRetraction[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS] \\
\quad \land\ SupportsRetraction[M\_DATA, SS\_DATA, M\_REQ, SS\_REQ, RESP, ANS]
\end{array}
$$

6.6.4 Transitivity

Transitivity really only applies to a particular class of policies that are defined in terms of a trinary relation between a set of domains (collections of active entities) DOM, a set of types (collections of passive entities) $TYPE$ and a set of permissions $PERM$ allowing elements of a domain to perform a given operation on elements of a type. In order to formalize this dimension we define the following mappings on SS_REQ:

$$
\begin{array}{l}
[SS\_REQ, DOM, TYPE, PERM] \\
\hline
Domain : SS\_REQ \longrightarrow DOM \\
Type : SS\_REQ \longrightarrow TYPE \\
Perm : SS\_REQ \longrightarrow PERM \\
Perm\_triple : SS\_REQ \longrightarrow DOM \times TYPE \times PERM \\
\hline
\forall\, ss\_req : SS\_REQ \\
\bullet\ Perm\_triple(ss\_req) = (Domain(ss\_req), Type(ss\_req), Perm(ss\_req))
\end{array}
$$

We divide the set of permissions into two not necessarily disjoint classes, $Observe\_perms$ and $Alter\_perms$. The former contains the permissions that allow an element of a domain to observe information from an element of a type. The latter contains the permission that allow an element of a domain to alter information in an element of a type. Every permission must be in at least one of these sets.

$$
\begin{array}{l}
[PERM] \\
\hline
Observe\_perms, Alter\_perms : \mathbb{P}\ PERM \\
\hline
Observe\_perms \cup Alter\_perms = PERM
\end{array}
$$

We now define two types of transitivity, *alter-transitivity* and *observe-transitivity*. An alter-transitive system is one in which the following property holds. If domain d_1 has permission to alter elements of type t_1 and domain d_2 has permission both to observe elements of type t_1 and alter elements of type t_2, then d_1 has permission to alter elements of type t_2. Note that we do not require that d_1 have permission to observe elements of t_1 nor that d_2 have permission to observe elements of t_2.

$$
\begin{array}{l}
\underline{\mathit{AlterTrans}[MD, SSD, M\_REQ, SS\_REQ, RESP, ANS, DOM, TYPE, PERM]} \\
\mathit{System}[MD, SSD, M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
\textbf{let } \mathit{triple} == \mathit{Perm\_triple}[SS\_REQ, DOM, TYPE, PERM] \\
\bullet \forall d_1, d_2 : DOM; t_1, t_2 : TYPE \\
\quad \mid (\exists ss\_req_1, ss\_req_2, ss\_req_3 : policy\_allows; \\
\qquad\qquad alter_1, alter_2 : Alter\_perms; obs_1 : Observe\_perms \\
\qquad\quad \bullet triple(ss\_req_1) = (d_1, t_1, alter_1) \\
\qquad\qquad \wedge triple(ss\_req_2) = (d_2, t_1, obs_1) \\
\qquad\qquad \wedge triple(ss\_req_3) = (d_2, t_2, alter_2)) \\
\quad \bullet (\exists ss\_req_4 : policy\_allows; alter_3 : Alter\_perms \\
\qquad\quad \bullet triple(ss\_req_4) = (d_1, t_2, alter_3))
\end{array}
$$

An observe-transitive system is similarly defined. If domain d_1 has permission both to observe elements of type t_1 and alter elements of type t_2 and domain d_2 has permission to observe elements of type t_2, then d_2 has permission to observe elements of type t_1.

$$
\begin{array}{l}
\underline{\mathit{ObserveTrans}[MD, SSD, M\_REQ, SS\_REQ, RESP, ANS, DOM, TYPE, PERM]} \\
\mathit{System}[MD, SSD, M\_REQ, SS\_REQ, RESP, ANS] \\
\hline
\textbf{let } \mathit{triple} == \mathit{Perm\_triple}[SS\_REQ, DOM, TYPE, PERM] \\
\bullet \forall d_1, d_2 : DOM; t_1, t_2 : TYPE \\
\quad \mid (\exists ss\_req_1, ss\_req_2, ss\_req_3 : policy\_allows; \\
\qquad\qquad alter_1 : Alter\_perms; obs_1, obs_2 : Observe\_perms \\
\qquad\quad \bullet triple(ss\_req_1) = (d_1, t_1, obs_1) \\
\qquad\qquad \wedge triple(ss\_req_2) = (d_1, t_2, alter_1) \\
\qquad\qquad \wedge triple(ss\_req_3) = (d_2, t_2, obs_2)) \\
\quad \bullet (\exists ss\_req_4 : policy\_allows; obs_3 : Alter\_perms \\
\qquad\quad \bullet triple(ss\_req_4) = (d_2, t_1, obs_3))
\end{array}
$$

A transitive system is one that is both alter-transitive and observe-transitive.

$$
\begin{array}{l}
\mathit{Transitive}[MD, SSD, M\_REQ, SS\_REQ, RESP, ANS, DOM, TYPE, PERM] \\
\quad \widehat{=}\ \mathit{AlterTrans}[MD, SSD, M\_REQ, SS\_REQ, RESP, \\
\qquad\qquad ANS, DOM, TYPE, PERM] \\
\qquad \wedge\ \mathit{ObserveTrans}[MD, SSD, M\_REQ, SS\_REQ, RESP, \\
\qquad\qquad\quad ANS, DOM, TYPE, PERM]
\end{array}
$$

Section 7 DTOS Microkernel

In this section we specify the DTOS microkernel as an instantiation of the generic manager described in Section 5. The DTOS system is object-based in that its overall structure is a collection of active components (a *microkernel* for each system node and a set of *tasks*) that communicate by sending messages through ports or by reading and writing shared memory. Each of these components may be considered to be a manager that controls a group of objects, performing operations to effect changes and communicating values. The ports and the memory objects through which the communication occurs, and the creation, destruction, and scheduling of tasks, are managed by the microkernels.

The Mach microkernel design on which DTOS is based (and also several other microkernel designs) provides some security through capabilities. Unfortunately, these capabilities are rather limited in that they only provide control over sending to and receiving from ports. Potentially, they could be used in the implementation of various security policies. However, the security functions added in DTOS also control IPC operations (as well as operations on many other objects); indeed, possession of a send capability for a port is not sufficient to allow a task to send a message to that port. We therefore will not consider the Mach capabilities further in the description of the DTOS microkernel as a manager.

7.1 Instantiation of Generic Types

We begin by defining types for each of the generic parameters of a manager, M_DATA, M_REQ, SS_REQ, $RESP$ and ANS. We then specify additional constraints on the components of $MgrState$ and the manager operations to restrict the behavior of the manager to that of the microkernel.

There are three kinds of computation requests that may be generated by the DTOS Microkernel: permission requests, information requests and notifications. This specification will focus primarily upon permission requests. A permission request to the Security Server consists of a $PermReq$ which contains a requested permission $perm$, a subject security identifier ssi and an object security identifier osi.

$$[PERMISSION, SSI, OSI]$$

Among the defined permissions in DTOS are the following:

$$\begin{array}{|l} Cross\_context\_create, Create\_task, Have\_read, Have\_write, Have\_execute, \\ \quad Can\_send, Can\_receive, Make\_sid : PERMISSION \end{array}$$

We mention these permissions because they are needed in the specifications of the example Security Servers included later in this report. A complete list of permissions can be found in the DTOS Formal Security Policy Model [27].

$$\begin{array}{|l} \hline PermReq \\ \hline perm : PERMISSION \\ ssi : SSI \\ osi : OSI \\ \hline \end{array}$$

We will model information requests and notifications simply as elements on the types D_INFO_REQ and D_NOTIF_REQ.

$$[\,D\_INFO\_REQ, D\_NOTIF\_REQ\,]$$

A DTOS Security Server computation request is modeled by the type D_SS_REQ.

$$D\_SS\_REQ ::= Perm\_req\langle\langle PermReq\rangle\rangle \mid Info\_req\langle\langle D\_INFO\_REQ\rangle\rangle \\ \mid Notif\_req\langle\langle D\_NOTIF\_REQ\rangle\rangle$$

The Security Server responds to permission and information requests with one or more answers of type D_ANS.

$$[\,D\_ANS\,]$$

For permission requests the Security Server responds with a ruling containing the following information:

- $access\_vector$ — a set of permission that are granted,
- $control\_vector$ — a set of permissions (granted or denied) that may be retained in the cache,
- $notification\_vector$ — a set of permissions (granted or denied) for which the Security Server wishes to be notified when the permission is checked,
- $expiration\_value$ — a time at which the vectors are no longer considered valid, and
- $time\_stamp$ — the time at which the ruling was sent.

Editorial Note:
The implementation does not include a time stamp. This component has been introduced as an abstraction device to model the behavior of the kernel when it receives a flush request. See Section 7.3 for more information on flush requests.

To model the association between the permission request and the ruling from the security server, we include the permission request as part of the ruling.

$$ACCESS\_VECTOR == \mathbb{P}\ PERMISSION$$

$DtosRuling$

$$\begin{array}{l} ss\_req : PermReq \\ access\_vector : ACCESS\_VECTOR \\ control\_vector : ACCESS\_VECTOR \\ notification\_vector : ACCESS\_VECTOR \\ expiration\_value : \mathbb{N} \\ time\_stamp : \mathbb{N} \end{array}$$

We model returned information with the type D_INFO.

$$[\,D\_INFO\,]$$

A Security Server response is modeled by the type D_RESP

$$D\_RESP ::= Ruling\_resp\langle\langle DtosRuling\rangle\rangle \mid Info\_resp\langle\langle D\_INFO\rangle\rangle$$

A request to the microkernel includes an operation op to be performed and a sequence of parameters for that operation. The $client$ of the request is a thread. In addition to the values of $KERNEL\_OP$ defined in the FSPM we include such operations as reading, writing and executing a word from a memory page. This is necessary because, at the microkernel level, these are the operations that many security policies must control, and the ability to perform these operations in DTOS is based upon the protection bits associated with a page rather than with an explicit check to the security policy. For a static policy this distinction is not important, but for dynamic policies that must retract permissions it is crucial.

$$[\, THREAD, TASK, KERNEL\_OP, KERNEL\_PARAM \,]$$

The following operations are needed later in the descriptions of the example Security Servers:

$$\begin{array}{l} Avc\_flush\_cache\_id, Read\_page, Write\_page, Execute\_page, \\ \quad\quad Vm\_read\_id, Vm\_write\_id, Task\_create\_id, Task\_create\_secure\_id, \\ \quad\quad Task\_change\_sid\_id \\ \quad : KERNEL\_OP \end{array}$$

For a complete list of operations, refer to the document DTOS Mach Kernel Interfaces [31].

$MkRequest$

$$\begin{array}{l} client : THREAD \\ client\_task : TASK \\ op : KERNEL\_OP \\ params : \mathrm{seq}\, KERNEL\_PARAM \end{array}$$

The microkernel maintains data that is relevant to security decisions in the $cache$ of access vectors. The cache is a set of $CacheEntry$ values. A $CacheEntry$ value has the following components:

- ssi — the SSI associated with the cached access computation
- osi — the OSI associated with the cached access computation
- $access\_vector$ — the set of permissions the Security Server indicated for ssi to osi
- $control\_vector$ — the cache control vector the Security Server indicated for ssi to osi
- $notification\_vector$ — the notification vector the Security Server indicated for ssi to osi
- $expiration\_value$ — the time at which the Security Server indicated $access\_vector$ needs to be recomputed

$CacheEntry$

$$\begin{array}{l} ssi : SSI \\ osi : OSI \\ access\_vector : ACCESS\_VECTOR \\ control\_vector : ACCESS\_VECTOR \\ notification\_vector : ACCESS\_VECTOR \\ expiration\_value : \mathbb{N} \end{array}$$

The cache contains at most one non-expired $CacheEntry$ for each (ssi, osi) pair. The value $host\_time$ is the current time that is used in determining expiration of cache entries, and $time\_stamp$ is the time at which the most recently processed flush request was sent.

Editorial Note:
The implementation does not include a time stamp. This component has been introduced as an abstraction device to model the behavior of the kernel when it receives a flush request. See Section 7.3 for more information on flush requests.

An instance pr of $PermReq$ is granted by the cache if there exists a non-expired cache entry $entry$ with ssi and osi matching pr such that $pr.perm$ is in both the access vector and control vector of $entry$. The expression $granted\_by\_cache$ denotes the set of $PermReq$ values that are granted by the cache. An instance pr of $PermReq$ is denied by the cache if there exists a non-expired cache entry $entry$ with ssi and osi matching pr such that $pr.perm$ is in the control vector of $entry$ and not in the access vector. The expression $denied\_by\_cache$ denotes the set of $PermReq$ values that are denied by the cache. Note that the permissions stored in the cache may in principle be consulted with respect to any request req. It is the kernel's control policy mgr_policy that determines which permissions are actually consulted.

MkCache

$$
\begin{array}{l}
cache : \mathbb{P}\ CacheEntry \\
host\_time : \mathbb{N} \\
time\_stamp : \mathbb{N} \\
granted\_by\_cache : \mathbb{P}\ PermReq \\
denied\_by\_cache : \mathbb{P}\ PermReq \\
\hline
(\forall e_1, e_2 : CacheEntry \\
\mid \{ e_1, e_2 \} \subseteq cache \\
\quad \wedge\ e_1.ssi = e_2.ssi \\
\quad \wedge\ e_1.osi = e_2.osi \\
\quad \wedge\ e_1.expiration\_value \geq host\_time \\
\quad \wedge\ e_2.expiration\_value \geq host\_time \\
\bullet\ e_1 = e_2) \\
\\
granted\_by\_cache \\
\quad = \{pr : PermReq \\
\qquad \mid (\exists\ entry : cache \\
\qquad\quad \bullet\ entry.expiration\_value \geq host\_time \\
\qquad\qquad \wedge\ pr.ssi = entry.ssi \\
\qquad\qquad \wedge\ pr.osi = entry.osi \\
\qquad\qquad \wedge\ pr.perm \in entry.access\_vector \cap entry.control\_vector)\} \\
denied\_by\_cache \\
\quad = \{pr : PermReq \\
\qquad \mid (\exists\ entry : cache \\
\qquad\quad \bullet\ entry.expiration\_value \geq host\_time \\
\qquad\qquad \wedge\ pr.ssi = entry.ssi \\
\qquad\qquad \wedge\ pr.osi = entry.osi \\
\qquad\qquad \wedge\ pr.perm \in entry.control\_vector \setminus entry.access\_vector)\}
\end{array}
$$

In the DTOS kernel certain permissions may *migrate* beyond the cache. By this we mean that these permissions are checked at some point and then stored as data within the system state. Certain later operations are based upon this data rather than on the contents of the cache or new responses of the Security Server. An example of this is in the setting of the page protection bits. We represent this with the functions $migrated\_grantings$ and $migrated\_denials$. The expression

$migrated\_grantings(req)$ denotes the set of granted permissions that have migrated beyond the cache for future use with the request req. The expression $migrated\_denials(req)$ denotes the set of denied permissions that have migrated beyond the cache for future use with the request req. Unlike the permissions in the cache, migrated permissions apply to only certain requests. In terms of our example, this reflects the fact that the protection bits are consulted only for requests for words within the appropriate page. No permission may be in both $migrated\_grantings(req)$ and $migrated\_denials(req)$ for any req.

$MkMigratedPermissions$

$$migrated\_grantings : MkRequest \longrightarrow \mathbb{P}\ PermReq$$
$$migrated\_denials : MkRequest \longrightarrow \mathbb{P}\ PermReq$$

$$\forall\ req : MkRequest$$
$$\bullet\ migrated\_grantings(req) \cap migrated\_denials(req) = \varnothing$$

Each port in the kernel is labeled with an OSI. The label is denoted by the expression $port\_osi(port)$. The kernel labels each memory object (via its pager port) and each memory region with an OSI. These labels are returned by the functions $memory\_osi$ and $region\_osi$, respectively. When a region is mapped to a memory object, the region's label is derived from the label of the object. The function $memory\_osi\_region\_osi$ denotes this relationship between object security identifiers. The OSI osi_1 equals $memory\_osi\_region\_osi(osi_2)$ when osi_1 is the OSI used to label regions that are mapped to memory objects labeled with osi_2. The kernel labels each task with an SSI denoted by $task\_ssi(task)$ used to control the actions of that task and with an OSI denoted by $task\_osi(task)$ used to control actions performed upon that task. Two tasks have the same SSI if and only if they have the same OSI. The expression $ssi\_osi(ssi)$ denotes the OSI of a task with SSI ssi. The mapping osi_ssi is the inverse of ssi_osi. Both of these functions are injections. All memory region, memory object, and task OSI's are also port OSI's.

One of the features of Mach is that it allows tasks to perform operations on other tasks that have not traditionally been provided by operating systems. For example, Mach allows tasks to access memory regions in other tasks while one of the features of traditional operating systems is the separation of address spaces. To provide finer control over task accesses, DTOS defines $task\_self\_sid$ to be a value to be used in access computations governing accesses a task makes to itself. No kernel entities are ever assigned $task\_self\_sid$ as their SID. Instead, this SID indicates to security servers that the kernel requires an access computation to be performed between a task and the task itself. One potential use of this finer control would be to contain a faulty task by preventing it from corrupting other tasks having the same SID.

We define $task\_target(task_1, task_2)$ to be the OSI of $task_2$'s self port if $task_1$ and $task_2$ are different and $task\_self\_sid$, otherwise. When $task_1$ attempts to operate on $task_2$, the kernel enforces accesses on the pair $(task\_ssi(task_1), task\_target(task_1, task_2))$. This allows separate permissions sets to be applied when a task operates on itself versus operating on another task with the same SSI.

$$[PORT, REGION, MEMORY, PAGE]$$

$$task\_self\_sid : OSI$$

MkLabels

$$
\begin{array}{|l}
task\_ssi : TASK \pfun SSI \\
task\_osi : TASK \pfun OSI \\
task\_target : TASK \times TASK \pfun OSI \\
port\_osi : PORT \pfun OSI \\
region\_osi : REGION \pfun OSI \\
memory\_osi : MEMORY \pfun OSI \\
ssi\_osi : SSI \rightarrowtail OSI \\
osi\_ssi : OSI \pinj SSI \\
memory\_osi\_region\_osi : OSI \pfun OSI \\
\hline
\mathsf{disjoint}\ \langle \mathrm{ran}\ task\_osi, \mathrm{ran}\ region\_osi, \mathrm{ran}\ memory\_osi, \{task\_self\_sid\}\rangle \\
\mathrm{ran}\ task\_osi \cup \mathrm{ran}\ region\_osi \cup \mathrm{ran}\ memory\_osi \subseteq \mathrm{ran}\ port\_osi \\
task\_self\_sid \notin \mathrm{ran}\ port\_osi \\
\mathrm{dom}\ ssi\_osi = \mathrm{ran}\ task\_ssi \\
task\_osi = task\_ssi \mathbin{;} ssi\_osi \\
\mathrm{dom}\ memory\_osi\_region\_osi = \mathrm{ran}\ memory\_osi \\
\mathrm{ran}\ memory\_osi\_region\_osi \subseteq \mathrm{ran}\ region\_osi \\
osi\_ssi = ssi\_osi^{\sim} \\
\\
\forall\, client, task : \mathrm{dom}\ task\_ssi \\
\bullet\ (client, task) \in \mathrm{dom}\ task\_target \\
\land\ task\_target(client, task) \\
\qquad = \mathbf{if}\ client = task\ \mathbf{then}\ task\_self\_sid \\
\qquad \mathbf{else}\ task\_osi(task)
\end{array}
$$

A region may be mapped to a memory.

MkAddressSpace

$$
\begin{array}{|l}
mapped\_memory : REGION \pfun MEMORY
\end{array}
$$

The data controlled by the DTOS microkernel includes the Mach state, the cached and migrated permissions and the security labels.

$[MACH\_STATE\_DATA]$

MkMachState

$$
\begin{array}{|l}
mach\_state\_data : MACH\_STATE\_DATA
\end{array}
$$

$[OTHER\_MK\_DATA]$

MkData

$$
\begin{array}{|l}
MkMachState \\
MkCache \\
MkMigratedPermissions \\
MkLabels \\
MkAddressSpace \\
other\_mk\_data : OTHER\_MK\_DATA
\end{array}
$$

7.2 DTOS State

In this section we define the DTOS microkernel state. First, we constrain the control policy of the kernel. These constraints represent only a small portion of the constraints imposed by the DTOS Formal Security Policy Model [27]. We have focused on those security server requests that are necessary for the security servers defined in the following sections of this document.

Since mgr_policy is defined as a function from a kernel request to a set of Security Server computation requests, and since the parameters of the kernel request are important in determining this relationship, we define several functions that interpret kernel request parameters returning the appropriate kernel or security objects. The function $Find\_task$ returns a task indicated by an IPC name supplied as a parameter of a request from task $client\_task$. The function $Find\_ssi$ returns the SSI indicated by a parameter. The function $Find\_region$ returns a memory region indicated by an address supplied as a parameter of a request from task $client\_task$. The region is within $client\_task$'s address space. Finally, $Find\_task\_region$ returns a memory region of a task indicated by a name parameter interpreted in the IPC space of $client\_task$ where the start address and size or the region are also indicated by parameters.

$$
\begin{array}{|l}
Find\_task : TASK \times KERNEL\_PARAM \pfun TASK \\
Find\_ssi : KERNEL\_PARAM \pfun SSI \\
Find\_region : TASK \times KERNEL\_PARAM \pfun REGION \\
Find\_task\_region : (TASK \times KERNEL\_PARAM \times KERNEL\_PARAM \\
\quad \times KERNEL\_PARAM) \pfun REGION
\end{array}
$$

The schema $MkClientNeedsPermissionAux$ is used in defining the control policy constraints. It requires for a kernel request req that $mgr\_policy(req)$ contain a Security Server request for permission the_perm from the SSI of the client task[21] to the OSI the_osi.

$$
\begin{array}{|l}
\hline
MkClientNeedsPermissionAux \\
\hline
MgrState[MkData, MkRequest, D\_SS\_REQ, D\_RESP, D\_ANS] \\
req : MkRequest \\
the\_perm : PERMISSION \\
the\_osi : OSI \\
\hline
\exists\, preq : PermReq \\
\mid Perm\_req(preq) \in mgr\_policy(req) \\
\bullet\ preq.perm = the\_perm \\
\quad \land preq.ssi = mgr\_data.task\_ssi(req.client\_task) \\
\quad \land preq.osi = the\_osi \\
\hline
\end{array}
$$

We now constrain the policy as follows:

- For any **task_create** operation the client must have $Create\_task$ permission to SID of the target task.
- For any **task_create_secure** operation the client must have $Cross\_context\_create$ permission to the SID of the new task.
- For any **task_change_sid** operation the client must have $Make\_sid$ permission to the new SID.
- For any **vm_read** operation the client must have $Have\_read$ permission to the SID of the region being read.

[21] We note that in DTOS, some requests require permission requests in which the ssi is not that of the client.

- For any **vm_write** operation the client must have *Have_write* permission to the SID of the region being written.
- For any **read_page** operation the client must have *Have_read* permission to the SID of the indicated region of its address space.
- For any **write_page** operation the client must have *Have_write* permission to the SID of the indicated region of its address space.
- For any **execute_page** operation the client must have *Have_execute* permission to the SID of the indicated region of its address space.

We note that we have been somewhat careless here regarding the domain restrictions of functions. For any given kernel state, there are requests in *MkRequest* that are for non-existent threads or tasks. Furthermore, we have not constrained the *client* to be a thread belonging to task *client_task*. It should ideally be stated that any request that is actually made to the kernel is by an existing thread operating within an existing task which is the *client_task* of the request. However, our intent is that even ill-formed requests will be in the domain of *mgr_policy*. The value of *mgr_policy* for such a request should be the set of all *SS_REQ*. Then, when a new thread is created, *mgr_policy* for requests by that thread can be shrunk to the set of permission requests appropriate for the SSI assigned to the thread. This shrinking will not disturb the monotonicity of *mgr_policy*.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{MkPolicy}} \\
MgrState[MkData, MkRequest, D\_SS\_REQ, D\_RESP, D\_ANS] \\
\hline
\forall\, req : MkRequest \\
\bullet\ (req.op = Task\_create\_id \\
\qquad \Rightarrow (\textbf{let}\ the\_osi == mgr\_data.task\_target(req.client\_task, \\
\qquad\qquad\qquad Find\_task(req.client\_task, (req.params)(1))) \\
\qquad\quad \bullet\ MkClientNeedsPermissionAux[Create\_task/the\_perm])) \\
\wedge\ (req.op = Task\_create\_secure\_id \\
\qquad \Rightarrow (\textbf{let}\ the\_osi == mgr\_data.ssi\_osi(Find\_ssi((req.params)(4))) \\
\qquad\quad \bullet\ MkClientNeedsPermissionAux[Cross\_context\_create/the\_perm])) \\
\wedge\ (req.op = Task\_change\_sid\_id \\
\qquad \Rightarrow (\textbf{let}\ the\_osi == mgr\_data.ssi\_osi(Find\_ssi((req.params)(2))) \\
\qquad\quad \bullet\ MkClientNeedsPermissionAux[Make\_sid/the\_perm])) \\
\wedge\ (req.op = Vm\_read\_id \\
\qquad \Rightarrow (\textbf{let}\ the\_osi == mgr\_data.region\_osi(Find\_task\_region(req.client\_task, \\
\qquad\qquad\qquad (req.params)(1), (req.params)(2), (req.params)(3))) \\
\qquad\quad \bullet\ MkClientNeedsPermissionAux[Have\_read/the\_perm])) \\
\wedge\ (req.op = Vm\_write\_id \\
\qquad \Rightarrow (\textbf{let}\ the\_osi == mgr\_data.region\_osi(Find\_task\_region(req.client\_task, \\
\qquad\qquad\qquad (req.params)(1), (req.params)(2), (req.params)(4))) \\
\qquad\quad \bullet\ MkClientNeedsPermissionAux[Have\_write/the\_perm])) \\
\wedge\ (req.op = Read\_page \\
\qquad \Rightarrow (\textbf{let}\ the\_osi == mgr\_data.region\_osi(Find\_region(req.client\_task, \\
\qquad\qquad\qquad (req.params)(1))) \\
\qquad\quad \bullet\ MkClientNeedsPermissionAux[Have\_read/the\_perm])) \\
\wedge\ (req.op = Write\_page \\
\qquad \Rightarrow (\textbf{let}\ the\_osi == mgr\_data.region\_osi(Find\_region(req.client\_task, \\
\qquad\qquad\qquad (req.params)(1))) \\
\qquad\quad \bullet\ MkClientNeedsPermissionAux[Have\_write/the\_perm])) \\
\wedge\ (req.op = Execute\_page \\
\qquad \Rightarrow (\textbf{let}\ the\_osi == mgr\_data.region\_osi(Find\_region(req.client\_task, \\
\qquad\qquad\qquad (req.params)(1))) \\
\qquad\quad \bullet\ MkClientNeedsPermissionAux[Have\_execute/the\_perm])) \\
\hline
\end{array}
$$

Editorial Note:
It would be better to add the FSPM concept of services in here and use it in defining *MkPolicy*. This would reduce the conceptual dependence of this document upon the FTLS.

We can now define the microkernel state as an instantiation of *MgrState*. We define *retained* and *retained_denial* in terms of the cache and the migrated permissions. The expression $retained(req)$ is taken to be the union of the permissions that have migrated for use with *req* together with the set of all permissions granted by the cache unless the cached permissions are overridden by a migrated denial for use with *req*. Similarly, the expression $retained\_denial(req)$ is taken to be the union of the denials that have migrated for use with *req* together with the set of all denials by the cache unless the cached denials are overridden by a migrated granting. Note that the migrated permissions take precedence over those in the cache. In the current version of DTOS, there are no kernel requests that volunteer permissions.[22] For any response *response* that is a permission ruling *ruling*, the expression $grants(response)$ denotes the set of *PermReq*s whose

[22] This is not true for the latest releases of DTOS in which the request **avc_cache_control** may be used not only

ssi and osi match that of $ruling.ss\_req$ and whose $perm$ is an element of $ruling.access\_vector$. The expression $denies(response)$ denotes the set of $PermReq$s whose ssi and osi match that of $ruling.ss\_req$ and whose $perm$ is not an element of $ruling.access\_vector$.

This model differs from that in the FTLS. In the absence of migrated permissions the microkernel always requests a Security Server computation if the desired permission is marked as non-cachable (i.e., not in $control\_vector$). In this case the $valid\_for$ component of a $CacheEntry$ in the FTLS is useless and is omitted. (This component is only an abstract device used in the FTLS and does not exist in the prototype at all.) The fact that a non-cachable permission may still be checked once (as part of the processing of the request that caused the query to the Security Server) is modeled here by the schemas $MkNegativeResponse$ and $MkAffirmativeResponse$ which perform a permission check based upon the $DtosRuling$ ignoring any cachability issues.

$$
\begin{array}{l}
\textbf{MkState} \\
\hline
MgrState[MkData, MkRequest, D\_SS\_REQ, D\_RESP, D\_ANS] \\
MkPolicy \\
\hline
(\forall req : MkRequest \\
\bullet Perm\_req^{\sim}(\!|retained(req)|\!) = mgr\_data.migrated\_grantings(req) \\
\quad \cup (mgr\_data.granted\_by\_cache \setminus mgr\_data.migrated\_denials(req)) \\
\land Perm\_req^{\sim}(\!|retained\_denial(req)|\!) = mgr\_data.migrated\_denials(req) \\
\quad \cup (mgr\_data.denied\_by\_cache \setminus mgr\_data.migrated\_grantings(req)) \\
\land voluntarily\_grants(req) = \varnothing) \\
\\
permission\_requests = \mathrm{ran}\ Perm\_req \\
information\_requests = \mathrm{ran}\ Info\_req \\
notifications = \mathrm{ran}\ Notif\_req \\
\forall response : \mathrm{ran}\ Ruling\_resp \\
\bullet \mathbf{let}\ ruling == Ruling\_resp^{\sim}(response) \\
\quad \bullet Perm\_req^{\sim}(\!|grants(response)|\!) \\
\qquad = \{pr : PermReq \\
\qquad\quad \mid pr.perm \in ruling.access\_vector \\
\qquad\quad \land pr.ssi = ruling.ss\_req.ssi \\
\qquad\quad \land pr.osi = ruling.ss\_req.osi\} \\
\quad \land Perm\_req^{\sim}(\!|denies(response)|\!) \\
\qquad = \{pr : PermReq \\
\qquad\quad \mid pr.perm \notin ruling.access\_vector \\
\qquad\quad \land pr.ssi = ruling.ss\_req.ssi \\
\qquad\quad \land pr.osi = ruling.ss\_req.osi\} \\
\land first(interpret\_response(response)) = Perm\_req(ruling.ss\_req) \\
\hline
\end{array}
$$

to flush permissions, but also to volunteer them. Rather than updating this report we will assume the older flush interface that could only remove permissions from the cache.

The question of volunteered permissions in DTOS has always been open to interpretation. For permission requests from the kernel the DTOS Security Server returns an access vector by sending a message to a reply port provided by the kernel in its `SSI_compute_access_vector` request. The operation identifier in this message is 100 plus the identifier for `SSI_compute_access_vector`. Since a permission check is done by the kernel when it receives this reply message, it has at least some of the properties of a kernel request, and we frequently think of it as a request. However, unlike other requests,

- it has a reply operation identifier (i.e., the third least significant digit is odd),
- it is not sent to the kernel port of a Mach entity and
- the sending thread (i.e., the Security Server) does not begin processing its own message in kernel mode.

Given these differences it is probably more accurate to think of this message as a response to a request than as volunteered permissions.

7.3 DTOS Operations

Now, we describe the operations that change the microkernel state. Many of these operations require only minor modifications from the instantiations of operations in the generic manager description. In particular we require the following of all DTOS transitions:

- the SIDs on ports,[23] memory objects and memory regions are stable, and
- the $host\_time$ and $time\_stamp$ are monotonically non-decreasing.

These requirements are captured in the schema $MkStep$.

$$
\begin{array}{|l}
\hline
\textit{MkStep} \\
MgrStep[MkData, MkRequest, D\_SS\_REQ, D\_RESP, D\_ANS] \\
\hline
mgr\_data'.host\_time \geq mgr\_data.host\_time \\
mgr\_data'.time\_stamp \geq mgr\_data.time\_stamp \\
\\
\forall p : PORT; r : REGION; m : MEMORY \\
\bullet (p \in \operatorname{dom} mgr\_data.port\_osi \cap \operatorname{dom} mgr\_data'.port\_osi \\
\qquad \Rightarrow mgr\_data'.port\_osi(p) = mgr\_data.port\_osi(p)) \\
\land (r \in \operatorname{dom} mgr\_data.region\_osi \cap \operatorname{dom} mgr\_data'.region\_osi \\
\qquad \Rightarrow mgr\_data'.region\_osi(r) = mgr\_data.region\_osi(r)) \\
\land (m \in \operatorname{dom} mgr\_data.memory\_osi \cap \operatorname{dom} mgr\_data'.memory\_osi \\
\qquad \Rightarrow mgr\_data'.memory\_osi(m) = mgr\_data.memory\_osi(m)) \\
\hline
\end{array}
$$

Furthermore, the mgr_policy is non-increasing unless a **task_change_sid** request is being processed. Schema $MkPolicyShrinkStep$ models an $MkStep$ in which the mgr_policy does not increase.

$$
\begin{array}{|l}
\hline
\textit{MkPolicyShrinkStep} \\
MkStep \\
\hline
mgr\_policy' \subseteq mgr\_policy \\
\hline
\end{array}
$$

[23] This should really not apply to task and thread ports during a **task_change_sid** request since the SIDs on these ports are changed at that time.

$$
\begin{aligned}
&MkReceiveRequest \\
&\quad \widehat{=}\ MgrReceiveRequest[MkData, MkRequest, D\_SS\_REQ, D\_RESP, D\_ANS] \\
&\qquad \land\ MkPolicyShrinkStep \\
&MkRequestComputation \\
&\quad \widehat{=}\ MgrRequestComputation[MkData, MkRequest, D\_SS\_REQ, D\_RESP, D\_ANS] \\
&\qquad \land\ MkPolicyShrinkStep \\
&MkSendNotification \\
&\quad \widehat{=}\ MgrSendNotification[MkData, MkRequest, D\_SS\_REQ, D\_RESP, D\_ANS] \\
&\qquad \land\ MkPolicyShrinkStep \\
&MkReceiveResponse \\
&\quad \widehat{=}\ MgrReceiveResponse[MkData, MkRequest, D\_SS\_REQ, D\_RESP, D\_ANS] \\
&\qquad \land\ MkPolicyShrinkStep \\
&MkDenyRequest \\
&\quad \widehat{=}\ MgrDenyRequest[MkData, MkRequest, D\_SS\_REQ, D\_RESP, D\_ANS] \\
&\qquad \land\ MkPolicyShrinkStep \\
&MkAcceptRequest \\
&\quad \widehat{=}\ MgrAcceptRequest[MkData, MkRequest, D\_SS\_REQ, D\_RESP, D\_ANS] \\
&\qquad \land\ MkPolicyShrinkStep
\end{aligned}
$$

$MkProcessRequest$

$$
\begin{aligned}
&MgrProcessRequest[MkData, MkRequest, D\_SS\_REQ, D\_RESP, D\_ANS] \\
&MkStep \\
\hline
&(req?.op \neq Task\_change\_sid\_id \\
&\quad \Rightarrow mgr\_policy' \subseteq mgr\_policy)
\end{aligned}
$$

The only significant addition to the model is a description of the changes to the microkernel's access vector cache. This cache may change in two ways:

- When a response is received from the Security Server the cache is updated to contain the new information. One or more old cache entries might also be flushed, either to make room for the new one or because they hold outdated information that is superseded by the data being added.

- The Security Server may issue a kernel request to flush some set of access vectors from the cache.

We first define a generic $MkUpdateCache$ that will be used to describe the changes to the cache that occur for both affirmative and negative rulings from the Security Server. A ruling is cached only if its time stamp is no older than the time stamp on the cache. Note that even if a ruling is not cached, it is still used for the permission check that generated the security computation request.

$$
\begin{array}{l}
\textbf{MkUpdateCache} \\
\hline
ss\_response? : \mathrm{ran}\ Ruling\_resp \\
MkPolicyShrinkStep \\
\hline
(ss\_response? \in \mathrm{ran}\ Ruling\_resp \\
\wedge\ mgr\_data.time\_stamp \leq (Ruling\_resp^{\sim}(ss\_response?)).time\_stamp \\
\wedge\ (\exists\, new\_entry : CacheEntry; \\
\qquad\qquad cleaned\_cache : \mathbb{P}\, mgr\_data.cache; \\
\qquad\qquad ruling : DtosRuling \\
\qquad \mid ss\_response? = Ruling\_resp(ruling) \\
\qquad\quad \wedge\ new\_entry.ssi = ruling.ss\_req.ssi \\
\qquad\quad \wedge\ new\_entry.osi = ruling.ss\_req.osi \\
\qquad\quad \wedge\ new\_entry.access\_vector = ruling.access\_vector \\
\qquad\quad \wedge\ new\_entry.control\_vector = ruling.control\_vector \\
\qquad\quad \wedge\ new\_entry.notification\_vector = ruling.notification\_vector \\
\qquad\quad \wedge\ new\_entry.expiration\_value = ruling.expiration\_value \\
\qquad \bullet\ mgr\_data'.cache \\
\qquad\quad = cleaned\_cache \\
\qquad\qquad \setminus \{\, entry : CacheEntry \\
\qquad\qquad\quad \mid entry.ssi = ruling.ss\_req.ssi \\
\qquad\qquad\quad \wedge\ entry.ssi = ruling.ss\_req.ssi \,\} \\
\qquad\qquad \cup \{\, new\_entry \,\})) \\
\vee\ mgr\_data'.cache \subseteq mgr\_data.cache
\end{array}
$$

Using *MkUpdateCache* we can specify *MkNegativeResponse* and *MkAffirmativeResponse* as follows:

$$
\begin{array}{l}
MkNegativeResponse \\
\quad \widehat{=}\ MgrNegativeResponse[MkData, MkRequest, D\_SS\_REQ, D\_RESP, D\_ANS] \\
\qquad \wedge\ MkUpdateCache \\
MkAffirmativeResponse \\
\quad \widehat{=}\ MgrAffirmativeResponse[MkData, MkRequest, D\_SS\_REQ, D\_RESP, D\_ANS] \\
\qquad \wedge\ MkUpdateCache
\end{array}
$$

Editorial Note:
The use of time stamps in flush requests and in the cache does not actually occur in DTOS. The kernel achieves the same effect through manipulation of low-level details of the cache lookup code and the code that sends computation requests to the Security Server. These details are at a lower level of abstraction than the current specification, so we have introduced time stamps as an abstraction of the actual implementation.

For Security Server requests to flush the cache we specify a *MkFlushCache* operation as a special case of *MkProcessRequest*.[24] There are four parameters for this request:

- the host control port (to which the request is sent)
- the source SID
- the object SID
- the time stamp indicating when the flush request was sent.

[24] The operation described here corresponds to an earlier implementation of the flush operation in DTOS. The new interface is a little more flexible and powerful than described here. In addition to flushing, the new interface also allows rulings to be volunteered.

We use the functions $Source\_sid$, $Target\_sid$ and $Find\_number$ to interpret the parameters of the request. The function $Source\_sid$ maps a parameter to an element of the free type $SOURCE\_SID$ which contains an element for each SSI plus the special element $Wildcard\_ssid$. The function $Target\_sid$ maps a parameter to an element of the free type $TARGET\_SID$ which contains an element for each OSI plus the special element $Wildcard\_tsid$. The function $Find\_number$ maps a parameter to a natural number.

$$SOURCE\_SID ::= Source\langle\langle SSI\rangle\rangle \mid Wildcard\_ssid$$
$$TARGET\_SID ::= Target\langle\langle OSI\rangle\rangle \mid Wildcard\_tsid$$

$$\begin{array}{|l}
Source_sid : KERNEL_PARAM \nrightarrow SOURCE_SID \\
Target_sid : KERNEL_PARAM \nrightarrow TARGET_SID \\
Find_number : KERNEL_PARAM \nrightarrow \mathbb{N}
\end{array}$$

$MkFlushCache$

$$\begin{array}{|l}
MkProcessRequest \\
MkPolicyShrinkStep \\
\hline
\textbf{let } req == active_request(req_num?); \\
\quad pars == (active_request(req_num?)).params \\
\bullet\ mgr_data'.time_stamp = Find_number(pars(3)) \\
\quad \land\ req.op = Avc_flush_cache_id \\
\quad \land\ pars(2) \in \mathrm{dom}\, Source_sid \\
\quad \land\ pars(3) \in \mathrm{dom}\, Target_sid \\
\quad \land\ (\exists\, source_set : \mathbb{P}\, SSI;\ target_set : \mathbb{P}\, OSI \\
\qquad \mid source_set = \textbf{if } Source_sid(pars(2)) = Wildcard_ssid \\
\qquad\quad \textbf{then } SSI \\
\qquad\quad \textbf{else } \{\, Source^{\sim}(Source_sid(pars(2)))\,\} \\
\qquad \land\ target_set = \textbf{if } Target_sid(pars(3)) = Wildcard_tsid \\
\qquad\quad \textbf{then } OSI \\
\qquad\quad \textbf{else } \{\, Target^{\sim}(Target_sid(pars(3)))\,\} \\
\qquad \bullet\ mgr_data'.cache = mgr_data.cache \\
\qquad\quad \setminus \{\, entry : CacheEntry \\
\qquad\qquad \mid entry.ssi \in source_set \\
\qquad\qquad \lor\ entry.osi \in target_set\,\})
\end{array}$$

DTOS also allows notifications from the kernel to the Security Server if the Security Server is the receiver of the audit port. The Security Server may request notifications in two ways. It may set a bit in a notification vector when it sends a ruling. Whenever the corresponding access vector bit is checked a record containing the source and target SIDs, the access vector and the permission being checked is placed in a buffer. The contents of this buffer are occasionally sent to the audit port. The second way to request notifications is to ask that the kernel send notifications of the failure of any permission check performed against the cache. The same information is buffered as above. We do not formalize either of these types of notification at this time since the current framework does not have an explicit transition for performing a permission check against retained information. Rather, the retention of answers is considered in determining $required$ and $denied\_requests$.

7.4 Component Specification

We can use the above to specify the DTOS kernel as a component.

The **guar** for the kernel allows any of the transitions described in Section 7.3. This is modeled by mk_guar.

$$\begin{aligned}
&MkGuarStep \\
&\mathrel{\widehat=} MkReceiveRequest \\
&\quad \lor MkRequestComputation \\
&\quad \lor MkSendNotification \\
&\quad \lor MkReceiveResponse \\
&\quad \lor MkNegativeResponse \\
&\quad \lor MkAffirmativeResponse \\
&\quad \lor MkDenyRequest \\
&\quad \lor MkAcceptRequest \\
&\quad \lor MkProcessRequest
\end{aligned}$$

MkGuar

$mk\_guar : \mathbb{P}\ MkGuarStep$

$mk\_guar = MkGuarStep$

Note that the operation $MkFlushCache$ is a special case of $MkProcessRequest$ and therefore need not be explicitly mentioned in the **guar**.

The kernel assumes that the assumptions of generic managers in $MgrRely$ are satisfied.

MkRely

$mk\_rely : \mathbb{P}\ \Delta\ MkState$
$MgrRely[MkData, MkRequest, D\_SS\_REQ, D\_RESP, D\_ANS]$

$mk\_rely = mgr\_rely$

The set of allowed initial states is modeled by mk_init. We require only that an initial state satisfy the constraints imposed by $MgrInit$.

Editorial Note:
Additional constraints are probably required. For example, we may need to constrain the initial state of the cache.

MkInit

$MgrInit[MkData, MkRequest, D\_SS\_REQ, D\_RESP, D\_ANS]$
$mk\_init : \mathbb{P}\ MkState$

$mk\_init \subseteq mgr\_init$

All information in $MkState$ is visible to the kernel.

MkView

$mk\_view : MkState \leftrightarrow MkState$

$\forall\ st_1, st_2 : MkState$
$\bullet\ (st_1, st_2) \in mk\_view \Leftrightarrow st_1 = st_2$

$MkComponent$
$\quad \hat{=} \; MkGuar \wedge MkRely \wedge MkInit \wedge MkView$

The following results are easy to prove.

Lemma 7.1 *mgr_policy is monotone non-increasing in the DTOS kernel if no **task_change_sid** request is ever processed.*

Proof: $MkPolicyShrinkStep$ requires $mgr\_policy' \subseteq mgr\_policy$. This handles all manager transitions except a $MkProcessRequest$ transition with $op(req?) = Task\_change\_sid\_id$, and by hypothesis, such a transition cannot occur. Also, $mk\_rely = mgr\_rely$ which requires $\Xi\, MgrInternals$. $MgrInternals$ includes mgr_policy, and this completes the proof. □

Theorem 7.2 (DTOS Consistency Theorem) *Let Sys be a system including the DTOS kernel as its manager and having the property that no SS_REQ that has been either granted or volunteered by the security server is ever subsequently removed from $policy\_allows$. Assume further that the kernel never processes a **task_change_sid** request. If the DTOS kernel processes a request $req?$ in some state s, then for each $r \in mgr\_policy(req?) \cap permission\_requests$ in s, either*

- *$r \in retained(req?)$ in the system initial state, or*
- *$r \in policy\_allows$ in state s.*

Proof: This is a straightforward application of the generic Consistency Theorem. The first hypothesis of the Consistency Theorem is supported by Lemma 7.1, and the second hypothesis is included as a hypothesis of the DTOS Consistency Theorem. The third hypothesis is trivially satisfied since $MkState$ requires that $voluntarily\_grants(req) = \varnothing$ for all req. □

Corollary 7.3 (DTOS Complete Consistency) *Let Sys be a system that satisfies the hypotheses of the DTOS Consistency Theorem and in which, for some request $req?$, $policy\_allows$ in all reachable states contains all elements from $retained(req?)$ in the initial state. In Sys, if the manager processes $req?$ in some state s, then for each $r \in mgr\_policy(req?) \cap permission\_requests$ in s, $r \in policy\_allows$ in state s.*

Section 8
MLS/TE Policy

A MultiLevel Secure (MLS) policy is defined using a lattice of levels [10]. The lattice defines a partial ordering on levels called a dominance relation. In many cases, a level consists of two parts: an element of an ordered sequence of sensitivities (i.e. unclassified, confidential, secret, top secret) and a set of compartments. In this case level a dominates level b if a's sensitivity is at least as high as b's and b's set of compartments is a subset of a's. However, a level need not have this structure. Each subject is assigned a level representing its level of trust and each object is assigned a level representing the sensitivity of the information that it contains. Each access is classified as a *read* and/or a *write*. The Bell-LaPadula version of MLS [3] allows a subject to perform a *read* access only if its level dominates that of the object, and a *write* access only if its level is dominated by that of the object.

Another form of access control policy is type enforcement [5, 22]. Allowed accesses are specified with a Domain Definition Table (DDT), which is a coarse version of the access control matrix in which objects that have equal access privileges are clustered into groups called *types* and the environments, that may represent a cluster of individuals, are called *domains*. Because of the coarseness of specification, type enforcement is frequently used to constrain the flow of information between groups of objects (the types) rather than to control access to particular objects. A typical example of its use is for a guard between a group of sensitive objects and the rest of the system. In this case, the only domain in the DDT that permits reading from the sensitive object type and writing to objects of other types is the one in which the guard executes and no other process can execute in this domain. Thus, information may only move from sensitive objects to the rest of the system by passing through a guard. This movement of information would be in violation of a strict MLS policy, but it is usually necessary in special cases to obtain a workable system. Changes to the DDT potentially impact on many objects and therefore are usually reserved for a very few trusted domains.

The MLS/TE policy is a combination of MLS and Type Enforcement. In MLS/TE, all operations must obey Type Enforcement. Furthermore, operations by normal subjects must obey MLS. A specified set of *exceptional* subjects are allowed to violate MLS. Exceptional subjects require close analysis to ensure proper functioning (e.g., to ensure that no classified information is written to an object labeled "Unclassified"). MLS/TE can be thought of as relaxation of MLS, within the bounds of Type Enforcement, that allows for the existence of subjects such as the guard discussed above (a form of downgrader) without unnecessarily complicating the task of analyzing the system. The analysis can focus on the subjects that can violate MLS.

In Section 8.1 we describe the elements of the system state that are relevant to the MLS/TE security policy, and we state the requirements placed upon these state elements to define the MLS/TE policy. Section 8.2 defines the correspondence between the concepts of the abstract, high-level definition of MLS/TE and the entities of the DTOS kernel. Sections 8.3–8.5 describe a Security Server that enforces the MLS/TE policy, and Section 8.6 composes the MLS/TE security server with the DTOS kernel and demonstrates that the high-level policy is satisfied.

8.1 Formal MLS/TE Definition

As noted in the introduction to this section, in a practical system we typically need exceptions (e.g., a downgrader) to the standard MLS restrictions on subjects and objects. This is in part a result of the level at which the MLS policy is typically stated. The major underlying concern in an MLS system is that no subject should gain access to information for which it does not possess the proper clearance. This subject/information formulation of MLS is somewhat less implementation-oriented than the typical statement of the policy in terms of subjects reading and writing objects. In addition, it recognizes that not all information in a single object necessarily has the same level. For example, a tuple in a relational database may contain information at various levels. The database system will be trusted not to divulge classified information to users who do not have the appropriate clearance. If a tuple containing both Secret and Unclassified fields is stored in a single object labeled Secret, then, to service queries from processes labeled Unclassified, the database system must be able read a Secret object and write to an Unclassified object. This means that it can either read up or write down, which is a violation of the typical formulation of MLS. However, if the database system is correctly implemented, no Secret information is being divulged — only the Unclassified fields are returned to the Unclassified process. Subjects such as this database system will be called *exceptional subjects*. To guarantee that the system is secure, the exceptional subjects must be analyzed to be sure they do not divulge information. Thus, they require much more scrutiny than the non-exceptional subjects.

While intuitively satisfying, the high-level view of MLS is very difficult to formalize. First, the concept of information is not easily mapped in any direct way to data in the system. The number 3 in the computer may denote many different pieces of information (e.g., the number of missiles at a given base) or no information at all (e.g., a number typed randomly at a keyboard). It is the information, not the number, to which we want to assign a level. Furthermore, the information content of some data may depend upon other data (from the computer or from other sources) that is already held. In fact, the information content of data is intimately connected to the beliefs of the people possessing the data. If a user obtains the data "Base X has 3 missiles" and this sentence is true and the user believes that it is true, then the user has obtained information. If the sentence is false or the user does not believe the truth of the sentence then no information is obtained. Similarly, a user may choose to believe things about unclassified data that result in the user believing correct classified information. For example, an unclassified file might contain the number 3 referring to the number of meals in a day. A user may choose to believe that this number is actually the number of missiles at Base X. If Base X really does have 3 missiles then the set of beliefs for the user includes classified information. Of course, this is really no more than a lucky guess, and it does not constitute a true disclosure of information.

As another example, assume that a corrupted top secret task reads the top secret file and then makes a series of downgrader requests to put each of the characters of this sentence in a separate unclassified file in some predetermined order. A task operating at unclassified could read these files, reassembling the characters in the correct order. Although none of the individual files would seem to carry any top secret information, in the context of a user that knows how to reassemble the data the collection of files does in fact carry top secret information. In summary, there is clearly no straightforward connection between data and information.

Due to the difficulty of formalizing an information-based definition of the high-level MLS policy, we will follow tradition and consider only the reading and writing of data.

It is also worth noting here that if we were to use a subject/information formulation of MLS then MLS/TE can quite naturally be viewed as the *conjunction* of MLS with Type Enforcement.

An operation is allowed only if no information is divulged and no Type Enforcement rules are violated. With the more traditional formulation of MLS dealing with subjects reading and writing objects, MLS/TE essentially reduces to a particular flavor of Type Enforcement policy — MLS is overridden by Type Enforcement for exceptional domains. In this case, MLS provides a set of underlying defaults that are ignored when deemed appropriate. When ignored, only Type Enforcement applies. The same effect could be achieved by a pure Type Enforcement policy if the level were considered part of the domain and type.

The primary entity types in our model of the MLS/TE policy are subjects, objects and users. We start by defining given types for these entities.

$$[\, MT\_SUBJECT, MT\_OBJECT, MT\_USER \,]$$

Various labels are attached to subjects and objects. We define the following three types of label:

$$[\, MT\_LEVEL, MT\_DOMAIN, MT\_TYPE \,]$$

The expressions mt_levels, $mt\_domains$ and mt_types denote the recognized levels, domains and types in the system. A partial ordering $\preceq$ is defined on mt_levels. We use $level_1 \preceq level_2$ to denote that $level_1$ and $level_2$ are recognized levels and that $level_1$ is at or below $level_2$. For convenience, we introduce $dominated\_by$ as a prefix form of the relation $\preceq$. In other words, $dominated\_by(level_1, level_2)$ holds exactly when $level_1 \preceq level_2$. The set $mt\_exc\_domains$ contains those domains that are considered exceptional (i.e., in which a subject may read up or write down).

$$
\begin{array}{|l}
\hline
\textit{MlsTeLabel} \\
mt\_levels : \mathbb{P}\, MT\_LEVEL \\
mt\_domains : \mathbb{P}\, MT\_DOMAIN \\
mt\_types : \mathbb{P}\, MT\_TYPE \\
\_ \preceq \_ : MT\_LEVEL \leftrightarrow MT\_LEVEL \\
dominated\_by : MT\_LEVEL \leftrightarrow MT\_LEVEL \\
mt\_exc\_domains : \mathbb{P}\, MT\_DOMAIN \\
\hline
\forall\, level : MT\_LEVEL \mid level \in mt\_levels \\
\bullet\; level \preceq level \\
\forall\, level_1, level_2 : MT\_LEVEL \mid \{\, level_1, level_2 \,\} \subseteq mt\_levels \\
\bullet\; (level_1 \preceq level_2 \wedge level_2 \preceq level_1 \\
\Rightarrow level_1 = level_2) \\
\forall\, level_1, level_2, level\_3 : MT\_LEVEL \mid \{\, level_1, level_2, level\_3 \,\} \subseteq mt\_levels \\
\bullet\; (level_1 \preceq level_2 \wedge level_2 \preceq level\_3 \\
\Rightarrow level_1 \preceq level\_3) \\
\forall\, level_1, level_2 : MT\_LEVEL \mid level_1 \preceq level_2 \\
\bullet\; \{\, level_1, level_2 \,\} \subseteq mt\_levels \\
dominated\_by = (\_ \preceq \_) \\
mt\_exc\_domains \subseteq mt\_domains \\
\hline
\end{array}
$$

In any given system state, $mt\_objects$ denotes the set of all objects in the system. Each object has an associated level and type.

$$
\begin{array}{|l}
\hline
\textit{MlsTeObject} \\
\textit{MlsTeLabel} \\
mt\_objects : \mathbb{P}\, MT\_OBJECT \\
mt\_object\_level : MT\_OBJECT \rightarrowtail MT\_LEVEL \\
mt\_object\_type : MT\_OBJECT \rightarrowtail MT\_TYPE \\
\hline
\mathrm{dom}\, mt\_object\_level = mt\_objects \\
\mathrm{ran}\, mt\_object\_level \subseteq mt\_levels \\
\mathrm{dom}\, mt\_object\_type = mt\_objects \\
\mathrm{ran}\, mt\_object\_type \subseteq mt\_types \\
\hline
\end{array}
$$

The current set of subjects in the system is $mt\_subjects$. Each subject has an associated level, domain and user.

$$
\begin{array}{|l}
\hline
\textit{MlsTeSubject} \\
\textit{MlsTeLabel} \\
mt\_subjects : \mathbb{P}\, MT\_SUBJECT \\
mt\_subject\_level : MT\_SUBJECT \rightarrowtail MT\_LEVEL \\
mt\_subject\_domain : MT\_SUBJECT \rightarrowtail MT\_DOMAIN \\
mt\_subject\_user : MT\_SUBJECT \rightarrowtail MT\_USER \\
\hline
\mathrm{dom}\, mt\_subject\_level = \mathrm{dom}\, mt\_subject\_domain \\
\quad = \mathrm{dom}\, mt\_subject\_user = mt\_subjects \\
\mathrm{ran}\, mt\_subject\_level \subseteq mt\_levels \\
\mathrm{ran}\, mt\_subject\_domain \subseteq mt\_domains \\
\hline
\end{array}
$$

The expressions mt_auth_users denotes the set of authorized users for the system. Associated with each user are a set of domains and a set of levels that are allowed for that user's subjects. These sets are modeled by $mt\_auth\_domains(user)$ and $mt\_auth\_levels(user)$. An unauthorized user is not authorized for any domains or levels.

$$
\begin{array}{|l}
\hline
\textit{MlsTeUser} \\
\textit{MlsTeLabel} \\
mt\_auth\_users : \mathbb{P}\, MT\_USER \\
mt\_auth\_domains : MT\_USER \longrightarrow \mathbb{P}\, MT\_DOMAIN \\
mt\_auth\_levels : MT\_USER \longrightarrow \mathbb{P}\, MT\_LEVEL \\
\hline
\{\, user : MT\_USER \mid mt\_auth\_domains(user) \neq \emptyset \,\} \subseteq mt\_auth\_users \\
\{\, user : MT\_USER \mid mt\_auth\_levels(user) \neq \emptyset \,\} \subseteq mt\_auth\_users \\
\forall\, user : MT\_USER \\
\bullet\; mt\_auth\_domains(user) \subseteq mt\_domains \\
\quad \wedge\; mt\_auth\_levels(user) \subseteq mt\_levels \\
\hline
\end{array}
$$

Subjects perform operations on objects. These operations have type $MT\_OPERATION$. There are two (not necessarily disjoint) classes of operations, those that alter the system state and those that observe it. These are represented by the expressions $mt\_alter\_operations$ and $mt\_observe\_operations$, respectively.

$$[\, MT\_OPERATION \,]$$

$$\begin{array}{|l}
\hline
\textit{MlsTeOperation} \\
mt_alter_operations, mt_observe_operations : \mathbb{P}\ MT_OPERATION \\
\hline
\end{array}$$

The MLS portion of the policy restricts the operations a *non*exceptional subject may perform on an object based upon the levels of the subject and object. If $slvl$ and $olvl$ are the respective levels of the subject and object, then

- the subject may perform an alter operation on the object only if $slvl \preceq olvl$, and
- the subject may perform an observe operation on the object only if $olvl \preceq slvl$.

The function mls defines the operations that nonexceptional subjects at one level are allowed to perform on objects at another. An operation is an element of the set denoted by $mls(slvl, olvl)$ only if it satisfies the restrictions on alter and observe operations described above. The expression $mls\_exceptions(d, t, slvl, olvl)$ defines the operations a subject in an exceptional domain d at level $slvl$ is allowed to perform on an object of type t at level $olvl$. We leave unspecified precisely what operations with be allowed for exceptional subjects as this will be determined by the needs of the subject and the degree of trust placed in it. We use mls and $mls\_exceptions$ to define mls_allows. Finally, every subject must execute at a level that is allowed for the user associated with it.

$$\begin{array}{|l}
\hline
\textit{MlsRestrictions} \\
MlsTeOperation \\
MlsTeSubject \\
MlsTeObject \\
MlsTeUser \\
mls : (MT_LEVEL \times MT_LEVEL) \longrightarrow \mathbb{P}\ MT_OPERATION \\
mls_exceptions : (MT_DOMAIN \times MT_TYPE \times MT_LEVEL \times MT_LEVEL) \\
\quad \longrightarrow \mathbb{P}\ MT_OPERATION \\
mls_allows : (MT_SUBJECT \times MT_OBJECT) \\
\quad \longrightarrow \mathbb{P}\ MT_OPERATION \\
\hline
(\forall\, slvl, olvl : MT_LEVEL \\
\mid mls(slvl, olvl) \neq \varnothing \\
\bullet \{slvl, olvl\} \subseteq mt_levels \\
\wedge (\forall\, op : mls(slvl, olvl) \\
\quad \bullet ((op \in mt_alter_operations \Rightarrow slvl \preceq olvl) \\
\quad \wedge (op \in mt_observe_operations \Rightarrow olvl \preceq slvl)))) \\
\\
(\forall\, d : MT_DOMAIN; t : MT_TYPE; slvl, olvl : MT_LEVEL \\
\mid mls_exceptions(d, t, slvl, olvl) \neq \varnothing \\
\bullet d \in mt_exc_domains) \\
\\
\forall\, subj : MT_SUBJECT; obj : MT_OBJECT \\
\bullet mls_allows(subj, obj) = \\
\qquad mls(mt_subject_level(subj), mt_object_level(obj)) \\
\qquad\quad \cup\, mls_exceptions(mt_subject_domain(subj), mt_object_type(obj), \\
\qquad\qquad mt_subject_level(subj), mt_object_level(obj)) \\
\quad \wedge\, mt_subject_level(subj) \in mt_auth_levels(mt_subject_user(subj)) \\
\hline
\end{array}$$

The Type Enforcement part of the policy restricts the operations that a subject may perform on an object based upon the domain of the subject and the type of the object. The expression

$te(d, t)$ denotes the set of operations that a subject with domain d is allowed to perform on an object with type t. The expression $te\_allows(subj, obj)$ denotes the operations $subj$ may perform on obj. Also, every subject must execute in a domain that is allowed for the user associated with it.

$$
\begin{array}{l}
\textit{TeRestrictions} \\
\hline
MlsTeLabel \\
MlsTeSubject \\
MlsTeObject \\
MlsTeUser \\
te : (MT\_DOMAIN \times MT\_TYPE) \longrightarrow \mathbb{P}\, MT\_OPERATION \\
te\_allows : (MT\_SUBJECT \times MT\_OBJECT) \\
\quad \longrightarrow \mathbb{P}\, MT\_OPERATION \\
\hline
\forall\, d : MT\_DOMAIN;\ t : MT\_TYPE \\
\mid te(d, t) \neq \varnothing \\
\bullet\ d \in mt\_domains \land t \in mt\_types \\
\\
\forall\, subj : MT\_SUBJECT;\ obj : MT\_OBJECT \\
\bullet\ te\_allows(subj, obj) \\
\qquad\quad = te(mt\_subject\_domain(subj), mt\_object\_type(obj)) \\
\quad \land\ mt\_subject\_domain(subj) \in mt\_auth\_domains(mt\_subject\_user(subj))
\end{array}
$$

The MLS/TE system state consists of the above information together with the function mls_te_allows which defines the operations allowed under the MLS/TE policy. To be allowed an operation must be in both $te\_allows(subj, obj)$ and $mls\_allows(subj, obj)$. Recall that the definition of mls_allows makes it possible for exceptional subjects to read up and write down.

$$
\begin{array}{l}
\textit{MlsTeSystemState} \\
\hline
MlsRestrictions \\
TeRestrictions \\
mls\_te\_allows : (MT\_SUBJECT \times MT\_OBJECT) \\
\quad \longrightarrow \mathbb{P}\, MT\_OPERATION \\
\hline
\forall\, subj : MT\_SUBJECT;\ obj : MT\_OBJECT \\
\bullet\ mls\_te\_allows(subj, obj) \\
\quad = te\_allows(subj, obj) \cap mls\_allows(subj, obj)
\end{array}
$$

8.2 MLS/TE Objects and the Kernel

The MLS/TE policy is defined in terms of subjects, objects and users. We first relate these concepts to microkernel entities. MLS/TE objects will be divided into two disjoint classes and realized in the kernel by two different types of entity: ports and memory regions. These are not the only objects that would be controlled by an MLS/TE policy in DTOS. However, to avoid obscuring the main issues in the analysis, we will restrict our attention to them. They are the primary overt information channels in Mach. The port associated with an object is denoted by $mt\_obj\_to\_port(obj)$, and the memory region by $mt\_obj\_to\_reg(obj)$. These mappings have disjoint domains, and they are both injections meaning that no two MLS/TE objects may have the same associated port or memory region. Subjects are represented by tasks. The task associated with a subject is denoted by $subj\_task(subj)$. This mapping is also an injection so that no two subjects may have the same associated task. Users will be discussed in the next section.

$$
\begin{array}{l}
\textit{MlsTeToDtos} \\
\hline
\textit{MlsTeSystemState} \\
mt\_obj\_to\_port : MT\_OBJECT \rightarrowtail\!\!\!\!\!\!\!\!\;\;\;\;\; PORT \\
mt\_obj\_to\_reg : MT\_OBJECT \rightarrowtail\!\!\!\!\!\!\!\!\;\;\;\;\; REGION \\
subj\_task : MT\_SUBJECT \rightarrowtail\!\!\!\!\!\!\!\!\;\;\;\;\; TASK \\
trusted\_task : \mathbb{P}\, MT\_SUBJECT \\
\hline
\operatorname{dom} mt\_obj\_to\_port = mt\_objects \\
\operatorname{dom} mt\_obj\_to\_reg = mt\_objects \\
\operatorname{dom} subj\_task = mt\_subjects
\end{array}
$$

The following assumptions relate to the ability of the DTOS kernel to enforce a given MLS/TE security policy. For a given high level MLS/TE policy it should be possible to define specific values for the mappings $Op\_requests$ and $Controlling\_perm$ declared below and then demonstrate the validity of Assumption 8.3.

Assumption 8.1 *There exists a mapping $Op\_requests$ from a $(MT\_OPERATION \times MT\_SUBJECT \times MT\_OBJECT)$ triple to a set of $MkRequest$ indicating the $MkRequests$ that perform the operation from the subject to the object.*

$$Op\_requests : (MT\_OPERATION \times MT\_SUBJECT \times MT\_OBJECT) \longrightarrow \mathbb{P}\, MkRequest$$

Assumption 8.2 *In the DTOS kernel each $MT\_OPERATION$ has an associated permission by which it is controlled. This permission is indicated by the $Controlling\_perm$ of the $MT\_OPERATION$. The relation $Controlled\_ops$ is the inverse of $Controlling\_perm$.*

$$
\begin{array}{l}
Controlling\_perm : MT\_OPERATION \longrightarrow PERMISSION \\
Controlled\_ops : PERMISSION \longrightarrow \mathbb{P}\, MT\_OPERATION \\
\hline
\forall\, p : PERMISSION \\
\bullet\ Controlled\_ops(p) = \{ op : MT\_OPERATION \mid Controlling\_perm(op) = p \}
\end{array}
$$

Assumption 8.3 *If a $MkRequest$ r performs a given $MT\_OPERATION$ m between a subject and object then in all states $mgr\_policy(r)$ includes a permission request for $Controlling\_perm(m)$ for the appropriate SIDs associated with the subject and object.*

8.3 MLS/TE Security Server

In this section we describe a Security Server that enforces the above security policy. Note that we will only address overt information flows. For an analysis of covert information flows in DTOS see [25, 26].

8.3.1 Security Database

8.3.1.1 Security Contexts

We first define the security contexts. A subject security context is associated with each subject. A subject context consists of the following:

- *level* — the level at which the subject is executing,

- $domain$ — the domain in which the subject is executing.
- $user$ — the user in whose behalf the subject is executing.

$$
\begin{array}{|l}
\hline
\textit{MlsTeSsc} \\
level : MT\_LEVEL \\
domain : MT\_DOMAIN \\
user : MT\_USER \\
\hline
\end{array}
$$

The object security context of an object consists of the following:

- $level$ — the object's level,
- $type$ — the object's type.

$$
\begin{array}{|l}
\hline
\textit{MlsTeOsc} \\
level : MT\_LEVEL \\
type : MT\_TYPE \\
\hline
\end{array}
$$

We define mt_sscs and mt_oscs to be the existing subject and object security contexts.

$$
\begin{array}{|l}
\hline
\textit{MlsTeContexts} \\
mt\_sscs : \mathbb{P}\ MlsTeSsc \\
mt\_oscs : \mathbb{P}\ MlsTeOsc \\
\hline
\end{array}
$$

Each subject security identifier is mapped by the function ssi_ssc to a subject security context, and each object identifier is mapped by osi_osc to an object context. The function osi_ssc maps an object SID to a subject security context. This is used when a subject is being operated upon. All three of these functions are injections.

$$
\begin{array}{|l}
\hline
\textit{MlsTeSidToContext} \\
MlsTeContexts \\
MkLabels \\
ssi\_ssc : SSI \rightarrowtail\!\!\!\!\!\!+\ MlsTeSsc \\
osi\_osc : OSI \rightarrowtail\!\!\!\!\!\!+\ MlsTeOsc \\
osi\_ssc : OSI \rightarrowtail\!\!\!\!\!\!+\ MlsTeSsc \\
\hline
\operatorname{ran} ssi\_ssc \subseteq mt\_sscs \\
\operatorname{ran} osi\_osc \subseteq mt\_oscs \\
osi\_ssc = osi\_ssi \mathbin{;} ssi\_ssc \\
\hline
\end{array}
$$

We make the following assumptions regarding consistency in the labeling of tasks and memory:

- Every OSI used to label memory or a port has an associated OSC.
- Every SSI used to label a task has an associated SSC.
- For every object j mapped to a port, the OSI used to label the port maps to an OSC with $level$ and $type$ equal to that of j.
- For every object j mapped to a memory region, the OSI used to label the region maps to an OSC with $level$ and $type$ equal to that of j.

- For every subject j, the SSI used to label the task with which j is associated maps to an SSC with *domain* and *level* equal to that of j.

If we had a more complete model of the system including the file server and the ways in which the DTOS kernel assigns SIDs, these assumptions would be justified in terms of that model.

$$
\begin{array}{|l}
\hline
\textit{MtConsistentLabels} \\
\textit{MlsTeSidToContext} \\
\textit{MlsTeToDtos} \\
\hline
\operatorname{ran} region\_osi \subseteq \operatorname{dom} osi\_osc \\
\operatorname{ran} port\_osi \subseteq \operatorname{dom} osi\_osc \\
\operatorname{ran} task\_ssi \subseteq \operatorname{dom} ssi\_ssc \\
(\forall j : \operatorname{dom} mt\_obj\_to\_port \\
\bullet (osi\_osc(port\_osi(mt\_obj\_to\_port(j)))).level = mt\_object\_level(j) \\
\quad \wedge (osi\_osc(port\_osi(mt\_obj\_to\_port(j)))).type = mt\_object\_type(j)) \\
\\
(\forall j : \operatorname{dom} mt\_obj\_to\_reg \\
\bullet (osi\_osc(region\_osi(mt\_obj\_to\_reg(j)))).level = mt\_object\_level(j) \\
\quad \wedge (osi\_osc(region\_osi(mt\_obj\_to\_reg(j)))).type = mt\_object\_type(j)) \\
\\
(\forall j : \operatorname{dom} subj\_task \\
\bullet (ssi\_ssc(task\_ssi(subj\_task(j)))).domain = mt\_subject\_domain(j) \\
\quad \wedge (ssi\_ssc(task\_ssi(subj\_task(j)))).level = mt\_subject\_level(j)) \\
\hline
\end{array}
$$

Note that we do not assume consistency for $mt\_subject\_user(j)$. Rather we will eventually justify this property based upon our interpretation of $mt\_subject\_user$ together with a pair of policy requirements. For any subject j we will interpret $mt\_subject\_user(j)$ to be

- the *USER* most recently assigned to j by an exceptional subject if such an assignment has occurred, or
- $mt\_subject\_user(p)$ where p is the parent subject and is not an exceptional subject.

Of course, the first subject created in the system (e.g., a bootstrap task) must be treated as a special case. Its user is assumed to be the system itself. This interpretation means that we trust exceptional subjects to correctly assign the user of any subjects they create. Any other subject may only create subjects that have the same user.

8.3.1.2 Policy Representation

We now define the structures used to store the Security Server representation of an MLS/TE policy.

The data maintained by the MLS/TE Security Server includes the sets of defined subject and object security contexts, the mappings from SIDs to contexts, the MLS/TE policy data (from *MlsTeSystemState*) and other miscellaneous data, *other_mt_data*, from which the generic Security Server information is extracted.

$$[\,OTHER\_MT\_DATA\,]$$

$$
\begin{array}{|l}
\hline
\textit{MlsTeData} \\
\textit{MlsTeSidToContext} \\
\textit{MlsTeSystemState} \\
other\_mt\_data : OTHER\_MT\_DATA \\
\hline
\end{array}
$$

8.3.2 Permission Requirements

In this section we state constraints on the permissions that are in $policy\_allows$. These constraints will be used below in defining the MLS/TE Security Server state.

Policy Requirement 8.1 : A permission is granted to ssi for osi only if every $MT\_OPERATION$ controlled by the permission is allowed by te from the domain of the context associated with ssi to the type of the context associated with osi.

$$\begin{array}{l}
\textit{EnforceTypes} \\
\hline
SsPolicyAllows[MlsTeData, D_SS_REQ, D_ANS] \\
\hline
\forall ss_req : policy_allows; preq : PermReq \\
\mid Perm_req(preq) = ss_req \\
\bullet \ \mathbf{let}\ sdom == ((ss_data.ssi_ssc)(preq.ssi)).domain; \\
\qquad\qquad otype == ((ss_data.osi_osc)(preq.osi)).type \\
\qquad \bullet \ Controlled_ops(preq.perm) \subseteq (ss_data.te)(sdom, otype)
\end{array}$$

Policy Requirement 8.2 : A permission is granted to ssi for osi only if every $MT\_OPERATION$ controlled by the permission is allowed by mls from the level of the context associated with ssi to the level of the context associated with osi or by $mls\_exceptions$ from the domain of the context associated with ssi to the type of the context associated with osi for the levels of the ssi and osi.

$$\begin{array}{l}
\textit{EnforceMls} \\
\hline
SsPolicyAllows[MlsTeData, D_SS_REQ, D_ANS] \\
\hline
\forall ss_req : policy_allows; preq : PermReq \\
\mid Perm_req(preq) = ss_req \\
\bullet \ \mathbf{let}\ the_ssc == ((ss_data.ssi_ssc)(preq.ssi)); \\
\qquad\qquad the_osc == ((ss_data.osi_osc)(preq.osi)) \\
\qquad \bullet \ Controlled_ops(preq.perm) \\
\qquad\qquad \subseteq (ss_data.mls_exceptions)(the_ssc.domain, the_osc.type, \\
\qquad\qquad\qquad\qquad the_ssc.level, the_osc.level) \\
\qquad\qquad\qquad \cup (ss_data.mls)(the_ssc.level, the_osc.level)
\end{array}$$

Note that we cannot just define a pair of policy requirements for simple security and the *-property because we do not know that mls in the high-level policy definition allows *all* accesses that are permitted by the dominance relation. If it does not, then the high level policy would not be achieved by the policy requirements.

To ensure monotonicity we prohibit the **task_change_sid** request by denying $Make\_sid$ permission.

Policy Requirement 8.3 : $Make\_sid$ permission is always denied.

$$\begin{array}{l}
\textit{MtMakeSid} \\
\hline
SsPolicyAllows[MlsTeData, D_SS_REQ, D_ANS] \\
\hline
\forall ss_req : policy_allows; preq : PermReq \\
\mid Perm_req(preq) = ss_req \\
\bullet \ preq.perm \neq Make_sid
\end{array}$$

Policy Requirement 8.4 : $Create\_task$ and $Cross\_context\_create$ permissions for osi (for a subject being acted upon) are given to ssi only if both SIDs map to subject contexts with the same $user$ component or the domain of the context associated with ssi is in $mt\_exc\_domains$.

$$
\begin{array}{l}
\textit{MtCorrectUser} \\
\hline
SsPolicyAllows[MlsTeData, D\_SS\_REQ, D\_ANS] \\
\hline
\forall ss\_req : policy\_allows; preq : PermReq \\
\mid Perm\_req(preq) = ss\_req \\
\quad \land preq.perm \in \{Create\_task, Cross\_context\_create\} \\
\bullet ((ss\_data.ssi\_ssc)(preq.ssi)).domain \in (ss\_data.mt\_exc\_domains) \\
\quad \lor ((preq.osi) \in \mathrm{dom}(ss\_data.osi\_ssc) \\
\qquad \land ((ss\_data.ssi\_ssc)(preq.ssi)).user \\
\qquad\quad = ((ss\_data.osi\_ssc)(preq.osi)).user)
\end{array}
$$

Policy Requirement 8.5 : $Create\_task$ and $Cross\_context\_create$ permissions for osi (the SID to be assigned to the subject) are given only if the domain and level of $osi\_ssc(osi)$ are allowed for the user of $osi\_ssc(osi)$.

$$
\begin{array}{l}
\textit{MtDomainLevel} \\
\hline
SsPolicyAllows[MlsTeData, D\_SS\_REQ, D\_ANS] \\
\hline
\forall ss\_req : policy\_allows; preq : PermReq \\
\mid Perm\_req(preq) = ss\_req \\
\quad \land preq.perm \in \{Create\_task, Cross\_context\_create\} \\
\bullet ((preq.osi) \in \mathrm{dom}(ss\_data.osi\_ssc) \\
\quad \land (\mathbf{let}\ the\_ssc == ((ss\_data.osi\_ssc)(preq.osi)) \\
\qquad \bullet the\_ssc.domain \in (ss\_data.mt\_auth\_domains)(the\_ssc.user) \\
\qquad \land the\_ssc.level \in (ss\_data.mt\_auth\_levels)(the\_ssc.user)))
\end{array}
$$

8.3.3 Security Server State

The MLS/TE Security Server combines the general properties of a DTOS Security Server (i.e., the types $MkRequest$, D_SS_REQ, D_RESP and D_ANS) with the MLS/TE Security Server data and the above policy requirements.

$$
\begin{array}{l}
\textit{MlsTeState} \\
\hline
SsState[MkRequest, MlsTeData, D\_SS\_REQ, D\_RESP, D\_ANS] \\
EnforceTypes \\
EnforceMls \\
MtMakeSid \\
MtCorrectUser \\
MtDomainLevel
\end{array}
$$

8.4 Operations

The operations of the MLS/TE Security Server are basically those of the generic server. The only additional constraint is that the policy remain unchanged. This requirement is formalized by $MlsTePolicyInvariant$.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad MlsTePolicyInvariant \quad}} \\
SsStep[MkRequest, MlsTeData, D_SS_REQ, D_RESP, D_ANS] \\
\hline
policy_allows' = policy_allows \\
\hline
\end{array}$$

$$\begin{aligned}
&MlsTeSendNegativeResponse \\
&\quad \widehat{=}\ SsSendNegativeResponse[MkRequest, MlsTeData, D_SS_REQ, D_RESP, D_ANS] \\
&\qquad \land\ MlsTePolicyInvariant \\
&MlsTeSendAffirmativeResponse \\
&\quad \widehat{=}\ SsSendAffirmativeResponse[MkRequest, MlsTeData, D_SS_REQ, D_RESP, D_ANS] \\
&\qquad \land\ MlsTePolicyInvariant \\
&MlsTeReceiveRequest \\
&\quad \widehat{=}\ SsReceiveRequest[MkRequest, MlsTeData, D_SS_REQ, D_RESP, D_ANS] \\
&\qquad \land\ MlsTePolicyInvariant \\
&MlsTeMgrRequest \\
&\quad \widehat{=}\ SsMgrRequest[MkRequest, MlsTeData, D_SS_REQ, D_RESP, D_ANS] \\
&\qquad \land\ MlsTePolicyInvariant \\
&MlsTeInternalTransition \\
&\quad \widehat{=}\ SsInternalTransition[MkRequest, MlsTeData, D_SS_REQ, D_RESP, D_ANS] \\
&\qquad \land\ MlsTePolicyInvariant
\end{aligned}$$

8.5 Component Specification

We can use the above to specify the MLS/TE security server as a component.

The **guar** for the MLS/TE security server allows any of the transitions described in Section 8.4. This is modeled by mt_guar.

$$\begin{aligned}
&MtGuarStep \\
&\widehat{=}\ MlsTeReceiveRequest \\
&\quad \lor\ MlsTeSendNegativeResponse \\
&\quad \lor\ MlsTeSendAffirmativeResponse \\
&\quad \lor\ MlsTeMgrRequest \\
&\quad \lor\ MlsTeInternalTransition
\end{aligned}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad MtGuar \quad}} \\
mt_guar : \mathbb{P}\ MtGuarStep \\
\hline
mt_guar = MtGuarStep \\
\hline
\end{array}$$

The MLS/TE security server assumes that the assumptions of generic security servers in $SsRely$ are satisfied.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad MtRely \quad}} \\
mt_rely : \mathbb{P}\ \Delta\ MlsTeState \\
SsRely[MkRequest, MlsTeData, D_SS_REQ, D_RESP, D_ANS] \\
\hline
mt_rely = ss_rely \\
\hline
\end{array}$$

The set of allowed initial states is modeled by mt_init. We require that an initial state satisfy the constraints imposed by $SsInit$.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{MtInit}} \\
\hline
SsInit[MkRequest, MlsTeData, D\_SS\_REQ, D\_RESP, D\_ANS] \\
mt\_init : \mathbb{P}\ MlsTeState \\
\hline
mt\_init \subseteq ss\_init \\
\hline
\end{array}
$$

All information in $MlsTeState$ is visible to the MLS/TE security server.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{MtView}} \\
\hline
mt\_view : MlsTeState \leftrightarrow MlsTeState \\
\hline
\forall st_1, st_2 : MlsTeState \\
\bullet\ (st_1, st_2) \in mt\_view \Leftrightarrow st_1 = st_2 \\
\hline
\end{array}
$$

$$
\begin{array}{l}
MtComponent \\
\quad \widehat{=}\ MtGuar \land MtRely \land MtInit \land MtView
\end{array}
$$

The following lemma is a trivial consequence of the invariance of $policy\_allows$ and the definition of mt_rely.

Lemma 8.4 *No SS_REQ that has been either granted or volunteered by the security server is ever subsequently removed from $policy\_allows$.*

8.6 Composing DTOS and MLS/TE

In this section, we first compose the DTOS kernel with the MLS/TE security server. We then show that this composite system implements the abstract MLS/TE policy.

The composite state is $MkMtState$ which contains all the components of $MkState$ and $MlsTeState$.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{MkMtState}} \\
\hline
MkState \\
MlsTeState \\
\hline
\end{array}
$$

Each kernel state is associated with the set of all $MkMtState$ that have the same value for each of the components of $MkState$. Similarly, each MLS/TE security server state is associated with the set of all $MkMtState$ that have the same value for each of the components of $MlsTeState$.

The set of allowed initial states for the composition of two components is the intersection of the two sets (after mapping them into the composite state). This set of states is modeled by mk_mt_init. Since mk_mt_init is nonempty, the kernel and MLS/TE security server are composable.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad MkMtInit \quad}} \\
mk\_mt\_init : \mathbb{P}\ MkMtState \\
\hline
\forall\ st : mk\_mt\_init \\
\bullet\ st.pending\_responses = \varnothing \\
\qquad \land\ st.pending\_requests = \varnothing \\
\qquad \land\ st.active\_request = \varnothing \\
\qquad \land\ st.sent = \varnothing \\
\qquad \land\ st.obtained = \varnothing \\
\qquad \land\ st.allowed = \varnothing \\
\qquad \land\ st.responses = \varnothing \\
\qquad \land\ st.active\_computations = \varnothing \\
\hline
\end{array}
$$

In composing the kernel and MLS/TE security server we will use respect relations that require each component to leave alone its peer's internal data.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad MkMtRespect \quad}} \\
mk\_respect\_mt : \mathbb{P}\ \Delta\ MkMtState \\
mt\_respect\_mk : \mathbb{P}\ \Delta\ MkMtState \\
\hline
mk\_respect\_mt = \{\ \Delta\ MkMtState \\
\qquad |\ \Xi\ SsInternals[MkRequest, MlsTeData, D\_SS\_REQ, D\_RESP, D\_ANS]\ \} \\
mt\_respect\_mk = \{\ \Delta\ MkMtState \\
\qquad |\ \Xi\ MgrInternals[MkData, MkRequest, D\_SS\_REQ, D\_RESP, D\_ANS]\ \} \\
\hline
\end{array}
$$

The **guar** of the composite is denoted by mk_mt_guar.

$$
\begin{array}{l}
MkMtStep \\
\mathrel{\widehat=} (\Xi\ SsInternals[MkRequest, MlsTeData, D\_SS\_REQ, D\_RESP, D\_ANS] \\
\qquad \land\ MkGuarStep) \\
\lor\ (\Xi\ MgrInternals[MkData, MkRequest, D\_SS\_REQ, D\_RESP, D\_ANS] \\
\qquad \land\ MtGuarStep)
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad MkMtGuar \quad}} \\
mk\_mt\_guar : \mathbb{P}\ MkMtStep \\
\hline
mk\_mt\_guar = MkMtStep \\
\hline
\end{array}
$$

The **rely** of the composite is the intersection of the two rely relations.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad MkMtRely \quad}} \\
mk\_mt\_rely : \mathbb{P}\ \Delta\ MkMtState \\
\hline
mk\_mt\_rely = \{\ \Delta\ MkMtState \\
\qquad |\ \Xi\ MgrInternals[MkData, MkRequest, D\_SS\_REQ, D\_RESP, D\_ANS] \\
\qquad \land\ \Xi\ SsInternals[MkRequest, MlsTeData, D\_SS\_REQ, D\_RESP, D\_ANS] \\
\qquad \land\ \Xi\ SharedInterpretation[MkRequest, D\_SS\_REQ, D\_RESP, D\_ANS] \\
\qquad \land\ pending\_ss\_requests \sqsubseteq pending\_ss\_requests' \\
\qquad \land\ pending\_responses' = pending\_responses\ \} \\
\hline
\end{array}
$$

To show that the Composition Theorem applies to the composite, we must show that

$$mk\_guar \cap mk\_respect\_mt \subseteq mt\_guar \cup mt\_rely \cup mt\_view,$$

and

$$mt\_guar \cap mt\_respect\_mk \subseteq mk\_guar \cup mk\_rely \cup mk\_view.$$

The proofs of these properties are essentially identical to the corresponding proofs for the composition of the generic manager and security server.

Theorem 8.5 (DTOS MLS/TE Consistency Theorem) *Let Sys be a system including the MLS/TE Security Server together with the DTOS kernel as the manager. If the DTOS kernel processes a request* $req?$ *in some state* s*, then for each* $r \in mgr\_policy(req?) \cap permission\_requests$ *in* s*, either*

- $r \in retained(req?)$ *in the system initial state, or*
- $r \in policy\_allows$ *in state* s*.*

Proof: Follows immediately from Lemma 8.4 and the DTOS Consistency Theorem. Note that the DTOS kernel will never process a **task_change_sid** request since $Make\_sid$ permission is denied in all cases. □

Since $policy\_allows$ is static the following Corollary holds:

Corollary 8.6 (DTOS MLS/TE Complete Consistency) *Let Sys be a system that satisfies the hypotheses of the DTOS MLS/TE Consistency Theorem and in which, for some request* $req?$*,* $policy\_allows$ *in the initial state contains all elements from* $retained(req?)$ *in the initial state. In Sys, if the manager processes* $req?$ *in some state* s*, then for each* $r \in mgr\_policy(req?) \cap permission\_requests$ *in* s*,* $r \in policy\_allows$ *in state* s*.*

Note that if we were to completely define $retained$ in the initial state for the DTOS kernel and $policy\_allows$ for a given MLS/TE security server then we could demonstrate that for every $req?$ the hypotheses of the DTOS MLS/TE Complete Consistency lemma is satisfied. We could then deduce a simple consistency property that $policy\_allows$ in any reachable state s contains all permissions required for any request executed in state s.

Definition 8.7 *If the composition of a manager with a security server has the property that in the initial state, for every request* r*,* $retained(r) \subseteq policy\_allows$ *then it is* ***initially consistent****.*

Theorem 8.8 *For an initially consistent system, if the DTOS kernel performs* op *on* obj *for* $subj$ *then* $op \in te\_allows(subj, obj)$*.*

Proof: By Assumption 8.1 DTOS performs op on obj for $subj$ by performing an $MkRequest$ r. Given a specific value for the mapping $Op\_requests$, we could perform an analysis of cases at this point, demonstrating that each of the possible requests for each operation has a permission requirement that ensures compliance with the high-level policy. Lacking such a definition we instead refer to Assumption 8.3 to conclude that $mgr\_policy(r)$ contains a $PermReq$ p with $perm = Controlling\_perm(op)$, and with ssi and osi taken to be the SIDs of the kernel entities associated with the subject and object. Since the system is assumed to be initially consistent the DTOS MLS/TE Complete Consistency theorem implies that $p \in policy\_allows$.

Here we must perform a case analysis on the possible types of SID that can occur in any $p \in mgr\_policy(r)$. Since mgr_policy for DTOS is only partially specified we will merely provide an example here.

Consider the case where the SSI is $task\_ssi(subj\_task(subj))$ and the OSI is $region\_osi(mt\_obj\_to\_reg(obj))$. **PR8.1** implies that

$$\begin{aligned} &Controlled\_ops(Controlling\_perm(op)) \\ &\quad \subseteq te(ssi\_ssc(task\_ssi(subj\_task(subj))), \\ &\qquad\qquad osi\_osc(region\_osi(mt\_obj\_to\_reg(obj)))) \end{aligned}$$

Applying $MtConsistentLabels$ and the definition of te_allows we find that

$$\begin{aligned} &Controlled\_ops(Controlling\_perm(op)) \\ &\quad \subseteq te(mt\_subject\_domain(subj), mt\_object\_type(obj)) \\ &\quad = te\_allows(subj, obj) \end{aligned}$$

To conclude the proof we note that $op \in Controlled\_ops(Controlling\_perm(op))$. □

Theorem 8.9 *For an initially consistent system, if the DTOS kernel performs* op *on* obj *for* $subj$ *then* $op \in mls\_allows(subj, obj)$.

Proof: The proof is analogous to that for Theorem 8.8. □

Definition 8.10 *If in a state* s*, for every subject* j

$$(ssi\_ssc(task\_ssi(subj\_task(j)))).user = mt\_subject\_user(j)$$

we will say s *is* ***user consistent***.

Lemma 8.11 *If the DTOS MLS/TE system is initially consistent and is user consistent in the initial state, then it is user consistent in every state.*

Proof: Recall that we interpret $mt\_subject\_user$ of a subject to be the user assigned to it by an exceptional subject if this has happened and otherwise the user associated with the parent subject. This theorem is proved by induction. The base case is true by hypothesis. To prove the inductive step we show that

1. The only way for a security context to be associated with a subject is through creation of the subject (i.e., task creation),
2. Whenever a new context is associated with a subject the new context does not introduce a violation of user consistency.

We assume that for any existing subject the task associated with that subject by $subj\_task$ does not change. Furthermore we will assume that the context that ssi_ssc associates with any SSI assigned to an existing task does not change. Since the security context associated with a subject j is defined by $ssi\_ssc(task\_ssi(subj\_task(j)))$, the only way to associate a context with a subject is by creating a new subject, with its associated task, (i.e., extending $subj\_task$) or by modifying $task\_ssi$. The former occurs only during task creation, and the latter occurs only during task creation and the **task_change_sid** operation. The **task_change_sid** operation never occurs since $Make\_sid$ permission is always prohibited and always required by mgr_policy. This demonstrates part (1).

For part (2), if a context is assigned to a subject in a request by an exceptional subject, then by definition, user consistency holds for that subject in the resulting state. So we need only consider the case of a task creation request r by a nonexceptional subject n to create a task t. In this case $mt\_subject\_user'((subj\_task')^{\sim}(t)) = mt\_subject\_user(n)$. A task creation operation will require either $Create\_task$ or $Cross\_context\_create$ permission from the SID of n to the SID to be assigned to t. The DTOS MLS/TE Complete Consistency theorem implies that these permissions are in $policy\_allows$. **PR8.4** then implies that

$$\begin{aligned} ssi\_ssc(task\_ssi(subj\_task(n))).user &= osi\_ssc'(new\_sid).user \\ &= ssi\_ssc'(task\_ssi'(t)).user. \end{aligned}$$

Since by the inductive hypothesis the system is user consistent in the state in which r is processed, the left hand side is $mt\_subject\_user(n)$, and this concludes the proof. □

Theorem 8.12 *Assume the DTOS MLS/TE system is initially consistent, user consistent in the initial state and that for every subject j in the initial state the following properties hold:*

$$mt\_subject\_domain(j) \in mt\_auth\_domains(mt\_subject\_user(j))$$
$$mt\_subject\_level(j) \in mt\_auth\_levels(mt\_subject\_user(j))$$

Then, the above properties hold for all j in every reachable state.

Proof: We will prove only the first property. The proof of the second is nearly identical. Applying $MtConsistentLabels$ and Theorem 8.11 the first property becomes

$$\begin{aligned} &ssi\_ssc(task\_ssi(subj\_task(j))).domain \\ &\quad \in mt\_auth\_domains(ssi\_ssc(task\_ssi(subj\_task(j))).user) \end{aligned}$$

That is, we must show that in all states and for every subject j, in the context assigned to j the domain is an authorized domain for the user.

Following the reasoning used in the proof of Theorem 8.11, a context can be assigned to j only during the creation of j. The permission required for the relevant request must be in $policy\_allows$. The client of the request that assigns an SSI s to j must hold at least one of the permissions $Create\_task$ and $Cross\_context\_create$ to $ssi\_osi(s)$. **PR8.5** implies that

$$osi\_ssc(ssi\_osi(s)).domain \in mt\_auth\_domains(osi\_ssc(ssi\_osi(s)).user).$$

Using $MlsTeSidToContext$ this is equivalent to

$$ssi\_ssc(s).domain \in mt\_auth\_domains(ssi\_ssc(s).user).$$

Since in the resulting state $s = task\_ssi(subj\_task(j))$, we are done.

Section 9 Clark-Wilson Policy

The Clark-Wilson security policy is an integrity policy. As such it is concerned with the correctness of data and the prevention of fraud rather than the prevention of disclosure. The data items that are to be protected are called *constrained data items* (CDIs). The correctness of a CDI is protected in two ways. The first is through *integrity verification procedures* (IVPs). An IVP is intended to verify the integrity of a CDI in one or more of the following senses:

- the internal consistency of a particular CDI,
- the consistency of CDIs with each other, and
- the consistency of CDIs with the external world.

The second way in which CDI correctness is protected is by allowing CDIs to be modified only by certain programs, called *transformation procedures* (TPs), that have been certified to take the set of CDIs from one valid state to another. Validity is defined in some application-specific way.

Prevention of fraud is furthered by providing mechanisms for the separation of duty. Only certain users are allowed to modify a given CDI and then only by using a particular TP. Thus, we can prevent the person who can run the check-writing program from also running the equipment purchasing program. In this way no single person can produce a purchase order, discard it, and then write a check to pay for an item which is never ordered. Fraud then requires at least two people conspiring together.

In Section 9.1 we describe the elements of the system state that are relevant to the Clark-Wilson security policy, and we state the requirements placed upon these state elements to define the Clark-Wilson policy. Section 9.2 defines the correspondence between the concepts of the abstract, high-level definition of Clark-Wilson and the entities of the DTOS kernel. Sections 9.3–9.5 describe a Security Server that enforces the Clark-Wilson policy, and Section 9.6 composes the Clark-Wilson security server with the DTOS kernel and demonstrates that the high-level policy is satisfied.

9.1 Formal Clark-Wilson Definition

We first describe the entities relevant to defining the Clark-Wilson policy that are visible to a person using a Clark-Wilson system. They are operating system entities, not Security Server entities. We start with the following given types.

$$[DATA\_ITEM, INDIVIDUAL, PROCEDURE, CW\_PROCESS]$$

$DATA\_ITEM$ is the type of all data items, both constrained and unconstrained. The types $INDIVIDUAL$, $PROCEDURE$ and $CW\_PROCESS$ contain all individuals, procedures and processes, respectively. A procedure corresponds to an executable file and each process executes some procedure. The Clark-Wilson policy uses the term *user* rather than *individual*; since in other DTOS documents the term *user* refers to a component of a security context, we will substitute the term *individual* when defining Clark-Wilson (except when quoting the certification and enforcement rules of the policy). The term *user* will be used in Section 9.3 in defining the structure of security contexts.

The set of all data items in the system is $data\_items$. The set $constrained\_data\_items$ is the subset of $data\_items$ consisting of the constrained data items (CDIs). The set $unconstrained\_data\_items$ is the subset of $data\_items$ consisting of the unconstrained data items (UDIs). There is a log, indicated by log, which is a CDI that holds information on transformations that have been applied to constrained data items.

DataItem

$data\_items : \mathbb{P}\ DATA\_ITEM$
$constrained\_data\_items : \mathbb{P}\ DATA\_ITEM$
$unconstrained\_data\_items : \mathbb{P}\ DATA\_ITEM$
$log : DATA\_ITEM$

$constrained\_data\_items \subseteq data\_items$
$unconstrained\_data\_items = data\_items \setminus constrained\_data\_items$
$log \in constrained\_data\_items$

Transformation procedures (TPs) are used to transform one valid[25] system state to another. The set $procedures$ contains all of the defined procedures for the system, and $transformation\_procedures$ is the subset of $procedures$ consisting only of the certified transformation procedures. Each transformation procedure must be certified by a security officer to manipulate one or more sets of CDIs. This is modeled by the relation $may\_manipulate$. Note that a single TP might be certified for several different sets of CDIs. Thus, the $may\_manipulate$ relation might not be a function. To find all sets of CDIs that may be manipulated by a TP tp, we will take the relational image $may\_manipulate(\!| \{tp\} |\!)$. A TP may also read and write UDIs. Since the integrity of UDIs is not an explicit goal of a Clark-Wilson system, manipulation of them is not controlled by the system. Also, since we are concerned with integrity and not disclosure we focus on manipulation (i.e., modification of CDIs) and ignore the reading of CDIs. This pushes onto the certification process the burden of ensuring that a TP is run only with appropriate input information. This is unavoidable since new information can enter the system only through UDIs, and they have no integrity. A TP might also be executed without using any CDIs. For example, it might be possible to set program switches for the TP so that it only parses its UDI input files to check for correct syntax, and no files are modified.

Review Note:
It is unclear whether Clark and Wilson intended the range of $may\_manipulate$ to be $\mathbb{P}\ constrained\_data\_items$ or $\mathrm{seq}\ constrained\_data\_items$. They refer to it using both the terms "set" and "list". The latter emphasizes that each CDI must fill a particular role in the TP computation while the former says nothing about this. The latter would be a stronger statement of the Clark-Wilson policy. Either definition can be supported in DTOS. We have chosen $\mathbb{P}\ constrained\_data\_items$ in this report. Similar comments apply to $may\_execute$ below.

---

[25] The definition of a valid state depends upon the use of the system. A necessary condition would be that all the IVPs are satisfied.

$$
\begin{array}{|l}
\hline
\textit{TransformationProcedure} \\
\hline
\mathit{DataItem} \\
\mathit{procedures} : \mathbb{P}\, \mathit{PROCEDURE} \\
\mathit{transformation\_procedures} : \mathbb{P}\, \mathit{PROCEDURE} \\
\mathit{may\_manipulate} : \mathit{PROCEDURE} \leftrightarrow \mathbb{P}\, \mathit{DATA\_ITEM} \\
\hline
\mathit{transformation\_procedures} \subseteq \mathit{procedures} \\
\mathrm{dom}\, \mathit{may\_manipulate} \subseteq \mathit{transformation\_procedures} \\
\mathrm{ran}\, \mathit{may\_manipulate} \subseteq \mathbb{P}\, \mathit{constrained\_data\_items} \\
\hline
\end{array}
$$

Integrity verification procedures (IVPs) are used to confirm that all of the CDIs in the system conform to the integrity specification at the time at which the IVPs are executed. These procedures must also be certified by the security officer. We leave open the possibility that an IVP is also a TP.

$$
\begin{array}{|l}
\hline
\textit{IntegrityVerficationProcedure} \\
\hline
\mathit{TransformationProcedure} \\
\mathit{certified\_IVPs} : \mathbb{P}\, \mathit{PROCEDURE} \\
\hline
\mathit{certified\_IVPs} \subseteq \mathit{procedures} \\
\hline
\end{array}
$$

The set $\mathit{individuals}$ contains all existing individuals in the system, and $\mathit{authenticated\_individuals}$ is the set of all individuals who are authenticated for the system. (We allow the possibility of unauthenticated individuals, e.g., anonymous ftp individuals.) In the Clark-Wilson policy it is paramount that the person responsible for the execution of a TP can be identified. Thus, we will assume that stringent authentication mechanisms are applied to any person who wishes to execute a TP.

Each individual may execute only certain TPs manipulating only certain CDIs. This certification information is recorded by the function $\mathit{may\_execute}$. The security officer is called the $\mathit{certifier}$ and is one of the authenticated individuals.

$$
\begin{array}{|l}
\hline
\textit{Individual} \\
\hline
\mathit{DataItem} \\
\mathit{TransformationProcedure} \\
\mathit{individuals} : \mathbb{P}\, \mathit{INDIVIDUAL} \\
\mathit{authenticated\_individuals} : \mathbb{P}\, \mathit{INDIVIDUAL} \\
\mathit{may\_execute} : \mathit{INDIVIDUAL} \longrightarrow (\mathit{PROCEDURE} \leftrightarrow \mathbb{P}\, \mathit{DATA\_ITEM}) \\
\mathit{certifier} : \mathit{INDIVIDUAL} \\
\hline
\mathit{authenticated\_individuals} \subseteq \mathit{individuals} \\
\mathrm{ran}\, \mathit{may\_execute} \subseteq \mathbb{P}(\mathit{transformation\_procedures} \times \mathbb{P}\, \mathit{constrained\_data\_items}) \\
\mathit{certifier} \in \mathit{authenticated\_individuals} \\
\forall\, i : \mathit{INDIVIDUAL} \\
\bullet\; \mathit{may\_execute}(i) = \varnothing \vee i \in \mathit{authenticated\_individuals} \\
\hline
\end{array}
$$

As with $\mathit{may\_manipulate}$ the relation $\mathit{may\_execute}(u)$ for any individual u in the domain of $\mathit{may\_execute}$ might not be a function, and we will take the relational image (e.g., $(\mathit{may\_execute}(u))(\!|\{tp\}|\!)$) to find all sets of CDIs that may be manipulated by tp executing on behalf of u.

The set $\mathit{processes}$ contains all processes executing on the system. The function $\mathit{process\_individual}$ maps each executing process to the individual on whose behalf the process is executing, and *process_procedure* maps an executing process to the procedure that it is executing. The function *process_manipulates* indicates the constrained data items which are being (or have been) manipulated by each process. We assume that if a process has manipulated any CDIs then it is executing a procedure.

$$
\begin{array}{|l}
\textit{Process} \\
\hline
\textit{DataItem} \\
\textit{Individual} \\
\textit{TransformationProcedure} \\
processes : \mathbb{P}\, CW\_PROCESS \\
process\_individual : CW\_PROCESS \nrightarrow INDIVIDUAL \\
process\_procedure : CW\_PROCESS \nrightarrow PROCEDURE \\
process\_manipulates : CW\_PROCESS \nrightarrow \mathbb{P}\, DATA\_ITEM \\
\hline
\operatorname{dom} process\_individual = \operatorname{dom} process\_manipulates = processes \\
\operatorname{dom} process\_procedure \subseteq processes \\
\operatorname{ran} process\_manipulates \subseteq \mathbb{P}\, constrained\_data\_items \\
\operatorname{ran} process\_individual \subseteq individuals \\
\operatorname{ran} process\_procedure \subseteq procedures \\
\forall\, p : CW\_PROCESS \\
\bullet\ process\_manipulates(p) = \varnothing \vee p \in \operatorname{dom} process\_procedure \\
\hline
\end{array}
$$

The part of the system state relevant to the Clark-Wilson policy consists of all the above information.

$$
\begin{array}{|l}
\textit{State} \\
\hline
\textit{DataItem} \\
\textit{TransformationProcedure} \\
\textit{IntegrityVerficationProcedure} \\
\textit{Individual} \\
\textit{Process} \\
\hline
\end{array}
$$

Next, we describe the restrictions placed by Clark-Wilson on the system states as described above. These restrictions deal with the relationships between CDIs, individuals, TPs and processes. It is essential to keep in mind that these are operating system entities, not Security Server entities. We are describing the constraints on the entities visible at the level of a person using the system. This defines the goal we wish to achieve when defining the Security Server in Section 9.3.

The Clark-Wilson policy has two parts, certification and enforcement. Certification controls the contents of the sets *certified_IVPs*, *constrained_data_items*, *may_manipulate* and *may_execute*. Although the results of certification are registered in these sets, the certification itself is outside the system. The system handles enforcement only, so that is where we will focus our attention. A change to the certifications (and to the corresponding certification sets and relations) constitutes a change in the security policy. We ignore the way in which the certifications are entered into the system and treat the certifications (and hence the security policy) as static. For clarity we will include English statements of the certification parts of the policy, but they will not be formalized.

The certification and enforcement rules given in this section are taken from [8].

Certification 1 (C1) *All IVPs must properly ensure that all CDIs are in a valid state at the time the IVP is run.*

Certification 2 (C2) *All TPs must be certified to be valid. That is, they must take a CDI to a valid final state, given that it is in a valid state to begin with. For each TP, and each set of CDIs that it may manipulate, the security officer must specify a "relation," which defines that execution. A relation is thus of the form: (TPi, (CDIa, CDIb, CDIc, ...)), where the list of CDIs defines a particular set of arguments for which the TP has been certified.*

Enforcement 1 (E1) *The system must maintain the list of relations specified in rule C2, and must ensure that the only manipulation of any CDI is by a TP, where the TP is operating on the CDI as specified in some relation.*

Any process will either manipulate no CDIs, or it will manipulate a subset of some set of CDIs for which its executing TP has been certified.

E1TransProcMayManipulate

$$\begin{array}{l} State \\ \hline \forall\, p : processes \mid process\_manipulates(p) \neq \varnothing \\ \bullet\ (\exists\, S : may\_manipulate(\!|\{process\_procedure(p)\}|\!) \\ \qquad \bullet\ process\_manipulates(p) \subseteq S) \end{array}$$

Enforcement 2 (E2) *The system must maintain a list of relations of the form: (UserID, TPi, (CDIa, CDIb, CDIc, ...)), which relates a user, a TP, and the data objects that TP may reference on behalf of that user. It must ensure that only executions described in one of the relations are performed.*

Any process will either manipulate no CDIs, or it will manipulate a subset of some set of CDIs that its individual has been certified to manipulate via the executing TP.

E2IndividualMayExecute

$$\begin{array}{l} State \\ \hline \forall\, p : processes \mid process\_manipulates(p) \neq \varnothing \\ \bullet\ (\exists\, S : (may\_execute(process\_individual(p)))(\!|\{process\_procedure(p)\}|\!) \\ \qquad \bullet\ process\_manipulates(p) \subseteq S) \end{array}$$

As Clark and Wilson have noted, **E1** is subsumed by **E2**. They maintain that "for philosophical and practical reasons, it is helpful to have both sorts of relations." **E1** focuses on internal consistency — ensuring that CDIs are in a consistent state within and among themselves. **E2** focuses on separation of duty in an effort to prevent fraud and maintain external consistency with the real world. (Compare **C2** and **C3**.) They claim that, as a matter of practicality, it is also advantageous to keep both lists since it allows the use of wild card characters that match classes of TPs or CDIs when defining complex policies. We have maintained both **E1** and **E2** for consistency with Clark and Wilson.

Certification 3 (C3) *The list of relations in E2 must be certified to meet the separation of duty requirement.*

Enforcement 3 (E3) *The system must authenticate the identity of each user attempting to execute a TP.*

This enforcement rule depends upon authentication mechanisms, and it is largely beyond the scope of this report. If one person is able to log onto the system in someone else's name, that constitutes a severe violation of Clark-Wilson. However, there is nothing that the Security Server itself can do to prevent this. The enforcement of the Clark-Wilson policy will be only as good as the authentication of its users.

We will merely require that any individual who wishes to execute a TP must be an authenticated individual (as opposed to some sort of anonymous or guest individual). We rely on good authentication mechanisms combined with strong institutional policies for confidentiality of authentication information (e.g., passwords or personal identification numbers) to ensure that the person logged on to an account is really the intended person.

$E3AuthenticatedExecutor$

$State$

$\forall p : processes$
$\mid p \in \operatorname{dom} process\_procedure \wedge process\_procedure(p) \in transformation\_procedures$
$\bullet\ process\_individual(p) \in authenticated\_individuals$

Certification 4 (C4) *All TPs must be certified to write to an append-only CDI (the log) all information necessary to permit the nature of the operation to be reconstructed.*

Certification 5 (C5) *Any TP that takes a UDI as an input value must be certified to perform only valid transformations, or else no transformations, for any possible value of the UDI. The transformations should take the input from a UDI to a CDI, or the UDI is rejected. Typically, this is an edit program.*

Enforcement 4 (E4) *Only the agent permitted to certify entities may change the list of such entities associated with other entities: specifically, the associated with a TP [sic]. An agent that can certify an entity may not have any execute rights with respect to that entity.*

The first sentence is ignored here since it describes the way in which the certification of programs is described to the system (i.e., as certification lists) and therefore deals with certification. We formalize the second sentence by stating that the individual associated with any process that is executing a TP must not be the certifier.

Editorial Note:
Note that this formalization might be somewhat too strong in that it prevents the certifier from running *any* TPs. As a result, in order to modify the certifications of objects (i.e., change the security policy) the certifier must have available some secure mechanism other than TP execution. For example, if the Security Server executes on a different processor the certifier could directly interact with the Security Server, bypassing the Clark-Wilson system entirely. Of course, the certifier can still execute non-TPs (e.g., programs under development and programs submitted for certification as TPs).

A more practical Clark-Wilson design would probably treat many more objects including source and object code representations of TPs and IVPs and the certification data itself as CDIs. A TP would be used to modify the certification data. Since **E4** states that no agent may certify a program to which she has execute rights, the certifier may not certify the certification TP — some other agent must certify it. To do this the second agent must use a certification TP on a different piece of certification data. This certification TP must be a different one from the first since otherwise, Agent 2 could execute an item that she certified. Fortunately, we can stop here since Agent 1 can certify the second certification TP using the first one. We have chosen to avoid this complexity entirely in this report.

$E4MustNotBeCertifier$

$State$

$\forall p : processes$
$\mid p \in \operatorname{dom} process\_procedure \wedge process\_procedure(p) \in transformation\_procedures$
$\bullet\ process\_individual(p) \neq certifier$

A Clark-Wilson state is one that obeys the above constraints.

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\textit{ClarkWilson}} \\
\textit{State} \\
\textit{E1TransProcMayManipulate} \\
\textit{E2IndividualMayExecute} \\
\textit{E3AuthenticatedExecutor} \\
\textit{E4MustNotBeCertifier} \\
\hline
\end{array}
$$

9.2 Clark-Wilson Objects and the Kernel

The Clark-Wilson policy is defined in terms of data items, procedures, processes and individuals. Since these concepts do not exist at the microkernel level, we map them to microkernel entities. This mapping also implicitly associates SIDs with Clark-Wilson entities.

Data items and procedures will be represented by memory objects. The associated memories can be obtained via the mappings di_memory and $proc\_memory$. These mappings are injections, meaning that no two data items or procedures may have the same associated memory. Furthermore, no data item and procedure may be associated with the same memory. This assumption is made for simplicity. In reality we would want the TP executables to be CDIs as well. We assume that any region that is mapped to the memory object of a data item or procedure was mapped via the request **vm_map** and therefore bears a SID derived from the SID of the memory object. Processes are represented by tasks. The task associated with a process is denoted by $process\_task(process)$.

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\textit{CwToDtos}} \\
\textit{State} \\
\textit{MkAddressSpace} \\
\textit{MkLabels} \\
di\_memory : DATA\_ITEM \mathrel{\mkern1mu+\mkern-14mu\rightarrowtail} MEMORY \\
proc\_memory : PROCEDURE \mathrel{\mkern1mu+\mkern-14mu\rightarrowtail} MEMORY \\
process\_task : CW\_PROCESS \mathrel{\mkern1mu+\mkern-14mu\rightarrowtail} TASK \\
\hline
\mathrm{dom}\, di\_memory = data\_items \\
\mathrm{dom}\, proc\_memory = procedures \\
\mathrm{dom}\, process\_task = processes \\
\mathrm{ran}\, di\_memory \subseteq \mathrm{dom}\, memory\_osi \\
\mathrm{ran}\, proc\_memory \subseteq \mathrm{dom}\, memory\_osi \\
\mathrm{ran}\, process\_task \subseteq \mathrm{dom}\, task\_osi \\
\mathsf{disjoint}\, \langle \mathrm{ran}\, di\_memory, \mathrm{ran}\, proc\_memory \rangle \\
\forall\, region : REGION \\
\mid mapped\_memory(region) \in \mathrm{ran}\, di\_memory \cup \mathrm{ran}\, proc\_memory \\
\bullet\; region\_osi(region) \\
\qquad = memory\_osi\_region\_osi(memory\_osi(mapped\_memory(region))) \\
\hline
\end{array}
$$

We will assume that a TP is executed through requests to the operating system to create a process and load and execute the executable code for the TP. Furthermore, CDI access is controlled through requests to the memory manager which communicates with the DTOS microkernel. The actions of the microkernel will generate requests to the Security Server.

9.3 Clark-Wilson Security Server

In this section we describe a Security Server that enforces the above security policy.

9.3.1 Security Database

9.3.1.1 Security Contexts We first define the security contexts. The set of object security contexts is CW_OSC. An object security context may have associated with it either a $PROCEDURE$ or a $DATA\_ITEM$. These are obtained by the functions osc_proc and osc_di. Note that these mappings are invertible.

$$[\, CW\_OSC \,]$$

$$\begin{array}{|l}
osc_proc : CW_OSC \rightarrowtail\!\!\!\!\!\!+\!\!\!\!\rightarrow PROCEDURE \\
osc_di : CW_OSC \rightarrowtail\!\!\!\!\!\!+\!\!\!\!\rightarrow DATA_ITEM \\
\hline
\mathsf{disjoint}\ \langle \mathrm{dom}\ osc_proc, \mathrm{dom}\ osc_di \rangle
\end{array}$$

Subject security contexts are associated with both processes and individuals (e.g., login processes). A subject security context contains the following three components: the ind, the $procedure$ and the $index$. The ind provides a record of the individual on whose behalf the subject is executing. In the following we will constrain the creation of subjects so that when an individual or process creates a new process, the ind of the newly created process is identical to that of the creating individual or process. (Note that this does not apply to the creation of a new individual by a privileged process.) The $procedure$ indicates the procedure executed by the subject. For an individual, the $procedure$ component of the security context might be, for example, the procedure used for a login shell. The $index$ is included to allow a single ind to run several processes executing the same procedure at the same time, each with a distinct subject context.

$$\begin{array}{|l}
\multicolumn{1}{l}{CwSsc} \\
\hline
ind : INDIVIDUAL \\
procedure : PROCEDURE \\
index : \mathbb{N} \\
\hline
\end{array}$$

The set $Cw\_privileged\_contexts$ denotes a set of privileged subject contexts. These contexts may have permission to create a process in a context with a different ind. It is assumed that this context will be reserved for tasks such as the login task that must start individual processes in their correct context.

$$\begin{array}{|l}
Cw_privileged_contexts : \mathbb{P}\ CwSsc
\end{array}$$

We define cw_sscs and cw_oscs to be the existing subject and object security contexts.

$$\begin{array}{|l}
\multicolumn{1}{l}{CwContexts} \\
\hline
cw_sscs : \mathbb{P}\ CwSsc \\
cw_oscs : \mathbb{P}\ CW_OSC \\
\hline
\end{array}$$

Each subject security identifier is mapped by the function ssi_ssc to a subject security context, and each object identifier is mapped by osi_osc to an object context. The function osi_ssc maps an object SID to a subject security context. This is used when a subject is being operated upon.

$$
\begin{array}{l}
\textit{CwSidToContext} \\
\hline
\textit{CwContexts} \\
ssi\_ssc : SSI \pinj CwSsc \\
osi\_osc : OSI \pinj CW\_OSC \\
osi\_ssc : OSI \pfun CwSsc \\
\hline
\operatorname{ran} ssi\_ssc \subseteq cw\_sscs \\
\operatorname{ran} osi\_osc \subseteq cw\_oscs \\
\operatorname{ran} osi\_ssc \subseteq cw\_sscs
\end{array}
$$

We make the following assumptions regarding the labeling of memory and tasks:

- Every OSI used to label memory has an associated OSC.
- Every SSI used to label a task has an associated SSC.
- For every $d \in data\_items$, the OSI used to label the memory associated with d maps to an OSC associated with d by the security server.
- For every $p \in procedures$, the OSI used to label the memory associated with p maps to an OSC associated with p by the security server.

If we had a more complete model of the system including the file server, the ways in which individuals may log on to the system and the ways in which the DTOS kernel assigned SIDs, these assumptions would be justified in terms of that model.

$$
\begin{array}{l}
\textit{CwConsistentLabels} \\
\hline
\textit{CwSidToContext} \\
\textit{CwToDtos} \\
\hline
\operatorname{ran} memory\_osi \subseteq \operatorname{dom} osi\_osc \\
\operatorname{ran} task\_ssi \subseteq \operatorname{dom} ssi\_ssc \\
\forall d : data\_items; p : procedures \\
\bullet (osi\_osc(memory\_osi(di\_memory(d))), d) \in osc\_di \\
\quad \land (osi\_osc(memory\_osi(proc\_memory(p))), p) \in osc\_proc
\end{array}
$$

Paralleling the MLS/TE section, we do not assume consistency for $process\_individual(p)$. Rather we will eventually justify this property based upon our interpretation of $process\_individual$ together with the policy requirements. For any process p we will interpret $process\_individual(p)$ to be

- the $INDIVIDUAL$ most recently assigned to p by a privileged process if such an assignment has occurred, or
- $process\_individual(p)$ where p is the parent process and is not privileged.

Of course, the first process created in the system (e.g., a bootstrap task) must be treated as a special case. Its individual is assumed to be the system itself. This interpretation means that we trust privileged processes to correctly assign the individual of any processes they create. Any other process may only create processes that have the same individual.

9.3.1.2 Policy Representation

Policy decisions in the Clark-Wilson security server are sensitive to the history of permissions that have already been granted, so the Security Server must maintain a history. For example, assume a transformation procedure TP_1 is certified to manipulate the following sets of CDIs:

$$\{\{CDI_1, CDI_3\}, \{CDI_2, CDI_3\}, \{CDI_2, CDI_4\}\}$$

Initially, TP_1 could obtain $Have\_write$ permission to any of $CDI_1, \ldots, CDI_4$. However, if TP_1 is granted $Have\_write$ permission to CDI_2, then the policy should no longer grant $Have\_write$ permission to CDI_1 since there is no set above that contains both CDI_1 and CDI_2.

We introduce the schema $CwHistory$ to represent the history information maintained by the Security Server. The expression $have\_writes\_so\_far(ssc)$ denotes the set of all data items to which the tasks in subject context ssc have been given $Have\_write$ permission. For convenience, we define the relation $have\_writes\_so\_far\_rel$ so that $(ssc, di) \in have\_writes\_so\_far\_rel$ exactly when $di \in have\_writes\_so\_far(ssc)$.

```
── CwHistory ──────────────────────────────────────
│ CwContexts
│ have_writes_so_far : CwSsc ⟶ ℙ DATA_ITEM
│ have_writes_so_far_rel : CwSsc ⟷ DATA_ITEM
├──────────
│ dom have_writes_so_far_rel ⊆ cw_sscs
│ ∀ ssc : CwSsc
│ • have_writes_so_far_rel⦇{ssc}⦈ = have_writes_so_far(ssc)
└──────────────────────────────────────────────────
```

Finally, the data maintained by the Clark-Wilson Security Server includes the sets of defined subject and object security contexts, the mappings from SIDs to contexts, the Clark-Wilson policy data in $DataItem$, $TransformationProcedure$ and $Individual$, the history data and other miscellaneous data, $other\_cw\_data$, from which the generic Security Server information is extracted.

$$[\,OTHER\_CW\_DATA\,]$$

```
── CwData ─────────────────────────────────────────
│ CwSidToContext
│ CwHistory
│ DataItem
│ TransformationProcedure
│ Individual
│ other_cw_data : OTHER_CW_DATA
└──────────────────────────────────────────────────
```

9.3.2 Permission Requirements

In this section we state constraints on the permissions that are in $policy\_allows$. These constraints will be used below in defining the Clark-Wilson Security Server state. We must constrain the policy so that it enforces the rules **E1**–**E4** of the Clark-Wilson policy.

9.3.2.1 Valid SIDs Requirement

The following policy requirement is used only to guarantee well-definedness of several of the following requirements.

Policy Requirement 9.1 : No permissions are given if the SSI is not in $\mathrm{dom}\, ssi\_ssc$ or if the OSI is not in $\mathrm{dom}\, osi\_osc$.

$$
\begin{array}{|l}
\textit{ValidSids} \\
\hline
SsPolicyAllows[CwData, D\_SS\_REQ, D\_ANS] \\
\hline
\forall\, ss\_req : policy\_allows;\ preq : PermReq \\
\mid Perm\_req(preq) = ss\_req \\
\bullet\ (preq.ssi) \in \mathrm{dom}(ss\_data.ssi\_ssc) \\
\land\ (preq.osi) \in \mathrm{dom}(ss\_data.osi\_osc) \\
\hline
\end{array}
$$

9.3.2.2 SID Assignment Permission Requirements

Policy Requirement 9.2 : $Make\_sid$ permission is prohibited.

$$
\begin{array}{|l}
\textit{CwMakeSid} \\
\hline
SsPolicyAllows[CwData, D\_SS\_REQ, D\_ANS] \\
\hline
\forall\, ss\_req : policy\_allows;\ preq : PermReq \\
\mid Perm\_req(preq) = ss\_req \\
\bullet\ preq.perm \neq Make\_sid \\
\hline
\end{array}
$$

Policy Requirement 9.3 : $Cross\_context\_create$ permission to an OSI (for a subject being acted upon) is given to an SSI only if both SIDs map to subject contexts with the same ind field or the SSI identifies a privileged context.

$$
\begin{array}{|l}
\textit{CorrectUser} \\
\hline
SsPolicyAllows[CwData, D\_SS\_REQ, D\_ANS] \\
ValidSids \\
\hline
\forall\, ss\_req : policy\_allows;\ preq : PermReq \\
\mid Perm\_req(preq) = ss\_req \\
\quad \land\ preq.perm = Cross\_context\_create \\
\bullet\ (ss\_data.ssi\_ssc)(preq.ssi) \in Cw\_privileged\_contexts \\
\quad \lor\ ((preq.osi) \in \mathrm{dom}(ss\_data.osi\_ssc) \\
\qquad \land\ ((ss\_data.ssi\_ssc)(preq.ssi)).ind \\
\qquad\quad = ((ss\_data.osi\_ssc)(preq.osi)).ind) \\
\hline
\end{array}
$$

Clark-Wilson is implemented most naturally with an assumption that each task has a unique context. However, this is not strictly necessary. The security server defined here controls the actions of tasks based upon their context. If two tasks share the same context, this constrains them more than if their contexts are distinct. In effect the two tasks would be considered a single process to be controlled by Clark-Wilson.

The following two optional requirements can be used to ensure a one-to-one mapping between tasks and contexts. They would most likely be paired with the provision of an information request to ask for unused SIDs. These constraints are not essential in the subsequent proofs. (We do use **PR9.4** in one proof, but we could instead complete that proof by arguing that the $Task\_create\_id$ operation can only create a task with the same context as the parent task.)

Editorial Note:

These two requirements are stronger than strictly needed for one-to-oneness in Clark-Wilson. They could be relaxed to apply only to creation of tasks for TP execution. We state them in this way as a simplification.

Policy Requirement 9.4 : $Create\_task$ permission is prohibited.

$$
\begin{array}{l}
\textit{NoCreateTask} \\
\hline
SsPolicyAllows[CwData, D\_SS\_REQ, D\_ANS] \\
\hline
\forall ss\_req : policy\_allows; preq : PermReq \\
\mid Perm\_req(preq) = ss\_req \\
\bullet\ preq.perm \neq Create\_task
\end{array}
$$

Policy Requirement 9.5 : $Cross\_context\_create$ permission to an OSI (for a subject being acted upon) is given only if the OSI maps to an unused subject context.

$$
\begin{array}{l}
\textit{NewSid} \\
\hline
SsPolicyAllows[CwData, D\_SS\_REQ, D\_ANS] \\
\hline
\forall ss\_req : policy\_allows; preq : PermReq \\
\mid Perm\_req(preq) = ss\_req \\
\quad \land preq.perm = Cross\_context\_create \\
\bullet\ (preq.osi) \in \mathrm{dom}(ss\_data.osi\_ssc) \\
\quad \land (ss\_data.osi\_ssc)(preq.osi) \notin (ss\_data.cw\_sscs)
\end{array}
$$

9.3.2.3 $Have\_execute$ Permission Requirements

Policy Requirement 9.6 : $Have\_execute$ permission to an OSI, osi, is granted to ssi only if ssi maps to a context that has the object context of osi as its $procedure$ component.

$$
\begin{array}{l}
\textit{CorrectProcedure} \\
\hline
SsPolicyAllows[CwData, D\_SS\_REQ, D\_ANS] \\
ValidSids \\
\hline
\forall ss\_req : policy\_allows; preq : PermReq \\
\mid Perm\_req(preq) = ss\_req \\
\quad \land preq.perm = Have\_execute \\
\bullet\ ((ss\_data.osi\_osc)(preq.osi)) \in \mathrm{dom}\ osc\_proc \\
\quad \land osc\_proc((ss\_data.osi\_osc)(preq.osi)) \\
\qquad = ((ss\_data.ssi\_ssc)(preq.ssi)).procedure
\end{array}
$$

Policy Requirement 9.7 : $Have\_execute$ permission to a TP's OSI is granted to ssi only if ssi maps to a context with a ind field that is not the certifier.

$$
\begin{array}{l}
\textit{NoExecutionByCertifier} \\
\hline
SsPolicyAllows[CwData, D\_SS\_REQ, D\_ANS] \\
ValidSids \\
\hline
\forall ss\_req : policy\_allows; preq : PermReq \\
\mid Perm\_req(preq) = ss\_req \\
\quad \land preq.perm = Have\_execute \\
\quad \land osc\_proc((ss\_data.osi\_osc)(preq.osi)) \in ss\_data.transformation\_procedures \\
\bullet\ ((ss\_data.ssi\_ssc)(preq.ssi)).ind \neq (ss\_data.certifier)
\end{array}
$$

Policy Requirement 9.8 : $Have\_execute$ permission to a TP's OSI is granted to ssi only if ssi maps to a context with a ind field that is in the set $authenticated\_individuals$.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{UnauthenticatedIndividual}} \\
SsPolicyAllows[CwData, D\_SS\_REQ, D\_ANS] \\
ValidSids \\
\hline
\forall\, ss\_req : policy\_allows;\ preq : PermReq \\
\mid Perm\_req(preq) = ss\_req \\
\quad \land\ preq.perm = Have\_execute \\
\quad \land\ osc\_proc((ss\_data.osi\_osc)(preq.osi)) \in ss\_data.transformation\_procedures \\
\bullet\ ((ss\_data.ssi\_ssc)(preq.ssi)).ind \in (ss\_data.authenticated\_individuals) \\
\hline
\end{array}
$$

Multiple threads within a task are allowed here. However, they would all be required to execute procedures with the same TP SID, and they would all have the same history. Thus, if $thread_1$ is granted $Have\_write$ permission to a CDI C, so is its sister thread $thread_2$, and this might affect the CDIs to which $thread_2$ is granted $Have\_write$ permission.

9.3.2.4 $Have\_write$ Permission Requirements

Policy Requirement 9.9 : $Have\_write$ permission for a CDI's OSI, osi, is granted to ssi only if the data item to which osi maps, together with the set of data items associated with all previous CDI $Have\_write$ permissions for ssi, would be consistent with one of the sets in $may\_manipulate$ for the $procedure$ component of the context to which ssi is mapped.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{ManipulationAllowed}} \\
SsPolicyAllows[CwData, D\_SS\_REQ, D\_ANS] \\
ValidSids \\
\hline
\forall\, ss\_req : policy\_allows;\ preq : PermReq; \\
\quad the\_ssc : CwSsc;\ the\_di : DATA\_ITEM \\
\mid Perm\_req(preq) = ss\_req \\
\quad \land\ preq.perm = Have\_write \\
\quad \land\ the\_ssc = (ss\_data.ssi\_ssc)(preq.ssi) \\
\quad \land\ the\_di = osc\_di((ss\_data.osi\_osc)(preq.osi)) \\
\quad \land\ the\_di \in ss\_data.constrained\_data\_items \\
\bullet\ (\exists\, S : (ss\_data.may\_manipulate)(\!|\{the\_ssc.procedure\}|\!) \\
\quad \bullet\ \{the\_di\} \cup (ss\_data.have\_writes\_so\_far)(the\_ssc) \subseteq S) \\
\hline
\end{array}
$$

Policy Requirement 9.10 : $Have\_write$ permission for a CDI's OSI, osi, is granted to ssi only if the data item to which osi maps, together with the set of data items associated with all previous CDI $Have\_write$ permissions for ssi, would be consistent with one of the sets in $may\_execute$ for the ind and $procedure$ components of the context to which ssi is mapped.

$$
\begin{array}{l}
\textit{ExecutionAllowed} \\
\hline
SsPolicyAllows[CwData, D\_SS\_REQ, D\_ANS] \\
ValidSids \\
\hline
\forall ss\_req : policy\_allows;\ preq : PermReq; \\
\quad the\_ssc : CwSsc;\ the\_di : DATA\_ITEM \\
\mid Perm\_req(preq) = ss\_req \\
\quad \wedge\ preq.perm = Have\_write \\
\quad \wedge\ the\_ssc = (ss\_data.ssi\_ssc)(preq.ssi) \\
\quad \wedge\ the\_di = osc\_di((ss\_data.osi\_osc)(preq.osi)) \\
\quad \wedge\ the\_di \in ss\_data.constrained\_data\_items \\
\bullet\ the\_ssc.ind \in \mathrm{dom}(ss\_data.may\_execute) \\
\quad \wedge\ (\exists S : ((ss\_data.may\_execute)(the\_ssc.ind))(\!|\{the\_ssc.procedure\}|\!) \\
\qquad \bullet\ \{the\_di\} \cup (ss\_data.have\_writes\_so\_far)(the\_ssc) \subseteq S)
\end{array}
$$

9.3.3 Security Server State

The Clark-Wilson Security Server combines the general properties of the DTOS Security Server with the history information and the mappings from Clark-Wilson entities to security identifiers and contexts. The $policy\_allows$ must satisfy all constraints in the previous section.

$$
\begin{array}{l}
\textit{CwState} \\
\hline
SsState[MkRequest, CwData, D\_SS\_REQ, D\_RESP, D\_ANS] \\
ValidSids \\
CwMakeSid \\
NoCreateTask \\
NewSid \\
CorrectUser \\
CorrectProcedure \\
NoExecutionByCertifier \\
UnauthenticatedIndividual \\
ManipulationAllowed \\
ExecutionAllowed \\
\hline
\end{array}
$$

9.4 Operations

In sending a response to a security server request, the history data might need to be changed. In particular, if $Have\_write$ permission is granted in the response, this must be recorded in $have\_writes\_so\_far$. If $Cross\_context\_create$ permission granted, the context of the task to be created must be stored in cw_sscs. Given the above constraints on $policy\_allows$, the policy might change to remain consistent with the history data. We define a schema $CwUpdateHistory$ to describe the changes to the history. Note that we allow at most one (ssc, di) pair to be added to $have\_writes\_so\_far\_rel$ in any transition.[26]

[26] This is not essential for Clark-Wilson. However, if this assumption is not made then we would have to include in our specifications of $CwSendNegativeResponse$ and $CwSendAffirmativeResponse$ constraints similar to those in $ManipulationAllowed$ and $ExecutionAllowed$. In this case the constraints would be stated in terms of granted permissions rather than the contents of $policy\_allows$, and they would consider all additional $Have\_write$ permissions. This would be necessary since there might be multiple data items for which $Have\_write$ could be granted without violating Clark-Wilson constraints, and yet if $Have\_write$ is granted for some combination of these we might violate the high-level policy. We also note that this requirement does not really add anything new for Clark-Wilson together

$$
\begin{array}{l}
\text{\_\_ } \mathit{CwUpdateHistory} \text{ \_\_\_\_} \\
\mathit{SsSendResponseAux}[\mathit{MkRequest}, \mathit{CwData}, \mathit{D\_SS\_REQ}, \mathit{D\_RESP}, \mathit{D\_ANS}] \\
\hline
\mathit{ss\_data}'.\mathit{have\_writes\_so\_far\_rel} = \mathit{ss\_data}.\mathit{have\_writes\_so\_far\_rel} \\
\quad \cup \{\mathit{the\_ssc} : \mathit{CwSsc}; \mathit{the\_di} : \mathit{DATA\_ITEM} \\
\qquad \mid (\exists \mathit{preq} : \mathit{Perm\_req}^{\sim} (\!| \mathit{grants}(\mathit{ss\_response}?) |\!) \\
\qquad\quad \bullet (\mathit{preq}.\mathit{perm} = \mathit{Have\_write} \\
\qquad\qquad \land (\mathit{ss\_data}.\mathit{ssi\_ssc})(\mathit{preq}.\mathit{ssi}) = \mathit{the\_ssc} \\
\qquad\qquad \land \mathit{osc\_di}((\mathit{ss\_data}.\mathit{osi\_osc})(\mathit{preq}.\mathit{osi})) = \mathit{the\_di} \\
\qquad\qquad \land \mathit{the\_di} \in \mathit{ss\_data}.\mathit{constrained\_data\_items}))\} \\
\\
\#(\mathit{ss\_data}'.\mathit{have\_writes\_so\_far\_rel} \setminus \mathit{ss\_data}.\mathit{have\_writes\_so\_far\_rel}) \leq 1 \\
\\
\mathit{ss\_data}'.\mathit{cw\_sscs} = \mathit{ss\_data}.\mathit{cw\_sscs} \\
\quad \cup \{\mathit{the\_ssc} : \mathit{CwSsc} \\
\qquad \mid (\exists \mathit{preq} : \mathit{Perm\_req}^{\sim} (\!| \mathit{grants}(\mathit{ss\_response}?) |\!) \\
\qquad\quad \bullet \mathit{preq}.\mathit{perm} = \mathit{Cross\_context\_create} \\
\qquad\qquad \land (\mathit{ss\_data}.\mathit{osi\_ssc})(\mathit{preq}.\mathit{osi}) = \mathit{the\_ssc})\} \\
\hline
\end{array}
$$

We include $\mathit{CwUpdateHistory}$ in the definitions of $\mathit{CwSendNegativeResponse}$ and $\mathit{CwSendAffirmativeResponse}$.

$$
\begin{array}{l}
\mathit{CwSendNegativeResponse} \\
\quad \mathrel{\widehat{=}} \mathit{SsSendNegativeResponse}[\mathit{MkRequest}, \mathit{CwData}, \mathit{D\_SS\_REQ}, \mathit{D\_RESP}, \mathit{D\_ANS}] \\
\qquad \land \mathit{CwUpdateHistory} \\
\mathit{CwSendAffirmativeResponse} \\
\quad \mathrel{\widehat{=}} \mathit{SsSendAffirmativeResponse}[\mathit{MkRequest}, \mathit{CwData}, \mathit{D\_SS\_REQ}, \mathit{D\_RESP}, \mathit{D\_ANS}] \\
\qquad \land \mathit{CwUpdateHistory}
\end{array}
$$

To the remaining generic Security Server operations we add the requirement that the history information remain unchanged. This requirement is formalized by $\mathit{CwHistoryInvariant}$.

$$
\begin{array}{l}
\text{\_\_ } \mathit{CwHistoryInvariant} \text{ \_\_\_\_} \\
\mathit{SsStep}[\mathit{MkRequest}, \mathit{CwData}, \mathit{D\_SS\_REQ}, \mathit{D\_RESP}, \mathit{D\_ANS}] \\
\hline
\mathit{ss\_data}'.\mathit{have\_writes\_so\_far\_rel} = \mathit{ss\_data}.\mathit{have\_writes\_so\_far\_rel} \\
\mathit{ss\_data}'.\mathit{cw\_sscs} = \mathit{ss\_data}.\mathit{cw\_sscs} \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\mathit{CwReceiveRequest} \\
\quad \mathrel{\widehat{=}} \mathit{SsReceiveRequest}[\mathit{MkRequest}, \mathit{CwData}, \mathit{D\_SS\_REQ}, \mathit{D\_RESP}, \mathit{D\_ANS}] \\
\qquad \land \mathit{CwHistoryInvariant} \\
\mathit{CwMgrRequest} \\
\quad \mathrel{\widehat{=}} \mathit{SsMgrRequest}[\mathit{MkRequest}, \mathit{CwData}, \mathit{D\_SS\_REQ}, \mathit{D\_RESP}, \mathit{D\_ANS}] \\
\qquad \land \mathit{CwHistoryInvariant} \\
\mathit{CwInternalTransition} \\
\quad \mathrel{\widehat{=}} \mathit{SsInternalTransition}[\mathit{MkRequest}, \mathit{CwData}, \mathit{D\_SS\_REQ}, \mathit{D\_RESP}, \mathit{D\_ANS}] \\
\qquad \land \mathit{CwHistoryInvariant}
\end{array}
$$

with the DTOS kernel since all permissions granted in a $\mathit{D\_RESP}$ must be for the same SID pair.

9.5 Component Specification

We can use the above to specify the Clark-Wilson security server as a component.

The **guar** for the Clark-Wilson security server allows any of the transitions described in Section 9.4. This is modeled by cw_guar.

$$
\begin{array}{l}
CwGuarStep \\
\mathrel{\widehat=} CwReceiveRequest \\
\quad \lor CwSendNegativeResponse \\
\quad \lor CwSendAffirmativeResponse \\
\quad \lor CwMgrRequest \\
\quad \lor CwInternalTransition
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{CwGuar} \\
cw\_guar : \mathbb{P}\ CwGuarStep \\
\hline
cw\_guar = CwGuarStep \\
\hline
\end{array}
$$

The Clark-Wilson security server assumes that the assumptions of generic security servers in $SsRely$ are satisfied.

$$
\begin{array}{|l}
\multicolumn{1}{l}{CwRely} \\
cw\_rely : \mathbb{P}\ \Delta\ CwState \\
SsRely[MkRequest, CwData, D\_SS\_REQ, D\_RESP, D\_ANS] \\
\hline
cw\_rely = ss\_rely \\
\hline
\end{array}
$$

The set of allowed initial states is modeled by cw_init. We require that an initial state satisfy the constraints imposed by $SsInit$ and that $have\_writes\_so\_far$ be consistent with constraints on writing expressed in $may\_manipulate$ and $may\_execute$. Note that $have\_writes\_so\_far$ may be non-empty initially to reflect permissions that are "retained" in the kernel at system startup to allow successful booting of the system. If it is necessary to have initially "retained" permissions that are inconsistent with $may\_manipulate$ or $may\_execute$, then it may not be possible to combine the Clark-Wilson server and the DTOS kernel to implement the high-level policy. This is reflected by the hypotheses of the theorems below.

$$
\begin{array}{|l}
\multicolumn{1}{l}{CwInit} \\
SsInit[MkRequest, CwData, D\_SS\_REQ, D\_RESP, D\_ANS] \\
cw\_init : \mathbb{P}\ CwState \\
\hline
cw\_init \subseteq ss\_init \\
\forall st : cw\_init;\ ssc : CwSsc \\
\bullet (\exists S : (st.ss\_data.may\_manipulate)(\!|\{ssc.procedure\}|\!) \\
\qquad\quad \bullet st.ss\_data.have\_writes\_so\_far(ssc) \subseteq S) \\
\qquad \land (\exists S : ((st.ss\_data.may\_execute)(ssc.ind))(\!|\{ssc.procedure\}|\!) \\
\qquad\quad \bullet (st.ss\_data.have\_writes\_so\_far)(ssc) \subseteq S) \\
\hline
\end{array}
$$

All information in $CwState$ is visible to the Clark-Wilson security server.

$$\begin{array}{|l}
\textit{CwView} \\
\hline
cw_view : CwState \leftrightarrow CwState \\
\hline
\forall st_1, st_2 : CwState \\
\bullet (st_1, st_2) \in cw_view \Leftrightarrow st_1 = st_2 \\
\hline
\end{array}$$

$$CwComponent \\ \quad \widehat{=}\ CwGuar \land CwRely \land CwInit \land CwView$$

We now prove several lemmas regarding the Clark-Wilson security server. These will be used in the next section to show that the Clark-Wilson server when composed with the DTOS microkernel satisfies the Clark-Wilson requirements **E1**–**E4**. Since there is a one-to-one mapping between SIDs and security contexts, we will frequently use these concepts interchangeably. For example, we will sometimes refer to SIDs when discussing $policy\_allows$, and at other times refer to contexts.

Lemma 9.1 *If d is added to $have\_writes\_so\_far(s)$ in transition (t, t') for some $CwSsc$ s, then in state t $policy\_allows$ contains $Have\_write$ permission from s to $osc\_di^{\sim}(d)$, and d is in $constrained\_data\_items$.*

Proof: The component definition for the Clark-Wilson security server implies that if (s, d) is added to $have\_writes\_so\_far$ then it is during a $CwSendAffirmativeResponse$ or $CwSendNegativeResponse$ transition in which

$$\begin{array}{l}
\exists\, preq : Perm_req^{\sim} (\!| grants(ss_response?) |\!) \\
\bullet (preq.perm = Have_write \\
\quad \land (ss_data.ssi_ssc)(preq.ssi) = s \\
\quad \land osc_di((ss_data.osi_osc)(preq.osi)) = d \\
\quad \land d \in ss_data.constrained_data_items).
\end{array}$$

$CwSendAffirmativeResponse$ and $CwSendNegativeResponse$ require that $grants(ss\_response?) \subseteq policy\_allows$. This completes the proof. □

Lemma 9.2 *In any reachable $CwState$, for any $c \in CwSsc$, there exists*

$$S \in (ss\_data.may\_manipulate)(\!| \{c.procedure\} |\!)$$

such that

$$(ss\_data.have\_writes\_so\_far)(c) \subseteq S$$

Proof: We induct on the length of behavior prefixes. Since the lemma is true for all initial states (see $CwInit$), the base case is satisfied.

We next assume that the lemma holds for all states reachable in n transitions ($n \geq 0$) and show that it also holds for all states reachable in $n + 1$ transitions. Let t be a state reachable in n transitions and t' be a state reachable from t in one transition. Since $ss\_data.may\_manipulate$ is invariant, if the transition (t, t') does not add any elements to the set $(ss\_data.have\_writes\_so\_far)(c)$ for any c, then the lemma is true for t'. Otherwise, let (c, d) be the pair added to $have\_writes\_so\_far\_rel$ in transition (t, t'). (Recall there is at most one such addition in any single transition.) Lemma 9.1 implies that $policy\_allows$ in state t contains $Have\_write$ permission from c to $osc\_di^{\sim}(d)$, and $d \in constrained\_data\_items$.

$ManipulationAllowed$ (**PR9.9**) implies there exists $S \in (ss\_data.may\_manipulate)(\{c.procedure\})$ such that

$$\{d\} \cup t.ss\_data.have\_writes\_so\_far(c) \subseteq S.$$

Since the left hand side equals $t'.ss\_data.have\_writes\_so\_far(c)$, we are done.

□

Lemma 9.3 *In any reachable $CwState$ t, for any $c \in CwSsc$, there exists*

$$S \in ((ss\_data.may\_execute)(c.ind))(\{c.procedure\})$$

such that

$$(ss\_data.have\_writes\_so\_far)(c) \subseteq S$$

The proof of this lemma is essentially the same as for the preceding one with $ExecutionAllowed$ (**PR9.10**) used instead of $ManipulationAllowed$ (**PR9.9**).

9.6 Composing DTOS and Clark-Wilson

In this section, we first compose the DTOS kernel with the Clark-Wilson security server. We then show that this composite system implements the abstract Clark-Wilson policy.

The composite state is $MkCwState$ which contains all the components of $MkState$ and $CwState$.

```
MkCwState
  MkState
  CwState
```

Each kernel state is associated with the set of all $MkCwState$ that have the same value for each of the components of $MkState$. Similarly, each Clark-Wilson security server state is associated with the set of all $MkCwState$ that have the same value for each of the components of $CwState$.

The set of allowed initial states for the composition of two components is the intersection of the two sets (after mapping them into the composite state). This set of states is modeled by mk_cw_init. Since mk_cw_init is nonempty, the kernel and Clark-Wilson security server are composable.

$$
\begin{array}{l}
\textit{MkCwInit} \\
\hline
mk\_cw\_init : \mathbb{P}\ MkCwState \\
\hline
\forall\, st : mk\_cw\_init;\ ssc : CwSsc \\
\bullet\ st.pending\_responses = \varnothing \\
\quad \land\ st.pending\_requests = \varnothing \\
\quad \land\ st.active\_request = \varnothing \\
\quad \land\ st.sent = \varnothing \\
\quad \land\ st.obtained = \varnothing \\
\quad \land\ st.allowed = \varnothing \\
\quad \land\ st.responses = \varnothing \\
\quad \land\ st.active\_computations = \varnothing \\
\quad \land\ (\exists\, S : (st.ss\_data.may\_manipulate)(\!|\{ssc.procedure\}|\!) \\
\qquad \bullet\ st.ss\_data.have\_writes\_so\_far(ssc) \subseteq S) \\
\quad \land\ (\exists\, S : ((st.ss\_data.may\_execute)(ssc.ind))(\!|\{ssc.procedure\}|\!) \\
\qquad \bullet\ (st.ss\_data.have\_writes\_so\_far)(ssc) \subseteq S)
\end{array}
$$

In composing the kernel and Clark-Wilson security server we will use respect relations that require each component to leave alone its peer's internal data.

$$
\begin{array}{l}
\textit{MkCwRespect} \\
\hline
mk\_respect\_cw : \mathbb{P}\ \Delta\ MkCwState \\
cw\_respect\_mk : \mathbb{P}\ \Delta\ MkCwState \\
\hline
mk\_respect\_cw = \{\ \Delta\ MkCwState \\
\quad \mid \Xi\ SsInternals[MkRequest, CwData, D\_SS\_REQ, D\_RESP, D\_ANS]\ \} \\
cw\_respect\_mk = \{\ \Delta\ MkCwState \\
\quad \mid \Xi\ MgrInternals[MkData, MkRequest, D\_SS\_REQ, D\_RESP, D\_ANS]\ \}
\end{array}
$$

The **guar** of the composite is denoted by mk_cw_guar.

$$
\begin{array}{l}
MkCwStep \\
\mathrel{\widehat=} (\Xi\ SsInternals[MkRequest, CwData, D\_SS\_REQ, D\_RESP, D\_ANS] \\
\quad \land\ MkGuarStep) \\
\lor\ (\Xi\ MgrInternals[MkData, MkRequest, D\_SS\_REQ, D\_RESP, D\_ANS] \\
\quad \land\ CwGuarStep)
\end{array}
$$

$$
\begin{array}{l}
\textit{MkCwGuar} \\
\hline
mk\_cw\_guar : \mathbb{P}\ MkCwStep \\
\hline
mk\_cw\_guar = MkCwStep
\end{array}
$$

The **rely** of the composite is the intersection of the two rely relations.

$$
\begin{array}{l}
\textit{MkCwRely} \\
\hline
mk\_cw\_rely : \mathbb{P}\ \Delta\ MkCwState \\
\hline
mk\_cw\_rely = \{\ \Delta\ MkCwState \\
\quad \mid \Xi\ MgrInternals[MkData, MkRequest, D\_SS\_REQ, D\_RESP, D\_ANS] \\
\quad \land\ \Xi\ SsInternals[MkRequest, CwData, D\_SS\_REQ, D\_RESP, D\_ANS] \\
\quad \land\ \Xi\ SharedInterpretation[MkRequest, D\_SS\_REQ, D\_RESP, D\_ANS] \\
\quad \land\ pending\_ss\_requests \sqsubseteq pending\_ss\_requests' \\
\quad \land\ pending\_responses' = pending\_responses\ \}
\end{array}
$$

To show that the Composition Theorem applies to the composite, we must show that

$$mk\_guar \cap mk\_respect\_cw \subseteq cw\_guar \cup cw\_rely \cup cw\_view,$$

and

$$cw\_guar \cap cw\_respect\_mk \subseteq mk\_guar \cup mk\_rely \cup mk\_view.$$

The proof of these properties is essentially identical to the corresponding proof for the composition of the generic manager and security server.

We now demonstrate that the composition of the Clark-Wilson security server and the DTOS microkernel satisfies the high-level requirements of the Clark-Wilson security policy. We first must define our interpretations of $process\_manipulates$ and $process\_procedure$ in terms of the DTOS kernel. These components were not defined in terms of kernel entities in $CwToDtos$ because they do not correspond very cleanly to any component of the DTOS state. Rather, they are best defined in terms of behaviors in the kernel. We use the notation $[b, n]$ to denote the first n transitions of a behavior b. For any process p, $process\_manipulates(p)$ for $[b, n]$ is the set of all $DATA\_ITEM$ d such that $process\_task(p)$ has performed a $Write\_page$ or Vm_write_id operation on $di\_memory(d)$ in $[b, n]$. Similarly, $process\_procedure(p)$ for $[b, n]$ is the $PROCEDURE$ r such that $process\_task(p)$ has performed an $Execute\_page$ operation on $proc\_memory(r)$ in $[b, n]$.[27]

Editorial Note:
An alternative approach for interpreting $process\_manipulates$ and $process\_procedure$ would be to define new components in the composite state that are visible to neither the kernel nor the security server and to use the respect relations to constrain the changes to these components with respect to the visible components. For example, we could define $task\_manipulates$ as an invisible component mapping tasks to memory objects. The respect relations would constrain $task\_manipulates$ so that it records the writing that each task has done. There are drawbacks to this approach. In the terminology of the DTOS Composability Study, the composite would not be proper. That is, its transitions would be constrained by information that is not in its view. All the interesting properties of composition (e.g., the Composition Theorem, commutativity and associativity) still apply for improper components. However, the $view$ in an improper component does not correspond well to one's intuition. Further study would be necessary to determine if this alternative is appropriate. Perhaps the implications of this could be studied in future composability work.

In proving that the high level policy is satisfied by the implementation described in this section we could take at least three different approaches. In one approach, we could attempt to use the DTOS Consistency Theorem. In order to apply this theorem it would be necessary to ignore the optional policy requirement **PR9.5** since this causes a granted permission to become ungrantable. (In this case we would probably also ignore **PR9.4** since it serves little purpose in the absence of **PR9.5**.) This means that we would allow a Clark-Wilson process to consist of several "logical tasks" all with the same SID.

A second approach would involve proving a *partial consistency* theorem that could be used to prove consistency for some SS_REQs but not for others. If we chose to keep requirement **PR9.5**, then we could use the partial consistency theorem to demonstrate consistency with respect to permissions other than $Cross\_context\_create$, but could not use it to reason about $Cross\_context\_create$. Although we have not explored this approach, it appears likely that it would work. The proof in this case would be fairly similar to that for MLS/TE with the main difference being that, since the policy is dynamic, we must work harder to show that no granted permission (other than $Cross\_context\_create$) ever becomes ungrantable.

[27] For simplicity we assume there is only one such procedure. The security server really only requires that each procedure executed have the correct context relative to the context of the process executing it. The one-to-one correspondence can thus be obtained by labeling the procedures so that each has a different SID.

However, since this is a research report we will instead follow a third approach in which the two optional policy requirements may be kept if desired. We select this approach primarily because it demonstrates an alternative technique for proving satisfaction of a high-level policy without using consistency. We expect this technique to be useful for history-sensitive policies. The technique relies upon the use-of-permission assumption described in Section 6.3. The security server maintains a history of the policy-relevant actions it believes the kernel has performed. This history affects decisions of the security server regarding the granting and volunteering of permissions. The proof has two primary steps:

1. Show that in every valid state the set of policy-relevant actions the security server believes the kernel has taken is allowed by the high-level policy.
2. Show that at every point in any system behavior the set of all policy-relevant actions that the kernel has taken is a subset of the set of policy-relevant actions the security server believes the kernel has taken.

We cannot perform such a proof in general for the current framework because it does not require that the security server maintain a history.

As an example, consider the proof of requirement **E1** of the Clark-Wilson policy. The relevant history information is stored in $have\_writes\_so\_far$. The first step amounts to showing that if the system has only written CDIs as described by $have\_writes\_so\_far$ then **E1** is satisfied. This is essentially Lemma 9.2. Intuitively, the second step means showing that $process\_manipulates(p)$ is a "subset" of $have\_writes\_so\_far(p)$. This is shown in the following lemma:

Definition 9.4 *Let p be a process and s a CwSsc such that $s = task\_ssi(process\_task(p))$. If the "retention" of $Have\_write$ permission for s to $memory\_osi(di\_memory(e))$ in the initial state implies $have\_writes\_so\_far\_rel$ contains $(ssi\_ssc(s), e)$ in the initial state then we will say the system is **initially Have_write consistent for** p.*

Lemma 9.5 *Let $[b, n]$ be a behavior prefix, p a process and s a CwSsc such that $s = ssi\_ssc(task\_ssi(process\_task(p)))$. Assume that the system is initially $Have\_write$ consistent for p. Then*

$$process\_manipulates(p) \subseteq st.ss\_data.have\_writes\_so\_far(s)$$

where st is the state resulting from $[b, n]$.

Proof: Let d be an arbitrary element of $process\_manipulates(p)$. Then, there exists a transition (t_1, t_1') in $[b, n]$ during which $process\_task(p)$ performs a $Write\_page$ or Vm_write_id operation on $di\_memory(d)$. $Have\_write$ is required by mgr_policy from $task\_ssi(process\_task(p))$ to $memory\_osi(di\_memory(d))$ for both of these operations. Call this permission requirement r.

Note that mgr_policy is static, and therefore monotone non-increasing, for the DTOS kernel. Also, the Clark-Wilson security server does not volunteer any permissions. Therefore, Corollary 5.7 implies that either

1. r is retained in the initial state, or
2. there is a state t_2 preceding t_1 such that in t_2 there is an $ss\_response? \sqsubseteq pending\_responses$ where $(r, ans\_yes) \in answers(ss\_response?)$ (i.e., $r \in grants(ss\_response?)$).

In the first case, our hypotheses indicate that (s, d) is in $have\_writes\_so\_far\_rel$ in the initial state. Since $have\_writes\_so\_far\_rel$ is monotone non-decreasing, $(s, d) \in have\_writes\_so\_far\_rel$ in state st.

Now consider the second case. The initial state constraints require $pending\_responses = \emptyset$ in the system initial state. The **rely** for the composite implies that $pending\_responses$ is not changed by the environment of the composite. Inspection of the manager operations shows that the manager does not add items to $pending\_responses$. Thus, $ss\_response?$ must have been added to $pending\_responses$ by some prior security server transition T. Examination of the Clark-Wilson security server transitions indicates that T must add the pair (s, d) to $have\_writes\_so\_far\_rel$. Since $have\_writes\_so\_far\_rel$ is monotone non-decreasing, that pair must also be in $have\_writes\_so\_far\_rel$ in state st. □

Definition 9.6 *Let p be a process and s a CwSsc such that $s = task\_ssi(process\_task(p))$. If the "retention" of $Have\_execute$ permission for s to m in the initial state implies $policy\_allows$ in the initial state contains $Have\_execute$ permission from s to m then we will say the system is* ***initially Have_execute consistent for*** p.

Lemma 9.7 *Let st be the state resulting from a behavior prefix $[b, n]$, let p be a process, and take $r = process\_procedure(p)$ and $c = ssi\_ssc(task\_ssi(process\_task(p)))$. Assume that the system is initially $Have\_execute$ consistent for p. Then $c.procedure = r$*

Proof: From the interpretation of $process\_procedure(p)$ for $[b, n]$ we find that $process\_task(p)$ has performed an $Execute\_page$ operation on $proc\_memory(r)$ in $[b, n]$. $Have\_execute$ is required by mgr_policy from $task\_ssi(process\_task(p))$ to $memory\_osi(proc\_memory(r))$ for this operation. Call this permission requirement q.

Note that mgr_policy is static, and therefore monotone non-increasing, for the DTOS kernel. Also, the Clark-Wilson security server does not volunteer any permissions. Therefore, Corollary 5.7 implies that either

1. q is retained in the initial state, or
2. there is a state t_2 preceding t_1 such that in t_2 there is an $ss\_response? \sqsubseteq pending\_responses$ where $(q, ans\_yes) \in answers(ss\_response?)$ (i.e., $q \in grants(ss\_response?)$).

In the first case, our hypotheses indicate that $policy\_allows$ in the initial state contains $Have\_execute$ permission from $task\_ssi(process\_task(p))$ to $memory\_osi(proc\_memory(r)))$. **PR9.6** ($CorrectProcedure$) then requires that

$$osc\_proc(osi\_osc(memory\_osi(proc\_memory(r)))) = c.procedure.$$

According to the constraints in $CwConsistentLabels$ the left hand side equals r.

Now consider the second case. The initial state constraints require $pending\_responses = \emptyset$ in the system initial state. The **rely** for the composite implies that $pending\_responses$ is not changed by the environment of the composite. Inspection of the manager operations shows that the manager does not add items to $pending\_responses$. Thus, $ss\_response?$ must have been added to $pending\_responses$ by some prior security server transition T. Theorem 5.8 implies that $q \in policy\_allows$ in the start state of transition T. We then follow the same reasoning as in the first case to conclude that $c.procedure = r$. □

Theorem 9.8 ***E1*** *is satisfied by the composite of the DTOS kernel and the Clark-Wilson security server, assuming the system is initially $Have\_write$ consistent for all processes.*

Proof: This follows from Lemmas 9.2, 9.5 and 9.7. □

Theorem 9.9 *E2 is satisfied by the composite of the DTOS kernel and the Clark-Wilson security server, assuming the system is initially $Have\_write$ consistent for all processes.*

Proof: This follows from Lemmas 9.3, 9.5 and 9.7. □

Definition 9.10 *If in a state s, for every process p*

$$(ssi\_ssc(task\_ssi(process\_task(p)))).ind = process\_individual(p)$$

we will say s is ***individual consistent****.*

Lemma 9.11 *If the DTOS Clark-Wilson system is initially consistent (see Definition 8.7) and is individual consistent in the initial state, then it is individual consistent in every state.*

Proof: Recall that we interpret $process\_individual$ of a process to be the individual assigned to it by a privileged process if this has happened and otherwise the individual associated with the parent process. This theorem is proved by induction. The base case is true by hypothesis. To prove the inductive step we must show that

1. The only way for a security context to be associated with a process is through creation of a process (i.e., task creation),

2. Whenever a new context is associated with a process the new context does not introduce a violation of individual consistency.

We assume that for any existing process the task associated with that process by $process\_task$ does not change. Furthermore we will assume that the context that ssi_ssc associates with any SSI assigned to an existing task does not change. Since the security context associated with a process p is defined by $ssi\_ssc(task\_ssi(process\_task(p)))$, the only ways to associate a context with a process are by creating a new process, with its associated task, (i.e., extending $process\_task$) or by modifying $task\_ssi$. The former occurs only during task creation and the latter occurs only during task creation and the **task_change_sid** operation. The **task_change_sid** operation never occurs since $Make\_sid$ permission is always prohibited and always required by mgr_policy. This demonstrates part (1).

For part (2), if a context is assigned to a process in a request by a privileged process, then by definition, individual consistency holds for that process in the resulting state. So we need only consider the case of a task creation request r by an unprivileged process n to create a task t. In this case $process\_individual'((process\_task')^{\sim}(t)) = process\_individual(n)$. A task creation operation will require either $Create\_task$ or $Cross\_context\_create$ permission from the SID of n to the SID to be assigned to the new task. $Create\_task$ permission is always prohibited (**PR9.4**) so it cannot be used by the kernel. Thus, Corollary 5.7 implies that either the $Cross\_context\_create$ permission is retained in the initial state or there is some preceding state in which the kernel has a security server response granting the $Cross\_context\_create$ permission. If it is retained in the initial state, then initial consistency implies that it was in $policy\_allows$ in the initial state. If it was granted in a security server response, then it was in $policy\_allows$ in the state in which it was granted (see Theorem 5.8). Thus, we know there is some preceding state s in which $policy\_allows$ contains $Cross\_context\_create$ permission from the SID of n to the SID to be assigned to t. Since $Make\_sid$ is always prohibited, $ssi\_ssc(task\_ssi(process\_task(n))) \notin Cw\_privileged\_contexts$ in s. So **PR9.3** implies that the

$$\begin{aligned} ssi\_ssc(task\_ssi(process\_task(n))).ind &= osi\_ssc'(new\_sid).ind \\ &= ssi\_ssc'(task\_ssi'(t)).ind. \end{aligned}$$

Since by hypothesis the system is individual consistent in the state in which r is processed the left hand side is $process\_individual(n)$, and this concludes the proof. □

Theorem 9.12 *E3 is satisfied by the composite of the DTOS kernel and the Clark-Wilson security server, assuming the system is initially consistent and is individual consistent in the initial state.*

Proof: Let p be a process such that

$$\begin{aligned} p &\in \operatorname{dom} process\_procedure \\ process\_procedure(p) &\in transformation\_procedures \end{aligned}$$

Recall that $process\_procedure(p)$ is the $PROCEDURE$ r such that $process\_task(p)$ has performed an $Execute\_page$ operation on $proc\_memory(r)$. Following analysis similar to that in the proof of Lemma 9.11, we find that either $Have\_execute$ permission from $task\_ssi(process\_task(p))$ to $memory\_osi(proc\_memory(process\_procedure(p)))$ is retained in the the initial state or it is granted by the security server. Thus, there exists a preceding state s in which this $Have\_execute$ permission is in $policy\_allows$. $CwConsistentLabels$ implies that

$$\begin{aligned} &osc\_proc(osi\_osc(memory\_osi(proc\_memory(process\_procedure(p))))) \\ &\quad = process\_procedure(p). \end{aligned}$$

Thus, **PR9.8** requires

$$ssi\_ssc(task\_ssi(process\_task(p))).ind \in (ss\_data.authenticated\_individuals)$$

Application of Lemma 9.11 completes the proof. □

Theorem 9.13 *E4 is satisfied by the composite of the DTOS kernel and the Clark-Wilson security server, assuming the system is initially consistent and is individual consistent in the initial state.*

Proof: The proof of this property is essentially the same as for E3 with **PR9.7** used instead of **PR9.8**.

Section *10*
ORCON Policy

This section describes the use of an Originator Controlled (ORCON) [14, 16] policy on a DTOS system. An IBAC security policy usually allows a process that is able to read information from an object, as identified by the object's Access Control List (ACL), to make that information available to other processes at its discretion. This discretionary aspect of an IBAC policy can be eliminated by using an ORCON policy [2]. With an ORCON policy, only those processes allowed to read an object are able to read from any objects that may have been derived from that object.

The root policy is that the originator of a piece of information can specify who may see that information and that everyone who does see the information obeys the originator's wishes. Given the current state of the art, it is much more practical to use this policy with people than it is with computers. The originator of information trusts those to whom access has been granted not to divulge the information to others who are not on the list. However, on a computer system it is processes, not people, which are seeing the information and deciding how they can pass it on. Since these processes are frequently not entirely trusted, it is necessary to assume that when they write to a file they will divulge all information they have previously read. So, any time a process writes to a file, this will likely reduce the set of processes that may read that file. Such a system tends to converge on a state where each process can read only the files to which it can write, resulting in a rather segregated file system. Despite the impracticality of this policy for use on computers, it is still an interesting example to consider in this study because it involves retraction of permissions.

ORCON can be defined in terms of two access control lists. Each object has an associated Access Control List (ACL) as described above. In addition, each process has a Propagated Access Control List (PACL) that lists those processes allowed to receive all of the information that the process possesses. Whenever a process reads an object, the intersection of the processes allowed to read that object and the reader's PACL form a new PACL (a process will always be on its own PACL). Whenever a process writes an object, read access must be removed from the object's ACL for any processes not on the writer's PACL.

A process allowed to read a memory object may also communicate information to other processes through messages and various indirect channels (for example, by modifying the name space of the recipient process). To provide complete security, these channels also need to be protected. For messages, each port should have an ACL restricting the processes that can receive (read) from it. When a process sends a message (writes to a port), the port's ACL is intersected with the process's PACL. For clarity, the model given below only considers memory objects, but it could easily be extended to cover message passing.

We will take the view in this section that the manipulation of ACLs and PACLs as described above is really an implementation rather than a specification of the high-level policy. It belongs in the security server. As stated above, the high-level policy is that the originator of information may define who can see the information. As explained in the MLS/TE section, it is rather difficult to formally state the policy in terms of information.[28] However, rather than retreating

[28] This is probably the reason why one often sees an *implementation* of a policy serving as a *definition* of the policy. This allows the security analyst to avoid the difficulties inherent in an information-based definition. However, any proof that a system complies with the policy is really just a proof that a more detailed implementation (e.g., a decider-

to ACLs and PACLs in our policy definition, we will state the policy as follows: No system behavior is allowed that contains a finite sequence of alternating read and write actions by which it is possible for data to be transferred from an object j to a process p such that p is not allowed to read j.

In the next section we define this policy more formally. Section 10.2 defines the correspondence between the concepts of the abstract, high-level definition of ORCON and the entities of the DTOS kernel. Sections 10.3–10.5 describe a Security Server that enforces the ORCON policy, and Section 10.6 composes the ORCON security server with the DTOS kernel and demonstrates that the high-level policy is satisfied.

10.1 Formal ORCON Definition

The primary entity types in our model of the ORCON policy are processes and memory objects. We add individuals to this list of entity types since, as a matter of practicality, ACLs are defined based upon individuals rather than processes. It would be impractical to modify the ACL of every memory object in the system whenever a new process is created. An individual can be thought of as a user ID. We start by defining given types for these entities. The type for all processes is $ORCON\_PROCESS$, the type for all individuals is $ORCON\_INDIV$, and the type for all memory objects is $ORCON\_OBJ$. Processes perform operations on memory objects. $ORCON\_OP$ is the type for these operations, and $Read$ is one of the operations.

$$[\,ORCON\_PROCESS, ORCON\_INDIV, ORCON\_OBJ, ORCON\_OP\,]$$

$$|\ Read : ORCON\_OP$$

In any given system state, $orcon\_objects$ denotes the set of all memory objects in the system. Each memory object has an associated originator-assigned ACL denoted by $orig\_acl(obj)$. Each ACL is a total function from individuals to sets of memory operations. Unlike the ACL in the implementation which is automatically modified when information is written to an object, this ACL is changed only by explicit actions of the originator. For any object obj, $orcon\_readers(obj)$ is the set of all individuals, ind, such that $Read \in orig\_acl(obj)(ind)$.

$$\begin{array}{l}
\textit{OrconObject} \\
\hline
orcon_objects : \mathbb{P}\ ORCON_OBJ \\
orig_acl : ORCON_OBJ \longrightarrow (ORCON_INDIV \longrightarrow \mathbb{P}\ ORCON_OP) \\
orcon_readers : ORCON_OBJ \longrightarrow \mathbb{P}\ ORCON_INDIV \\
\hline
\forall\, obj : ORCON_OBJ;\ ind : ORCON_INDIV \\
\mid orig_acl(obj)(ind) \neq \varnothing \\
\bullet\ obj \in orcon_objects \\
\\
\forall\, obj : ORCON_OBJ;\ ind : ORCON_INDIV \\
\bullet\ ind \in orcon_readers(obj) \Leftrightarrow Read \in orig_acl(obj)(ind)
\end{array}$$

The set of current processes in the system is $orcon\_processes$. Each process $process$ is executing on the behalf of some individual denoted by $process\_individual(process)$.

enforcer implementation) is a correct implementation of the less detailed implementation. In the case of ORCON, an organization probably does not really care that ACLs and PACLs are manipulated properly. The real concern is that the desires of object originators are enforced.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{OrconProcess}} \\
\hline
OrconObject \\
orcon\_processes : \mathbb{P}\, ORCON\_PROCESS \\
process\_individual : ORCON\_PROCESS \rightarrowtail\!\!\!\!\mapsto ORCON\_INDIV \\
\hline
\mathrm{dom}\, process\_individual = orcon\_processes \\
\hline
\end{array}
$$

The expression $orcon\_indivs$ denotes the set of individuals recognized by the system. If an individual ind is not in this set, then, for all memory objects obj, ind is not allowed to perform any memory operations on obj.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{OrconIndiv}} \\
\hline
OrconObject \\
orcon\_indivs : \mathbb{P}\, ORCON\_INDIV \\
\hline
\forall\, ind : ORCON\_INDIV;\ obj : ORCON\_OBJ \\
\mid ind \notin orcon\_indivs \\
\bullet\ orig\_acl(obj)(ind) = \varnothing \\
\hline
\end{array}
$$

The ORCON system state consists of the above information together with the state of all objects and processes in the system which we summarize as $orcon\_system\_data$.

$$[ORCON\_SYSTEM\_DATA]$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{OrconSystemState}} \\
\hline
OrconObject \\
OrconProcess \\
OrconIndiv \\
orcon\_system\_data : ORCON\_SYSTEM\_DATA \\
\hline
\end{array}
$$

We define the term *application* to refer to the execution of an $ORCON\_OP$ in some state. $APPLICATION$ denotes the type of all applications. Each element of $APPLICATION$ encodes a state and an operation. $Read\_ops(p, j)$ denotes the set of all $APPLICATION$ in which process p reads an object j, and $Write\_ops(p, j)$ the $APPLICATION$ in which p writes to an object j. $Appl\_state$ maps an $APPLICATION$ to the $OrconSystemState$ in which the operation occurs. The set $Transfer\_possible$ denotes the set of all triples (s, j, p) where s is a finite sequence of applications through which it is possible for data to be transferred from object j to process p via a sequence of alternating read and write operations. The set $Traces$ denotes the set of allowed infinite application sequences for an ORCON system. A trace t is in $Traces$ only if it does not contain a finite subsequence[29] s such that for some j and p

- $(s, j, p) \in Transfer\_possible$, and
- the individual associated with p during the last application in s is not allowed to read j by the ACL of j in effect at the start of s.

Some care is required when constraining $Traces$ to avoid an overly strong constraint on the system. For example, assume two objects contain identical data but have different ACLs

[29] The Z function $squash$ takes a finite partial function on $\mathbb{N}_1$ as its argument and compacts it into a finite sequence preserving the order of the elements.

assigned by different originators. Unless it can be determined from any change of state what operation is being executed, a change of state resulting from reading one of the objects will be indistinguishable from a change of state resulting from reading the other. In this case, $Traces$ would require that both ACLs allow p to read the object. This would be rather difficult to implement. When we constrain $Traces$ in terms of $APPLICATION$, we know which object p asked to read, and we only need check the ACL of that object. In the constraint on $Traces$ we use the ACL of j at the start of s so that if the originator changes the ACL of an object the system need not enforce this change retroactively. We also require that the individual associated with an existing process not change in any trace.

$[APPLICATION]$

$$
\begin{array}{l}
Appl\_state : APPLICATION \longrightarrow OrconSystemState \\
Read\_ops, Write\_ops : (ORCON\_PROCESS \times ORCON\_OBJ) \longrightarrow \mathbb{P}\, APPLICATION \\
Transfer\_possible : \mathbb{P}(\mathrm{seq}\, APPLICATION \times ORCON\_OBJ \times ORCON\_PROCESS) \\
Traces : \mathbb{P}(\mathbb{N}_1 \longrightarrow APPLICATION) \\
\hline
(\forall\, t : Traces;\ m, n : \mathbb{N}_1 \mid n \in \mathrm{dom}\, t \land m < n \bullet m \in \mathrm{dom}\, t) \\
\\
(\forall\, s : \mathrm{seq}\, APPLICATION;\ j : ORCON\_OBJ;\ p : ORCON\_PROCESS \\
\mid (s, j, p) \in Transfer\_possible \\
\bullet \#s \bmod 2 = 1 \\
\quad \land (\exists\, k : \mathrm{seq}\, ORCON\_OBJ;\ r : \mathrm{seq}\, ORCON\_PROCESS \\
\qquad \bullet \#k = \#r = (\#s \operatorname{div} 2) + 1 \\
\qquad \land j = k(1) \\
\qquad \land p = r(\#r) \\
\qquad \land s(\#s) \in Read\_ops(r(\#r), k(\#k)) \\
\qquad \land (\forall\, i : 1 \mathrel{..} (\#k - 1) \\
\qquad\quad \bullet s(2 * i - 1) \in Read\_ops(r(i), k(i)) \\
\qquad\qquad \land s(2 * i) \in Write\_ops(r(i), k(i + 1))))) \\
\\
(\forall\, t : Traces;\ d : \mathbb{F}\, \mathbb{N}_1; \\
\quad s : \mathrm{seq}\, APPLICATION;\ j : ORCON\_OBJ;\ p : ORCON\_PROCESS; \\
\quad Initial\_acl : ORCON\_INDIV \longrightarrow \mathbb{P}\, ORCON\_OP \\
\mid s = squash\,(d \lhd t) \\
\quad \land (s, j, p) \in Transfer\_possible \\
\quad \land Initial\_acl = ((Appl\_state(s(1))).orig\_acl)(j) \\
\bullet Read \in Initial\_acl(((Appl\_state(s(\#s))).process\_individual)(p))) \\
\\
\forall\, t : Traces;\ i, j : \mathbb{N}_1;\ p : ORCON\_PROCESS \\
\mid \{i, j\} \subseteq \mathrm{dom}\, t \\
\quad \land p \in (Appl\_state(t(i))).orcon\_processes \\
\quad \land p \in (Appl\_state(t(j))).orcon\_processes \\
\bullet ((Appl\_state(t(i))).process\_individual)(p) = ((Appl\_state(t(j))).process\_individual)(p)
\end{array}
$$

10.2 ORCON Objects and the Kernel

The ORCON policy is defined in terms of processes, individuals and objects (e.g., files). Since these concepts do not exist at the microkernel level, we map them to microkernel entities. ORCON objects will be represented by memory objects. The associated memories can be obtained via the mapping obj_memory. This mapping is an injection, meaning that no two

ORCON objects may have the same associated memory. Processes are represented by tasks. The task associated with a process is denoted by $process\_task(process)$. This mapping is an injection, meaning that no two processes may have the same associated task. Individuals do not correspond to any microkernel entity, but rather to user identifiers. We consider them later when defining subject security contexts.

$$
\begin{array}{l}
\textit{OrconToDtos} \\
\hline
OrconSystemState \\
MkAddressSpace \\
MkLabels \\
obj\_memory : ORCON\_OBJ \rightarrowtail\!\!\!\!\!\!+\; MEMORY \\
process\_task : ORCON\_PROCESS \rightarrowtail\!\!\!\!\!\!+\; TASK \\
\hline
\mathrm{dom}\, obj\_memory = orcon\_objects \\
\mathrm{dom}\, process\_task = orcon\_processes \\
\mathrm{ran}\, obj\_memory \subseteq \mathrm{dom}\, memory\_osi \\
\mathrm{ran}\, process\_task \subseteq \mathrm{dom}\, task\_osi \\
\forall\, region : REGION \\
\mid mapped\_memory(region) \in \mathrm{ran}\, obj\_memory \\
\bullet\; region\_osi(region) \\
\qquad = memory\_osi\_region\_osi(memory\_osi(mapped\_memory(region)))
\end{array}
$$

Before proceeding to define an ORCON Security Server we consider several issues in supporting a retractive policy such as ORCON in the DTOS architecture. The retraction that happens in ORCON is characterized by the following sequence of events.

1. Task A reads an object that task B is not permitted to read.
2. Task B obtains read permission to object j.
3. Task A obtains write permission to j.
4. Task A writes to j.

To prevent B from obtaining data it is not permitted to read, we must ensure that task B loses its ability to read j no later than the start of Step 4.

The first issue is that in DTOS even if the Security Server sends a flush request to the manager to remove read permission from B to j the Security Server has no way of determining when this has actually occurred. As noted in Section 6.3, this could be remedied by having the flush thread send a notification to the Security Server when it has finished the flush.

The second issue deals with concurrency. Let us elaborate the above event sequence as follows:

1. Task A reads an object that task B is not permitted to read.
2. Task B (via the manager) requests read permission to object j.
3. The Security Server sends a response r to the manager granting read permission.
4. Task A requests write permission to j.
5. The Security Server sends a request to the manager to flush read permission from B to j.
6. The flush request is processed (with no effect since response r has not yet been processed), and a notification is sent back to the Security Server.

7. The Security Server responds to A's request granting write permission to j, and the manager processes this response.
8. Task A writes to j.
9. The manager processes response r giving B read permission to j.
10. Task B reads j.

This scenario can be dealt with by attaching information to Security Server responses and flush requests that allows the manager to detect a response that is overruled by a subsequent flush. The manager can then reissue the Security Server request or refuse to perform the operation. DTOS does support such information in Security Server responses and flush requests. However, it currently uses this information only to control caching. A ruling based upon a previous policy is still used for the permission check that caused the Security Server interaction.

Since there may be an arbitrarily long delay between the checking of a permission and the actual performance of the request, there is another variation on this scenario:

1. Task A reads an object that task B is not permitted to read.
2. Task B (via the manager) requests read permission to object j.
3. The Security Server sends a response r to the manager granting read permission.
4. The manager processes response r giving B read permission to j and recording this in $obtained$.
5. Task A requests write permission to j.
6. The Security Server sends a request to the manager to flush read permission from B to j.
7. The flush request is processed retracting read permission from B to j (but recall that this permission has already been checked for the above request) and a notification is sent back to the Security Server.
8. The Security Server responds to A's request granting write permission to j and the manager processes this response.
9. Task A writes to j.
10. Task B's read request is marked as allowed and the read operation is performed.

This case is essentially an instance of check-before-use and requires slightly more drastic measures. We either need to have an atomic operation in which we determine that all the required permissions are still held or we need to ensure that the data returned to B is equivalent to what it would have been if A had not written to j. The atomic operation is probably not feasible in the DTOS architecture, so we consider the second option and assume that the read operation occurs as follows:

1. Before any permission checking is performed for a read request (including checks on retained permissions as well as the sending of permission requests to the security server) the object to be read is locked so that it may not be written.[30]

[30] An alternative to this object locking schema is to copy the data to be read before the permission checking begins and to base the results on the copy of the data rather than the current contents.

2. The kernel performs all processing associated with the read request, resulting in either successful or unsuccessful completion of the request.

3. The lock on the object is released so that write operations may again occur.

In terms of the manager framework, an object to be read becomes locked no later than the first *MgrRequestComputation* transition, if one occurs, and no later than the *MgrAcceptRequest* transition, if one occurs.[31] The locking must extend through the transition in which the request is removed from *active_request*.

DTOS does not perform this type of read operation. Since DTOS does not adequately support ORCON, we will continue our analysis using a hypothetical manager that deals with the issues identified above. We first assume a flush operation which we call **complete_flush** that removes all matching retentions in a single transition. For simplicity, we will assume that **complete_flush** takes three parameters: the host control port, a sequence of (ssi, osi) pairs, and a time stamp. The second parameter is interpreted by the function *find_sid_pairs*, and the third by *Find_number*. All retained permissions for the indicated SID-pairs will be removed, and the time stamp will be recorded in the cache. We do not require retained denials to also be flushed. The schema *OrconMgrCompleteFlush* defines the behavior of the **complete_flush** request.

$$
\begin{array}{|l}
complete\_flush\_id : KERNEL\_OP \\
find\_sid\_pairs : KERNEL\_PARAM \rightarrowtail\!\!\!\!\!+ (SSI \leftrightarrow OSI)
\end{array}
$$

OrconMgrCompleteFlush

$$
\begin{array}{|l}
MkProcessRequest \\
\hline
\textbf{let } req == active\_request(req\_num?); \\
\quad pars == (active\_request(req\_num?)).params \\
\bullet\ mgr\_data'.time\_stamp = Find\_number(pars(3)) \\
\quad \land\ req.op = complete\_flush\_id \\
\quad \land\ find\_sid\_pairs(pars(2)) \in (SSI \leftrightarrow OSI) \\
\quad \land\ retained\_rel' \subseteq retained\_rel \\
\qquad \rhd \{ ss\_req : D\_SS\_REQ \\
\qquad\quad | (\exists pair : find\_sid\_pairs(pars(2)) \\
\qquad\qquad \bullet (Perm\_req^{\sim}(ss\_req)).ssi = first(pair) \\
\qquad\qquad \lor (Perm\_req^{\sim}(ss\_req)).osi = second(pair)) \}
\end{array}
$$

We assume a "flush complete" notification is sent back to the Security Server to indicate that the flush has completed. Later, we will specify that the Security Server waits until the notification is received before processing any additional security computations.

$$
\begin{array}{|l}
Flush\_complete : D\_NOTIF\_REQ
\end{array}
$$

FlushCompleteNotification

$$
\begin{array}{|l}
MkSendNotification \\
\hline
ss\_req? = Notif\_req(Flush\_complete)
\end{array}
$$

[31] If no *MgrAcceptRequest* transition occurs, then the request is never processed.

We will further assume that the manager processes an affirmative response (schema $MgrAffirmativeResponse$) only if the time stamp is at least as large as the time stamp stored with the cache.

$OrconMgrAffirmativeResponse$
$MkAffirmativeResponse$

$$mgr\_data.time\_stamp \leq (Ruling\_resp^{\sim}(ss\_response?)).time\_stamp$$

If a response with an out-of-date time stamp is received, the manager may remove the $ss\_req?$ involved from $sent$, allowing the Security Server request to be retried (note that this removal is allowed by the generic framework), but it may not consider the $ss\_req?$ as obtained.

$OrconMgrNegativeResponse$
$MkAffirmativeResponse$

$$mgr\_data.time\_stamp > (Ruling\_resp^{\sim}(ss\_response?)).time\_stamp$$
$$obtained\_rel' \subseteq obtained\_rel$$

10.3 ORCON Security Server

In this section we describe a Security Server that, when combined with the modified kernel described in the preceding section, satisfies the ORCON security policy. As a simplification, we will assume that $orig\_acl$ is invariant in the system. That is, the ACL assigned to each object by the originator of that object is known at system start-up and does not change. Without this assumption we would have to provide a mechanism for originators to supply the desired ACL to the security server when creating an object and a mechanism for the originator of an object to notify the security server of a desired change to the ACL of the object. The former mechanism could be implemented by providing a security server information request that takes a desired ACL as a parameter and returns a SID that may then be used to create an object with the desired ACL. When processing this request, the security server would allocate a security context for the anticipated new object, associate the supplied ACL with that context, allocate a SID to serve as an identifier for the new context, and return the SID to the client of the request. If an originator further restricts the ACL of an object at a later time, this restriction would only affect future operations on the object. It will be necessary to constrain an originator's ability to relax the ACL of an object. Since other individuals may have already written data to the object, the object's ACL must henceforth be constrained by the ACL of that data. Therefore, the originator cannot grant read access to anyone not allowed to read all the data written to the file.

For the ORCON Security Server, the processing of a permission request will require several transitions. To control this processing, we introduce $ORCON\_STATUS$ and $OrconProcessingState$. The values in $ORCON\_STATUS$ have the following meanings:

- $Locked$ — The Security Server has determined what response to send for a selected permission request and has determined that certain permissions must be flushed, but it has not yet sent the response nor issued the flush request.
- $Waiting$ — The Security Server has issued the flush request and is waiting for notification that the request has been completed.
- $Flush\_done$ — The Security Server has received notification that the flush request is done.

- *Unlocked* — The Security Server is not currently processing any permission request. Any permission requests that the Security Server has previously begun processing have been completed and the responses sent.

The *lock* and *status* fields record the security server computation (i.e., permission request) that is currently being processed and the point the security server has reached in the processing of that request. There may be at most one computation in *lock* at a time, and the server has status *Unlocked* exactly when *lock* is empty. The *formulated_response* and *required_flushes* fields store information that is determined at the beginning of the processing of a computation for use in later transitions. The *time_stamp* is incremented in each security server transition and is sent in each ruling and in each request to flush permissions from the kernel. This allows outdated rulings to be recognized. The use of this state information is described more fully in Section 10.4.

$$ORCON\_STATUS ::= Locked \mid Waiting \mid Flush\_done \mid Unlocked$$

$$\begin{array}{|l}
\hline
\textit{OrconProcessingState} \\
lock : \mathbb{P}\ COMP_NUMBER \\
status : ORCON_STATUS \\
formulated_response : D_RESP \\
required_flushes : SSI \leftrightarrow OSI \\
time_stamp : \mathbb{N} \\
\hline
\#lock \leq 1 \\
status = Unlocked \Leftrightarrow lock = \varnothing \\
\hline
\end{array}$$

10.3.1 Security Database

10.3.1.1 Security Contexts Now we define the security contexts. The set of object security contexts is $ORCON\_OSC$. We define this type to be identical to $ORCON\_OBJ$.

$$ORCON\_OSC == ORCON\_OBJ$$

Subject security contexts are associated with processes. A subject security context contains the following two components: the *user* and the *index*. The *user* provides a record of the individual on whose behalf the subject is executing. In the following we will constrain the creation of subjects so that when an unprivileged individual or process creates a new process, the *user* of the newly created process is identical to that of the creating individual or process. The *index* is included to allow a single user to run several processes at the same time, each with a distinct subject context and therefore a possibly distinct PACL.

$$\begin{array}{|l}
\hline
\textit{OrconSsc} \\
user : ORCON_INDIV \\
index : \mathbb{N} \\
\hline
\end{array}$$

The set *Orcon_privileged_contexts* denotes a set of privileged subject contexts. These contexts may have permission to create a process in a context with a different user.

$$\mid\ Orcon\_privileged\_contexts : \mathbb{P}\ OrconSsc$$

We define $orcon\_sscs$ and $orcon\_oscs$ to be the existing subject and object security contexts.

$$\begin{array}{|l}
\multicolumn{1}{l}{OrconContexts} \\
\hline
orcon_sscs : \mathbb{P}\ OrconSsc \\
orcon_oscs : \mathbb{P}\ ORCON_OSC \\
\hline
\end{array}$$

Each subject security identifier is mapped by the function ssi_ssc to a subject security context, and each object security identifier is mapped by osi_osc to an object context. The function osi_ssc maps the object security identifier of a task port to the subject context of that task.

$$\begin{array}{|l}
\multicolumn{1}{l}{OrconSidToContext} \\
\hline
OrconContexts \\
MkLabels \\
ssi_ssc : SSI \rightarrowtail\!\!\!\!\!\!+\; OrconSsc \\
osi_osc : OSI \rightarrowtail\!\!\!\!\!\!+\; ORCON_OSC \\
osi_ssc : OSI \mathrel{+\!\!\!\!\!\rightarrow} OrconSsc \\
\hline
\operatorname{ran} ssi_ssc \subseteq orcon_sscs \\
\operatorname{ran} osi_osc \subseteq orcon_oscs \\
\operatorname{dom} osi_osc = OSI \setminus \operatorname{dom} osi_ssi \\
osi_ssc = osi_ssi \mathbin{;} ssi_ssc \\
\hline
\end{array}$$

We make the following assumptions regarding the labeling of memory and tasks:

- Every OSI used to label memory has an associated OSC.
- Every SSI used to label a task has an associated SSC.
- For every $j \in \operatorname{dom} obj\_memory$, the OSI used to label the memory associated with j maps to an OSC associated with j by the security server.

If we had a more complete model of the system including the file server, the ways in which individuals may log on to the system and the ways in which the DTOS kernel assigned SIDs, these assumptions would be justified in terms of that model. To allow us to focus on the central ORCON requirement, rather than proving consistency for $process\_task$ as we have done in the preceding sections, we will state a consistency property: the context associated with the task of p has a $user$ field equal to $process\_individual(p)$.

$$\begin{array}{|l}
\multicolumn{1}{l}{OrconConsistentLabels} \\
\hline
OrconSidToContext \\
OrconToDtos \\
\hline
\operatorname{ran} memory_osi \subseteq \operatorname{dom} osi_osc \\
\operatorname{ran} task_ssi \subseteq \operatorname{dom} ssi_ssc \\
\forall j : \operatorname{dom} obj_memory \\
\bullet\ osi_osc(memory_osi(obj_memory(j))) = j \\
\forall p : \operatorname{dom} process_task \\
\bullet\ (ssi_ssc(task_ssi(process_task(p)))).user = process_individual(p) \\
\hline
\end{array}$$

10.3.1.2 Policy Representation We now define the structures used to store the Security Server representation of an ORCON policy. The function ss_pacl denotes the Security Server representation of the PACL described in the introduction to this section and ss_acl denotes the representation of the ACL. For any context c, $readers(c)$ is the set of all individuals, p, such that $Have\_read \in ss\_acl(c)(p)$.

$$
\begin{array}{|l}
\hline
\textit{OrconPolicyData} \\
ss\_pacl : OrconSsc \rightarrow\!\!\!\!+ \mathbb{P}\ ORCON\_INDIV \\
ss\_acl : ORCON\_OSC \rightarrow\!\!\!\!+ (ORCON\_INDIV \rightarrow\!\!\!\!+ \mathbb{P}\ PERMISSION) \\
readers : ORCON\_OSC \longrightarrow \mathbb{P}\ ORCON\_INDIV \\
\hline
\forall\, c : ORCON\_OSC;\ p : ORCON\_INDIV \\
\bullet\ p \in readers(c) \\
\quad \Leftrightarrow c \in \mathrm{dom}\, ss\_acl \\
\qquad \wedge\ p \in \mathrm{dom}(ss\_acl(c)) \\
\qquad \wedge\ Have\_read \in ss\_acl(c)(p) \\
\hline
\end{array}
$$

The data maintained by the ORCON Security Server includes the definition of the current policy in terms of contexts, the processing state and other miscellaneous data, $other\_orcon\_data$, from which the generic Security Server information is extracted.

$$[\,OTHER\_ORCON\_DATA\,]$$

$$
\begin{array}{|l}
\hline
\textit{OrconData} \\
OrconSidToContext \\
OrconPolicyData \\
OrconProcessingState \\
OrconObject \\
other\_orcon\_data : OTHER\_ORCON\_DATA \\
\hline
\end{array}
$$

10.3.2 Permission Requirements

In this section we state constraints on the permissions that are in $policy\_allows$. These constraints will be used below in defining the ORCON Security Server state.

Policy Requirement 10.1 : Each permission is granted to ssi for osi only if the permission is an element of the set of permissions allowed the user from the subject context of ssi in the ss_acl associated with the object context of osi.

$$
\begin{array}{|l}
\hline
\textit{InAcl} \\
SsPolicyAllows[OrconData, D\_SS\_REQ, D\_ANS] \\
\hline
\forall\, ss\_req : policy\_allows;\ preq : PermReq \\
\mid Perm\_req(preq) = ss\_req \\
\bullet\ \mathbf{let}\ the\_acl == ss\_data.ss\_acl; \\
\qquad the\_osc == (ss\_data.osi\_osc)(preq.osi); \\
\qquad user == ((ss\_data.ssi\_ssc)(preq.ssi)).user \\
\quad \bullet\ preq.perm \in the\_acl(the\_osc)(user) \\
\hline
\end{array}
$$

Note that because of the definition of D_RESP for the DTOS kernel, all the permissions granted in a single Security Server response must have the same SSI and OSI. As described in Section 10.4, the granting of read and write permission can cause the policy to change. To prevent unnecessary policy changes from occurring, an implementation of an ORCON Security Server should probably be stingy with respect to these permissions. That is, the permissions $Have\_read$ and $Have\_write$ should only be granted when specifically requested. It is important that any denied permission with respect to which the Security Server is stingy not be in the corresponding cache control vector. This signals the kernel to send another request when the permission is needed. Since stinginess deals with keeping the system flexible rather than ensuring that the policy is enforced, we omit stinginess from our specification of the ORCON Server.

Policy Requirement 10.2 : $Create\_task$ or $Cross\_context\_create$ permission to an OSI (of a task port) is given to an SSI only if both SIDs map to subject contexts with the same $user$ component or the SSI identifies a privileged context.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{OrconCorrectUser}} \\
\hline
SsPolicyAllows[OrconData, D\_SS\_REQ, D\_ANS] \\
\hline
\forall\, ss\_req : policy\_allows;\ preq : PermReq \\
\mid Perm\_req(preq) = ss\_req \\
\qquad \wedge\ preq.perm \in \{Cross\_context\_create, Create\_task\} \\
\bullet\ (ss\_data.ssi\_ssc)(preq.ssi) \in Orcon\_privileged\_contexts \\
\qquad \vee\ ((ss\_data.ssi\_ssc)(preq.ssi)).user = ((ss\_data.osi\_ssc)(preq.osi)).user \\
\hline
\end{array}
$$

Policy Requirement 10.3 : $Make\_sid$ permission is prohibited.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{OrconMakeSid}} \\
\hline
SsPolicyAllows[OrconData, D\_SS\_REQ, D\_ANS] \\
\hline
\forall\, ss\_req : policy\_allows;\ preq : PermReq \\
\mid Perm\_req(preq) = ss\_req \\
\bullet\ preq.perm \neq Make\_sid \\
\hline
\end{array}
$$

10.3.3 Security Server State

The ORCON Security Server combines the general properties of a DTOS Security Server (i.e., the types $MkRequest$, D_SS_REQ, D_RESP and D_ANS) with the ORCON Security Server data and the above policy requirements. Only an active Security Server computation may have the Security Server locked.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{OrconState}} \\
\hline
SsState[MkRequest, OrconData, D\_SS\_REQ, D\_RESP, D\_ANS] \\
InAcl \\
OrconCorrectUser \\
OrconMakeSid \\
\hline
ss\_data.lock \subseteq \mathrm{dom}\ active\_computations \\
\hline
\end{array}
$$

10.4 Operations

When the Security Server sends a ruling it might also need to change its policy data. In particular, if $Have\_write$ or $Have\_read$ permission is granted in the response, ss_acl or ss_pacl might change. Given the above constraints on $policy\_allows$, the policy might change to remain consistent with the policy data. Since one of the permissions $Have\_write$ or $Have\_read$ might change from being grantable to being ungrantable, and since these permissions might have been granted in a previous ruling, it is necessary for the ORCON Security Server to flush retained permissions from the kernel.

When a Security Server request is received there may be up to five Security Server transitions needed to complete the processing of that request:

- send a response
- modify the internal state
- send a flush request to the kernel
- receive a notification that the flush is complete
- process the "flush complete" notification

These transitions cannot be collapsed into a single Security Server transition because of the way in which the generic framework models the Security Server operations. The framework defines the first three transition types to be non-overlapping. For example, an *SsInternalTransition* requires *pending_requests* and *pending_responses* to be invariant whereas *SsMgrRequest* requires that *pending_requests* change, and *SsSendResponseAux* requires that *pending_responses* change. Even though *SsMgrRequest* allows the internal state to change, we must still separate it from the *SsInternalTransition* since the internal state may change without any flushing being necessary. Although we break the processing into smaller transitions, there must be a certain amount of consistency between them. In particular, the changes made to the policy and the flush request that is sent must both be based upon the response that is made.

The server is further complicated by the fact that the following sequencing constraints must be obeyed by the ORCON Security Server.

- If the flush for an active computation A occurs before the policy update, then no other active computation B can be allowed to generate a Security Server response between the flush and the policy update for A. This prevents flushed permissions from being reobtained before they are removed from the policy.
- Any necessary flushing must occur no later than the sending of the response granting the new permissions. This prevents a state where the kernel is simultaneously holding some permissions from the new policy and others from an old policy where the old ones are not allowed in the new policy and are to be flushed.
- If a response for an active computation A is determined in a transition t_1 that precedes the transition t_2 that changes the policy to reflect that response, then we must ensure that there is no other active computation B for which the response is determined in a transition that occurs between t_1 and t_2. The determination of each response must reflect all prior response determinations.

To satisfy all of these requirements, we will allow an active Security Server computation to lock the Security Server while it is being processed. That is, once the Security Server begins processing a particular request R that it has previously received, it will not do any processing of any other request (although it might receive new requests) until the processing for R is completed. This locking will be accomplished in the specification using the *lock* component of the internal data.[32] The component *status* will keep track of where the Security Server is in its processing of R, and *formulated_response* will be used in determining updates to the policy data, any necessary flushing and the response that is sent. The Security Server will perform the processing of R as follows:

[32] There is more than one way in which this could be achieved in an actual implementation. For example, the Security Server could be single-threaded, or it could use a lock variable as done in this specification.

1. Initialize the internal processing state to indicate that R is being processed, $formulated\_response$ is to be sent and the retained permissions for $required\_flushes$ must be flushed. Also update the policy data.[33] The $status$ moves from $Unlocked$ to either $Locked$, if a flush is necessary, or $Flush\_done$ otherwise.

2. If $required\_flushes$ is non-empty, then

   (a) send a flush request ($status$ moves from $Locked$ to $Waiting$),

   (b) receive the "flush complete" notification (note that other requests may also be received, but not processed, during this time), and

   (c) process the "flush complete" notification ($status$ moves from $Waiting$ to $Flush\_done$).

   (d) Send the response ($status$ moves from $Flush\_done$ to $Unlocked$).

Although we do not include this in our specification, the security server should be implemented to "time-out" on waiting for a flush complete notification. This would prevent the security server from becoming permanently blocked waiting for a notification that never comes. Without the time-out implementation, there is a potential denial of server attack. Note that under this implementation the security server must back out any changes it has made to ss_acl and ss_pacl while processing the current request before it begins processing another request.

We first formalize that a computation releases the lock only when the Security Server has finished processing the computation. After the transition in which a lock is obtained, ss_acl, ss_pacl, $formulated\_response$ and $required\_flushes$ remain constant until the lock is released. We also require that the time stamp be incremented in each Security Server transition.

$$
\begin{array}{l}
\textbf{OrconStep} \\
\hline
SsStep[MkRequest, OrconData, D\_SS\_REQ, D\_RESP, D\_ANS] \\
comp\_num? : COMP\_NUMBER \\
\hline
comp\_num? \in ss\_data.lock \setminus ss\_data'.lock \\
\quad \Rightarrow comp\_num? \in \mathrm{dom}\ active\_computations \setminus \mathrm{dom}\ active\_computations' \\
ss\_data.status \neq Unlocked \\
\quad \Rightarrow (ss\_data'.ss\_acl = ss\_data.ss\_acl \\
\qquad \wedge ss\_data'.ss\_pacl = ss\_data.ss\_pacl \\
\qquad \wedge ss\_data'.formulated\_response = ss\_data.formulated\_response \\
\qquad \wedge ss\_data'.required\_flushes = ss\_data.required\_flushes) \\
ss\_data'.time\_stamp = ss\_data.time\_stamp + 1 \\
\hline
\end{array}
$$

Since the first transition in the processing of a request is rather complicated we break its specification into pieces. To begin, we define schemas $NewPaclAux$ and $NewAclAux$. We then use these in defining $OrconFindAclsAux$. Schema $NewPaclAux$ models the computation of a new PACL as a result of the granting of $Have\_read$ permission to $client\_ssc?$ for $memory\_osc?$. Schema $NewAclAux$ models the computation of a new ACL as a result of the granting of $Have\_write$ permission to $client\_ssc?$ for $memory\_osc?$.

[33] In ORCON it is never the case that the granting of a permission from ssi to osi makes that permission ungrantable from ssi to osi in the future. If this did happen, then it would be necessary to delay the update of the policy data until the transition in which the response is sent. Otherwise, we would violate the generic framework restriction that the response only grant permissions that are allowed by the policy in effect when the response is sent.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_ \mathit{NewPaclAux} \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
old\_acl : ORCON\_OSC \pfun (ORCON\_INDIV \pfun \mathbb{P}\ PERMISSION) \\
old\_pacl, new\_pacl : OrconSsc \pfun \mathbb{P}\ ORCON\_INDIV \\
client\_ssc? : OrconSsc \\
memory\_osc? : ORCON\_OSC \\
\hline
new\_pacl(client\_ssc?) = old\_pacl(client\_ssc?) \\
\qquad \cap \{ user : ORCON\_INDIV \mid Have\_read \in old\_acl(memory\_osc?)(user)\} \\
\{ client\_ssc? \} \ndres new\_pacl = \{ client\_ssc? \} \ndres old\_pacl \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_ \mathit{NewAclAux} \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
old\_acl, new\_acl : ORCON\_OSC \pfun (ORCON\_INDIV \pfun \mathbb{P}\ PERMISSION) \\
old\_pacl : OrconSsc \pfun \mathbb{P}\ ORCON\_INDIV \\
client\_ssc? : OrconSsc \\
memory\_osc? : ORCON\_OSC \\
\hline
new\_acl(memory\_osc?) = old\_acl(memory\_osc?) \\
\qquad \oplus \{user : ORCON\_INDIV \mid user \notin old\_pacl(client\_ssc?) \\
\qquad\qquad \bullet user \mapsto old\_acl(memory\_osc?)(user) \setminus \{ Have\_read \}\} \\
\{ memory\_osc? \} \ndres new\_acl = \{ memory\_osc? \} \ndres old\_acl \\
\hline
\end{array}
$$

Schema $OrconFindAclsAux$ models the changes to ss_pacl and ss_acl that are necessary when $response$ is to be sent.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_ \mathit{OrconFindAclsAux} \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
SsInternalTransition[MkRequest, OrconData, D\_SS\_REQ, D\_RESP, D\_ANS] \\
OrconStep \\
response : D\_RESP \\
\hline
\mathbf{let}\ granted\_perms == \{ preq : Perm\_req^{\sim} \limg grants(response) \rimg \bullet preq.perm \}; \\
\qquad old\_acl == ss\_data.ss\_acl; \\
\qquad old\_pacl == ss\_data.ss\_pacl; \\
\qquad new\_acl == ss\_data'.ss\_acl; \\
\qquad new\_pacl == ss\_data'.ss\_pacl \\
\bullet (Have\_read \notin granted\_perms \\
\qquad \Rightarrow new\_pacl = old\_pacl) \\
\land (Have\_read \in granted\_perms \\
\qquad \Rightarrow (\forall preq : Perm\_req^{\sim} \limg grants(response) \rimg \\
\qquad\qquad \bullet \mathbf{let}\ client\_ssc? == (ss\_data.ssi\_ssc)(preq.ssi); \\
\qquad\qquad\qquad\qquad memory\_osc? == (ss\_data.osi\_osc)(preq.osi) \\
\qquad\qquad\qquad \bullet NewPaclAux)) \\
\land (Have\_write \notin granted\_perms \\
\qquad \Rightarrow new\_acl = old\_acl) \\
\land (Have\_write \in granted\_perms \\
\qquad \Rightarrow (\forall preq : Perm\_req^{\sim} \limg grants(response) \rimg \\
\qquad\qquad \bullet \mathbf{let}\ client\_ssc? == (ss\_data.ssi\_ssc)(preq.ssi); \\
\qquad\qquad\qquad\qquad memory\_osc? == (ss\_data.osi\_osc)(preq.osi) \\
\qquad\qquad\qquad \bullet NewAclAux)) \\
\hline
\end{array}
$$

Next we specify $OrconFindFlushesAux$ which models the calculation of the flush request that must be made. Let ssc and osc be the contexts associated with ssi and osi, respectively. Retained permissions for the pair (ssi, osi) must be flushed when either of the following conditions are satisfied:

- ss_pacl is changing, ssc is the context of the process being granted $Have\_read$ permission, and the current ACL of osc grants $Have\_write$ permission to $ssc.user$.

- ss_acl is changing, osc is the context of the memory object to which $Have\_write$ permission is being granted and the current ACL of osc grants $Have\_read$ permission to $ssc.user$.

$$
\begin{array}{l}
\textbf{OrconFindFlushesAux} \\
\hline
SsInternalTransition[MkRequest, OrconData, D\_SS\_REQ, D\_RESP, D\_ANS] \\
OrconStep \\
\hline
\exists\, pacl\_flushes, acl\_flushes : SSI \leftrightarrow OSI; \\
\qquad\quad preq : PermReq; \\
\qquad\quad client\_ssi : SSI;\ memory\_osi : OSI; \\
\qquad\quad user : ORCON\_INDIV;\ memory\_osc : ORCON\_OSC \\
\mid Perm\_req(preq) = active\_computations(comp\_num?) \\
\quad \land\ client\_ssi = preq.ssi \\
\quad \land\ memory\_osi = preq.osi \\
\quad \land\ user = ((ss\_data.ssi\_ssc)(client\_ssi)).user \\
\quad \land\ memory\_osc = (ss\_data.osi\_osc)(memory\_osi) \\
\quad \land\ pacl\_flushes = \textbf{if}\ ss\_data.ss\_pacl \neq ss\_data'.ss\_pacl\ \textbf{then} \\
\qquad\quad \{\, osi : OSI;\ osc : ORCON\_OSC \\
\qquad\qquad \mid osc = (ss\_data.osi\_osc)(osi) \\
\qquad\qquad\quad \land\ Have\_write \in ss\_data.ss\_acl(osc)(user) \\
\qquad\qquad \bullet\ (client\_ssi, osi)\,\} \\
\qquad\quad \textbf{else}\ \emptyset \\
\quad \land\ acl\_flushes = \textbf{if}\ ss\_data.ss\_acl \neq ss\_data'.ss\_acl\ \textbf{then} \\
\qquad\quad \{\, ssi : SSI;\ user : ORCON\_INDIV \\
\qquad\qquad \mid user = ((ss\_data.ssi\_ssc)(ssi)).user \\
\qquad\qquad\quad \land\ Have\_read \in ss\_data.ss\_acl(memory\_osc)(user) \\
\qquad\qquad \bullet\ (ssi, memory\_osi)\,\} \\
\qquad\quad \textbf{else}\ \emptyset \\
\bullet\ ss\_data'.required\_flushes = pacl\_flushes \cup acl\_flushes \\
\hline
\end{array}
$$

The transition $OrconBeginProcessing$ performs the first of the five steps involved in processing a request. If flushing is required the $status$ is set to $Locked$. Otherwise, it is set to $Flush\_done$ to indicate that no flush request is needed.

$$
\begin{array}{|l}
\hline
\textit{OrconBeginProcessing} \\
SsInternalTransition[MkRequest, OrconData, D\_SS\_REQ, D\_RESP, D\_ANS] \\
OrconStep \\
OrconFindFlushesAux \\
\hline
ss\_data.lock = \emptyset \\
comp\_num? \in \operatorname{dom} active\_computations \\
\\
ss\_data'.lock = \{comp\_num?\} \\
grants(ss\_data'.formulated\_response) \subseteq policy\_allows \\
ss\_data'.status = \mathbf{if}\ ss\_data'.required\_flushes = \emptyset \\
\quad \mathbf{then}\ Flush\_done \\
\quad \mathbf{else}\ Locked \\
\mathbf{let}\ response == ss\_data'.formulated\_response \\
\bullet\ OrconFindAclsAux \\
\hline
\end{array}
$$

The transition *OrconFlush* sends a **complete_flush** request to the kernel. The server status changes from *Locked* to *Waiting*. The $comp\_num?$ must be the locking computation.

$$
\begin{array}{|l}
\hline
\textit{OrconFlush} \\
SsMgrRequest[MkRequest, OrconData, D\_SS\_REQ, D\_RESP, D\_ANS] \\
OrconStep \\
\hline
comp\_num? \in ss\_data.lock \\
ss\_data.status = Locked \\
ss\_data.required\_flushes \neq \emptyset \\
req?.op = complete\_flush\_id \\
\mathbf{let}\ pars == req?.params \\
\bullet\ find\_sid\_pairs(pars(2)) = ss\_data.required\_flushes \\
\land\ Find\_number(pars(3)) = ss\_data.time\_stamp \\
\\
ss\_data'.status = Waiting \\
\hline
\end{array}
$$

The receipt of the "flush complete" notification is handled by *OrconReceiveRequest*, the same transition that receives all requests. This transition is modeled below. The transition *ProcessFlushNotification* models the completion of the RPC invocation of **complete_flush**. It processes the "flush complete" notification changing the status from *Waiting* to *Flush_done*.

$$
\begin{array}{|l}
\hline
\textit{ProcessFlushNotification} \\
comp\_num? : COMP\_NUMBER \\
ss\_req? : D\_SS\_REQ \\
SsInternalTransition[MkRequest, OrconData, D\_SS\_REQ, D\_RESP, D\_ANS] \\
OrconStep \\
\hline
ss\_data.status = Waiting \\
(comp\_num?, ss\_req?) \in active\_computations \\
ss\_req? = Notif\_req(Flush\_complete) \\
\\
active\_computations' = \{ comp\_num? \} \ndres active\_computations \\
pending\_responses' = pending\_responses \\
pending\_ss\_requests' = pending\_ss\_requests \\
pending\_requests' = pending\_requests \\
ss\_data'.status = Flush\_done \\
\hline
\end{array}
$$

The transition $OrconSendResponseAux$ describes the basic behavior of sending a response to the kernel from the ORCON server. The $comp\_num?$ must be the locking computation, and the response must be labeled with the current $time\_stamp$. The $status$ is changed from $Flush\_done$ to $Unlocked$. Schemas $OrconSendNegativeResponse$ and $OrconSendAffirmativeResponse$ use $OrconSendResponseAux$ to model the sending of negative and affirmative responses.

$$
\begin{array}{|l}
\hline
\textit{OrconSendResponseAux} \\
SsSendResponseAux[MkRequest, OrconData, D\_SS\_REQ, D\_RESP, D\_ANS] \\
OrconStep \\
\hline
comp\_num? \in ss\_data.lock \\
ss\_data.status = Flush\_done \\
grants(ss\_response?) = grants(ss\_data.formulated\_response) \\
(ss\_response? \in \operatorname{ran} Ruling\_resp \\
\quad \Rightarrow (Ruling\_resp^{\sim}(ss\_response?)).time\_stamp \\
\qquad = ss\_data.time\_stamp) \\
\\
ss\_data'.status = Unlocked \\
\hline
\end{array}
$$

$$
\begin{array}{l}
OrconSendNegativeResponse \\
\quad \widehat{=}\ SsSendNegativeResponse[MkRequest, OrconData, D\_SS\_REQ, D\_RESP, D\_ANS] \\
\qquad \land\ OrconSendResponseAux \\
OrconSendAffirmativeResponse \\
\quad \widehat{=}\ SsSendAffirmativeResponse[MkRequest, OrconData, D\_SS\_REQ, D\_RESP, D\_ANS] \\
\qquad \land\ OrconSendResponseAux
\end{array}
$$

Any other transitions are identical to the corresponding transitions of the generic security server with the additional constraints that the ORCON policy and processing data are invariant as specified by $OrconProcessingStateInv$.

$$
\begin{array}{|l}
\multicolumn{1}{l}{OrconProcessingStateInv} \\
OrconStep \\
\hline
ss\_data'.lock = ss\_data.lock \\
ss\_data'.formulated\_response = ss\_data.formulated\_response \\
ss\_data'.required\_flushes = ss\_data.required\_flushes \\
ss\_data'.status = ss\_data.status \\
ss\_data'.ss\_pacl = ss\_data.ss\_pacl \\
ss\_data'.ss\_acl = ss\_data.ss\_acl \\
\hline
\end{array}
$$

$$
\begin{array}{l}
OrconReceiveRequest \\
\quad \widehat{=}\ SsReceiveRequest[MkRequest, OrconData, D\_SS\_REQ, D\_RESP, D\_ANS] \\
\qquad \wedge\ OrconProcessingStateInv
\end{array}
$$

The $OrconMgrRequest$ schema includes $OrconFlush$. $OrconInternalTransition$ includes $OrconBeginProcessing$ and $ProcessFlushNotification$.

$$
\begin{array}{l}
OrconMgrRequest \\
\quad \widehat{=}\ OrconFlush \\
\qquad \vee\ (SsMgrRequest[MkRequest, OrconData, D\_SS\_REQ, D\_RESP, D\_ANS] \\
\qquad\quad \wedge\ OrconProcessingStateInv) \\
OrconInternalTransition \\
\quad \widehat{=}\ OrconBeginProcessing \\
\qquad \vee\ ProcessFlushNotification \\
\qquad \vee\ (SsInternalTransition[MkRequest, OrconData, D\_SS\_REQ, D\_RESP, D\_ANS] \\
\qquad\quad \wedge\ OrconProcessingStateInv)
\end{array}
$$

Since the PACL of a process denotes the individuals allowed to read all data read by the process, we assume that, when a new process is created with context ssc, $ss\_pacl(ssc) = ORCON\_INDIV$. All individuals are allowed to read an empty collection of data.[34] The $ss\_pacl(ssc)$ will be constrained the first time the process reads an object.

10.5 Component Specification

We can use the above to specify the ORCON security server as a component.

The **guar** for the ORCON security server allows any of the transitions described in Section 10.4. This is modeled by $orcon\_guar$.

$$
\begin{array}{l}
OrconGuarStep \\
\widehat{=}\ OrconReceiveRequest \\
\quad \vee\ OrconSendNegativeResponse \\
\quad \vee\ OrconSendAffirmativeResponse \\
\quad \vee\ OrconMgrRequest \\
\quad \vee\ OrconInternalTransition
\end{array}
$$

[34] This assumes that no read permissions for this context are retained by the manager when the new process is created. Otherwise, PACL-read-consistency (see page 149) must apply.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{OrconGuar}} \\
orcon_guar : \mathbb{P}\ OrconGuarStep \\
\hline
orcon_guar = OrconGuarStep \\
\hline
\end{array}$$

The ORCON security server assumes that the assumptions of generic security servers in $SsRely$ are satisfied.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{OrconRely}} \\
orcon_rely : \mathbb{P}\ \Delta\ OrconState \\
SsRely[MkRequest, OrconData, D_SS_REQ, D_RESP, D_ANS] \\
\hline
orcon_rely = ss_rely \\
\hline
\end{array}$$

The set of allowed initial states is modeled by $orcon\_init$. We require that an initial state satisfy the constraints imposed by $SsInit$. We also require that for every object obj, $readers(obj)$ is a subset of $orcon\_readers(obj)$. This requirement is called **ACL-read-consistency**.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{OrconInit}} \\
SsInit[MkRequest, OrconData, D_SS_REQ, D_RESP, D_ANS] \\
orcon_init : \mathbb{P}\ OrconState \\
\hline
orcon_init \subseteq ss_init \\
(\forall\, st : orcon_init;\ obj : ORCON_OBJ \\
\bullet\ st.ss_data.readers(obj) \subseteq st.ss_data.orcon_readers(obj)) \\
\hline
\end{array}$$

All information in $OrconState$ is visible to the ORCON security server.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{OrconView}} \\
orcon_view : OrconState \leftrightarrow OrconState \\
\hline
\forall\, st_1, st_2 : OrconState \\
\bullet\ (st_1, st_2) \in orcon_view \Leftrightarrow st_1 = st_2 \\
\hline
\end{array}$$

$$\begin{array}{l}
OrconComponent \\
\quad \widehat{=}\ OrconGuar \wedge OrconRely \wedge OrconInit \wedge OrconView
\end{array}$$

Lemma 10.1 *In all transitions, for every object context j and subject context p such that j and p are in the domains of ss_acl and ss_pacl, respectively, both before and after the transition*

$$\begin{array}{rcl}
ss_data'.ss_acl(j)(p.user) & \subseteq & ss_data.ss_acl(j)(p.user) \\
ss_data'.ss_pacl(p) & \subseteq & ss_data.ss_pacl(p)
\end{array}$$

Proof: Since ss_acl and ss_pacl are internal data of the Security Server, they are not changed during environment transitions. Examination of the Security Server operations indicates that ss_acl and ss_pacl change only during and $OrconBeginProcessing$ transition, and this transition requires the above to hold. □

10.6 Composing DTOS and ORCON

In this section, we first compose the modified DTOS kernel with the ORCON security server. We then show that this composite system implements the abstract ORCON policy.

The composite state is $MkOrconState$ which contains all the components of $MkState$ and $OrconState$.

$$\begin{array}{l} \textit{MkOrconState} \\ \hline MkState \\ OrconState \\ \hline \end{array}$$

Each kernel state is associated with the set of all $MkOrconState$ that have the same value for each of the components of $MkState$. Similarly, each ORCON security server state is associated with the set of all $MkOrconState$ that have the same value for each of the components of $OrconState$.

The set of allowed initial states for the composition of two components is the intersection of the two sets (after mapping them into the composite state). This set of states is modeled by mk_orc_init. Since mk_orc_init is nonempty, the kernel and ORCON security server are composable.

$$\begin{array}{l} \textit{MkOrconInit} \\ \hline mk\_orc\_init : \mathbb{P}\ MkOrconState \\ \hline \forall\, st : mk\_orc\_init;\ obj : ORCON\_OBJ \\ \bullet\ st.ss\_data.readers(obj) \subseteq st.ss\_data.orcon\_readers(obj) \\ \quad \land\ st.pending\_responses = \varnothing \\ \quad \land\ st.pending\_requests = \varnothing \\ \quad \land\ st.active\_request = \varnothing \\ \quad \land\ st.sent = \varnothing \\ \quad \land\ st.obtained = \varnothing \\ \quad \land\ st.allowed = \varnothing \\ \quad \land\ st.responses = \varnothing \\ \quad \land\ st.active\_computations = \varnothing \\ \hline \end{array}$$

In composing the kernel and ORCON security server we will use respect relations that require each component to leave alone its peer's internal data.

$$\begin{array}{l} \textit{MkOrconRespect} \\ \hline mk\_respect\_orc : \mathbb{P}\ \Delta\ MkOrconState \\ orc\_respect\_mk : \mathbb{P}\ \Delta\ MkOrconState \\ \hline mk\_respect\_orc = \{\ \Delta\ MkOrconState \\ \qquad \mid \Xi\ SsInternals[MkRequest, OrconData, D\_SS\_REQ, D\_RESP, D\_ANS]\ \} \\ orc\_respect\_mk = \{\ \Delta\ MkOrconState \\ \qquad \mid \Xi\ MgrInternals[MkData, MkRequest, D\_SS\_REQ, D\_RESP, D\_ANS]\ \} \\ \hline \end{array}$$

The **guar** of the composite is denoted by mk_orc_guar.

$$\begin{array}{l} MkOrconStep \\ \widehat{=}\ (\Xi\ SsInternals[MkRequest, OrconData, D\_SS\_REQ, D\_RESP, D\_ANS] \\ \quad \land\ MkGuarStep) \\ \lor\ (\Xi\ MgrInternals[MkData, MkRequest, D\_SS\_REQ, D\_RESP, D\_ANS] \\ \quad \land\ OrconGuarStep) \end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad MkOrconGuar \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
mk_orc_guar : \mathbb{P}\ MkOrconStep \\
\hline
mk_orc_guar = MkOrconStep \\
\hline
\end{array}$$

The **rely** of the composite is the intersection of the two rely relations.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad MkOrconRely \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
mk_orc_rely : \mathbb{P}\ \Delta\ MkOrconState \\
\hline
mk_orc_rely = \{\ \Delta\ MkOrconState \\
\qquad \mid \Xi\ MgrInternals[MkData, MkRequest, D_SS_REQ, D_RESP, D_ANS] \\
\qquad \land\ \Xi\ SsInternals[MkRequest, OrconData, D_SS_REQ, D_RESP, D_ANS] \\
\qquad \land\ \Xi\ SharedInterpretation[MkRequest, D_SS_REQ, D_RESP, D_ANS] \\
\qquad \land\ pending_ss_requests \sqsubseteq pending_ss_requests' \\
\qquad \land\ pending_responses' = pending_responses\ \} \\
\hline
\end{array}$$

To show that the Composition Theorem applies to the composite, we must show that

$$mk\_guar \cap mk\_respect\_orc \subseteq orcon\_guar \cup orcon\_rely \cup orcon\_view,$$

and

$$orcon\_guar \cap orc\_respect\_mk \subseteq mk\_guar \cup mk\_rely \cup mk\_view.$$

The proof of these properties is essentially identical to the corresponding proof for the composition of the generic manager and security server.

We now demonstrate that the composition of the ORCON security server and the DTOS microkernel satisfies the high-level requirements of the ORCON security policy. We will focus on the primary requirement, ignoring the requirement of invariance for $process\_individual(p)$. The goal will be to prove that in any possible transfer sequence the final reader has read permission in $orig\_acl$ to the object that is at the start of the transfer sequence.

In order to complete the proof, we will need the following additional assumptions about the initial state:[35]

PACL-read-consistency — For every request r, if $retained(r)$ contains $Perm\_req(pr)$ where $pr.perm = Have\_read$, then $ss\_pacl(ssi\_ssc(pr.ssi)) \subseteq readers(osi\_osc(pr.osi))$.

PACL-write-consistency — For every request r, if $retained(r)$ contains $Perm\_req(pr)$ where $pr.perm = Have\_write$, then $readers(osi\_osc(pr.osi)) \subseteq ss\_pacl(ssi\_ssc(pr.ssi))$.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad MkOrconInitExtra \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
MkOrconInit \\
\hline
\forall\ st : mk_orc_init; r : MkRequest; pr : PermReq \\
\mid Perm_req(pr) \in st.retained(r) \\
\bullet\ \mathbf{let}\ pacl == st.ss_data.ss_pacl(st.ss_data.ssi_ssc(pr.ssi)); \\
\qquad rdrs == st.ss_data.readers(st.ss_data.osi_osc(pr.osi)) \\
\qquad \bullet\ (pr.perm = Have_read \Rightarrow pacl \subseteq rdrs) \\
\qquad \land\ (pr.perm = Have_write \Rightarrow rdrs \subseteq pacl) \\
\hline
\end{array}$$

[35] An alternative way to express these assumptions within the composability framework is to add $retained$ to the security server state $OrconState$ and to the view. We could then place these assumptions in $OrconInit$ instead of making them hypotheses of Theorem 10.2.

Theorem 10.2 *Let s be a subsequence of length m of a trace t, let p be a process with context c_p, and j an object with context c_j and assume $(s, j, p) \in \mathit{Transfer\_possible}$. Assume the system is initially consistent (see 8.7), ACL-read-consistent, PACL-read-consistent and PACL-write-consistent and that processes do not have concurrent kernel requests. Then the individual associated with p in the final transition of s is allowed by $\mathit{orig\_acl}$ to read j.*

Proof: The proof is by induction on the length of s. Note that this length must be odd, so in the base case we will consider a sequence of length 1, and the inductive step will increment by 2. To support the inductive step we will also prove the auxiliary conclusion that at the start of the final transition in s, $\mathit{ss\_pacl}(c_p) \subseteq \mathit{orcon\_readers}(j)$.

Base Case: $m = 1$. The application $s(1)$ denotes the processing of a read operation on object j in state $\mathit{Appl\_state}(s(1))$ by process p. We consider two cases.

Base Case 1: p has been granted $\mathit{Have\_read}$ permission to j.

At some previous time, $\mathit{ss\_acl}(c_j)(c_p.\mathit{user})$ contained $\mathit{Have\_read}$. Monotonicity implies that this is also true of $\mathit{ss\_acl}$ in the initial state. The assumption of ACL-read-consistency completes the proof of the main conclusion in this case. When the $\mathit{OrconBeginProcessing}$ step for the $\mathit{Have\_read}$ permission request by p on j concludes, $\mathit{ss\_pacl}(c_p) \subseteq \mathit{readers}(c_j)$. The auxiliary conclusion follows from ACL-read-consistency and the monotonicity of $\mathit{ss\_acl}$ and $\mathit{ss\_pacl}$.

Base Case 2: p has never been granted $\mathit{Have\_read}$ permission to j.

In the initial state, p must have $\mathit{Have\_read}$ permission for j retained. The main conclusion follows from initial consistency, InAcl, and ACL-read-consistency. The auxiliary conclusion follows from PACL-read-consistency and monotonicity of $\mathit{ss\_pacl}$.

Inductive Step: *Assuming the auxiliary conclusion holds for all sequences of length n (for some odd number $n \geq 1$), the main and auxiliary conclusions must hold for all sequences of length $n + 2$.*

Let w be the final writer in s (i.e., the process in application $s(n + 1)$) and let c_w be the context of w. Let k, with context c_k, be the final object in s. For $s(n + 1)$ to be processed there must be a preceding transition in which this operation is marked as allowed. The same applies to $s(n + 2)$, the final read operation in s. For these requests to be marked allowed, one of the following must be true:

1. The relevant permission ($\mathit{Have\_write}$ or $\mathit{Have\_read}$) was retained from the initial state,
2. The kernel *obtained* the permission while doing the permission checking for the request, or
3. This permission was retained from the permission checking for an earlier request during which it was obtained.

We now consider the following cases:

Case 1: The $\mathit{Have\_write}$ permission used for $s(n + 1)$ is retained from the initial state.

Whenever $\mathit{ss\_pacl}(c_w)$ changes, it is due to the granting of $\mathit{Have\_read}$ permission to w, and all $\mathit{Have\_write}$ permissions for w are flushed before the $\mathit{Have\_read}$ permission is granted. Furthermore, the security server completes the processing of each permission request before starting another. So, $\mathit{ss\_pacl}(c_w)$ does not change from the time $\mathit{Have\_write}$ is retained (the initial state in this case) through the transition in which $s(n + 1)$ is marked allowed. Since w does not have concurrent requests, the invariance of $\mathit{ss\_pacl}(c_w)$ extends through the processing of $s(n + 1)$. From PACL-write-consistency and the inductive hypothesis we conclude that in the initial state

$$\mathit{readers}(c_k) \subseteq \mathit{ss\_pacl}(c_w) \subseteq \mathit{orcon\_readers}(j) \tag{1}$$

From initial consistency, $InAcl$, and the monotonicity of ss_acl we know that, whether the $Have\_read$ permission for $s(n+2)$ is initially retained or is obtained at some point, $c_p.user$ must be in $readers(c_k)$ in the initial state for p to read k. This demonstrates the main conclusion. Similarly, the auxiliary conclusion follows from (1), PACL-read-consistency and the monotonicity of ss_pacl.

Case 2: The $Have\_write$ permission used for $s(n+1)$ is obtained.

At the end of the $OrconBeginProcessing$ transition associated with the granting of the $Have\_write$ permission, we have $readers(c_k) \subseteq ss\_pacl(c_w)$. Following the same reasoning as in Case 1, $ss\_pacl(c_w)$ is invariant from the end of the $OrconBeginProcessing$ transition through the end of $s(n+1)$. (Note that this $OrconBeginProcessing$ transition might be associated with a request other then $s(n+1)$.) If $s(n)$ precedes the $OrconBeginProcessing$ transition, then by the inductive hypothesis and monotonicity of ss_pacl, at the end of this transition, we know $ss\_pacl(c_w) \subseteq orcon\_readers(j)$. If $s(n)$ follows, then $s(n)$ occurs in the period during which $ss\_pacl(c_w)$ is invariant. By the inductive hypothesis, at the end of the $OrconBeginProcessing$ transition we know $ss\_pacl(c_w) \subseteq orcon\_readers(j)$.

Thus, at the end of the $OrconBeginProcessing$ transition

$$readers(c_k) \subseteq ss\_pacl(c_w) \subseteq orcon\_readers(j) \tag{2}$$

We now further divide this case into two subcases.

Case 2a: The $Have\_read$ permission used for $s(n+2)$ is obtained after the $Have\_write$ permission for $s(n+1)$ is obtained.

The main and auxiliary conclusions follow from the monotonicity of ss_acl and ss_pacl, respectively, together with (2).

Case 2b: The $Have\_read$ permission used for $s(n+2)$ is *not* obtained after the $Have\_write$ permission for $s(n+1)$ is obtained.

We first note that, because k is locked against writing from the time at which permission checking for $s(n+2)$ begins through the processing of $s(n+2)$, $s(n+1)$ must be processed before the checking for $s(n+2)$ begins. This means the $Have\_read$ permission was not obtained during the permission checking for $s(n+2)$. It must be retained either from an earlier permission check or from the initial state.

Let t_w be the state in which $Have\_write$ becomes available for $s(n+1)$, and let t_r be the state from which $Have\_read$ is retained for $s(n+2)$ (i.e., either the initial state, or the state in which the $Have\_read$ most recently becomes available). We know t_r precedes t_w. Assume the processing of some permission request alters $ss\_acl(c_k)$ between t_r and t_w. This would generate a flush request which, due to the security server locking protocol, must be completed before t_w. This flush request would remove all $Have\_read$ permissions to k, but this means that $Have\_read$ cannot be retained from t_r through $s(n+2)$, and this is a contradiction. Thus, the $Have\_write$ permission check for $s(n+1)$ is made while $ss\_acl(c_k)$ has the same value as it does in state t_r, and $readers(c_k) \subseteq orcon\_readers(j)$ in t_r. The main conclusion follows directly from this. The auxiliary conclusion follows from the monotonicity of ss_pacl. □

Section 11 Conclusion

We have used a formal specification language, Z [32], to model a generic framework for systems containing an object manager that enforces policy decisions made by a security server. The DTOS microkernel has been specified as an instance of the generic manager and three example security servers have been modeled and analyzed. A lattice of policy characteristics has also been developed to study the implications that the separation of enforcement from decision-making has on the policy flexibility of a system. The key factors limiting policy flexibility under this separation are

- the interface between the object manager and security server,
- the ability to retract permissions, and
- the degree of synchronization required for dynamic policies.

An example of an interface limitation is that a manager will only send selected pieces of information to the security server in a permission request. Any policy that requires additional information may be difficult or impossible to implement. For example, the DTOS microkernel sends a requested permission identifier and a pair of security identifiers in a permission request. Any policy that requires, for example, a requested priority level or the amount of memory to be allocated could not be implemented. Early versions of DTOS were not completely capable of retracting permissions because Mach caches the permissions in the page protection bits. Although this has been resolved in later releases, there are still issues regarding the order in which interactions between the manager and security server are processed and the degree of synchronization required. Great care is required when implementing and analyzing a security server for a retractive policy.

It has been found in this study that the DTOS kernel supports MLS/TE with little difficulty. It only supports the piecemeal version of Clark-Wilson since the kernel will request permissions only with respect to a single pair of SIDs. Of course, this relates to the more fundamental inability of Mach tasks to request access to multiple objects in a single request. DTOS, as is, does not support ORCON, but with some modification it could. We reiterate that the failure of the kernel to support any policy is not a reflection on the general DTOS architecture, but only on the particular DTOS kernel. A modified kernel or a new object manager, together with an appropriate security server, could support Pure Clark-Wilson or ORCON on the objects that it manages.

Section *12*

Notes

12.1 Acronyms

ACL Access Control List

CDI Constrained Data Item

CMU Carnegie Mellon University

DDT Domain Definition Table

DTOS Distributed Trusted Operating System

FSPM Formal Security Policy Model

IBAC Identity Based Access Control

IPC InterProcess Communication

MLS Multi-Level Secure

ORCON Originator Controlled

OSC Object Security Context

OSI Object Security Identifier

PACL Propagated Access Control List

SSC Subject Security Context

SSI Subject Security Identifier

TE Type Enforcement

TP Transformation Procedure

UDI Unconstrained Data Item

12.2 Glossary

access control list: a column of the security policy matrix; a list of (principal, permitted access set) pairs.

capability: an object identifier and a set of accesses

environment: a row of the security policy matrix representing the permissions for a principal; a list of capabilities.

security identifier: the name by which a principal or an object is identified to the Security Server.

security policy: "the set of laws, rules, and practices that regulate how an organization manages, protects, and distributes sensitive information" [21].

12.3 Open Issues

- The formalization of the various sensitivities is not entirely satisfying. For example, the formalization of history sensitivity is too dependent upon the implementation of a policy rather than the policy itself. (This criticism also applies to the static/dynamic distinction.) Furthermore, the history sensitivity characteristic is probably too broad; a manager might support some types of history sensitivity but not others. It is also rather difficult to formally distinguish the other types of sensitivity from history-sensitivity as defined. This suggests that the identified types of sensitivity are not an ideal set of characteristics for use in the analysis of policies.
- When defining the policy characteristics and placing policies into the policy lattice it was at times necessary to consider how a policy might be implemented rather than the abstract properties of the policy itself. This suggests that it might be more appropriate to focus entirely on ways of implementing policies rather than on the policies themselves. Further work would be needed to determine if this leads to a better classification scheme.
- This study suggests that the problems of obtaining policy flexibility may be exacerbated by the level of separation between manager and security server present in the DTOS architecture. It is an open question to what degree this is actually true. Answering this question would require performing an analysis similar to the one in this study but with a more tightly coupled manager and security server. Of course there would also be disadvantages to this tighter coupling. For example, if the security server code were actually incorporated into the managers, then we could not change the security server code without at least relinking the managers. This, too, is counter to policy flexibility. The ideal solution most likely lies in solving the problems discussed in this report.
- Another possible security policy characteristic to which we have given some thought is whether each entity must have a unique SID and context. However, there are some subtleties to this question, and we have not included it in our list.

  Consider Clark-Wilson for example. Assume that individual i is authorized to execute TP p on each of the sets of CDIs $\{a, b, c\}$ and $\{d, e\}$, but no other CDI sets. Assume i creates one task t_1 for executing p on a, b and c and then, while t_1 is still executing, asks to create another task t_2 for executing p on d and e. There are two options: consider t_1 and t_2 to be separate Clark-Wilson processes, or consider them to be two tasks operating within the same Clark-Wilson process.[36] If we choose the latter, that means that i may only have one process at a time executing a given TP. It also makes it imperative that the manager notify the security server when a process is terminated so that the TP can be executed again. In this case t_2's requests for access to d and e must be denied since i is not allowed to have a process executing p on $\{a, b, c, d, e\}$. To ensure this, we must give the same SID to t_1 and t_2.

  If we choose to consider the two tasks to be different Clark-Wilson processes, then the second request should be allowed. We also must be careful that t_1 are t_2 are separated in such a way that t_1 cannot access d and e and t_2 cannot access a, b, and c. To achieve this, we will need different SIDs on t_1 and t_2. In fact, every Clark-Wilson process must have a unique SID. This approach is more flexible than the other one since an individual may simultaneously run p on two different sets of CDIs. It also makes it less crucial that the security server be notified of process termination.

  To answer the question of whether unique SIDs are needed we must decide how abstract concepts such as Clark-Wilson processes correspond to low-level, labeled entities such as

[36] We assume the former in our specification of Clark-Wilson.

tasks, threads and ports. The above example suggests that, for Clark-Wilson, it would be good to have a unique SID for each task. We suspect that this will also be true for many other policies.

The DTOS design, but *not* the kernel, offers as least some support for this 1-1 relationship. A security policy may be defined in such a way that the default version of task creation (i.e., **task_create**) in which the SID is inherited from the parent task is not allowed. Furthermore, **task_create_secure**, in which a SID for the new task is specified as a parameter, is disallowed unless the specified SID is different from that of any other task. The DTOS design contains a security server request **SSI_transition_domain**[37] that may be used to obtain a SID for use in creating a new task. This means that the task submitting the **task_create_secure** need not guess an unused SID but can simply use **SSI_transition_domain** to ask for one. This will typically require modified versions of all applications that create tasks through explicit kernel calls. Since the operating system would probably be modified to request distinct SIDs automatically during process creation, applications that create new tasks only through operating system calls need not be modified.

- The current generic security server specification requires that a permission be granted only if it is in $policy\_allows$. For history based policies it might be convenient to also allow the granting of permissions that are in $policy\_allows'$. Consider a policy in which a request for a permission A causes permission B to become grantable. To avoid an extra security server interaction, the designer of a server might decide to grant B along with A. This behavior does not pose a threat to any consistency theorem since by the time B can be used by the manager it will be in $policy\_allows$ (assuming it is not removed after being granted). This can be viewed as an issue of atomicity. An implementation of the security server could first add B to the policy and then send the response granting both A and B. This two step process could be collapsed into a single transition in the specification. Of course, the specification could also use a smaller atomicity breaking the larger transition into two smaller ones corresponding to those described for the implementation.

[37] This request was not added to support a 1-1 relationship, but it can be used for this purpose if desired.

Appendix A
Bibliography

[1] Martín Abadi, Michael Burrows, Butler Lampson, and Gordon Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, September 1993.

[2] Marshall D. Abrams. Renewed understanding of access control policies. In *Proceedings 16th National Computer Security Conference*, pages 87–96, Baltimore, MD, September 1993.

[3] D. Elliott Bell and Leonard J. La Padula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, The MITRE Corporation, May 1973.

[4] K. J. Biba. Integrity considerations for secure computer systems. Technical Report TR-3153, Mitre Corp., Bedford, MA, 1977. Also available through Nat'l Technical Information Service, Springfield, Va., Report No. NTIS AD-A039324.

[5] W. E. Boebert and R. Y. Kain. A practical alternative to hierarchical integrity policies. In *Proceedings 8th National Computer Security Conference*, pages 18–27, Gaithersburg, MD, October 1985.

[6] David F. C. Brewer and Michael J. Nash. The Chinese wall security policy. In *IEEE Symposium on Security and Privacy*, pages 206–214, Oakland, CA, May 1989.

[7] Maureen Harris Cheheyl, Morrie Gasser, George A. Huff, and Jonathan K. Millen. Verifying security. *ACM Computing Surveys*, 13(3):279–339, September 1981.

[8] David D. Clark and David R. Wilson. A comparison of commercial and military computer security policies. In *IEEE Symposium on Security and Privacy*, pages 184–194, Oakland, CA, April 1987.

[9] Ellis Cohen and David Jefferson. Protection in the Hydra operating system. In *Proceedings of the Fifth Symposium on Operating Systems Principles, Operating Systems Review 9,5*, pages 141–160, Austin, TX, November 1975.

[10] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[11] Todd Fine, J. Thomas Haigh, Richard C. O'Brien, and Dana L. Toups. Noninterference and unwinding for LOCK. In *Proceedings of Computer Security Foundations Workshop II*, pages 22–28, Franconia, NH, June 1989. IEEE.

[12] Joseph A. Goguen and José Meseguer. Security policy and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, Oakland, CA, April 1982.

[13] G. Scott Graham and Peter J. Denning. Protection – principles and practice. In *Proceedings AFIPS 1972 SJCC*, volume 40, pages 417–429. AFIPS Press, 1972.

[14] Richard Graubart. On the need for a third form of access control. In *Proceedings 12th National Computer Security Conference*, pages 147–156, Baltimore, MD, October 1989.

[15] Anita K. Jones and William A. Wulf. Towards the design of secure systems. *Software – Practice and Experience*, 5:321–336, 1975.

[16] Catherine Jensen McCollum, Judith R. Messing, and LouAnna Notargiacomo. Beyond the pale of MAC and DAC – defining new forms of access control. In *IEEE Symposium on Security and Privacy*, pages 190–200, Oakland, CA, May 1990.

[17] Daryl McCullough. Noninterference and the composability of security properties. In *IEEE Symposium on Security and Privacy*, pages 177–186, Oakland, CA, April 1988.

[18] John McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proceedings IEEE Computer Society Symposium on Research in Security and Privacy*, Oakland, CA, May 1994.

[19] Catherine Meadows. Extending the Brewer-Nash model to a multilevel context. In *IEEE Symposium on Security and Privacy*, pages 95–102, Oakland, CA, May 1990.

[20] Michael J. Nash and Keith R. Poland. Some conundrums concerning separation of duty. In *IEEE Symposium on Security and Privacy*, pages 201–207, Oakland, CA, May 1990.

[21] NCSC. Trusted computer systems evaluation criteria. Standard, DOD 5200.28-STD, US National Computer Security Center, Fort George G. Meade, Maryland 20755-6000, December 1985.

[22] Richard C. O'Brien and Clyde Rogers. Developing applications on LOCK. In *Proceedings 14th National Computer Security Conference*, pages 147–156, Washington, DC, October 1991.

[23] Edward A. Schneider, Stanley Perlo, and David Rosenthal. Discretionary access control mechanisms for distributed systems. Technical Report RADC-TR-90-275, Rome Air Development Center, June 1990.

[24] Secure Computing Corporation. DTOS Composability Study. DTOS CDRL AO20, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, September 1995.

[25] Secure Computing Corporation. DTOS Covert Channel Analysis Plan. DTOS CDRL AO17, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, April 1995.

[26] Secure Computing Corporation. DTOS Covert Channel Analysis Report. DTOS CDRL AO07, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, May 1995.

[27] Secure Computing Corporation. DTOS Formal Security Policy Model. DTOS CDRL AO04, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, September 1996.

[28] Secure Computing Corporation. DTOS Formal Security Policy Model (Non-Z Version). Technical report, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, September 1996.

[29] Secure Computing Corporation. DTOS Formal Top-Level Specification. DTOS CDRL AO05, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, December 1996.

[30] Secure Computing Corporation. DTOS Kernel and Security Server Software Design Document. DTOS CDRL A002, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, December 1996.

[31] Secure Computing Corporation. DTOS Kernel Interfaces Document. DTOS CDRL AO03, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, January 1997.

[32] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International, 1992.

[33] Phil Terry and Simon Wiseman. A 'new' security policy model. In *IEEE Symposium on Security and Privacy*, pages 215–228, Oakland, CA, May 1989.

[34] Thomas Y. C. Woo and Simon S. Lam. Authorization in distributed systems: A new approach. *Journal of Computer Security*, 1994.