

© Copyright by Timothy Jon Fraser, 1997

**AN OBJECT-ORIENTED FRAMEWORK FOR
SECURITY POLICY REPRESENTATION**

BY

TIMOTHY JON FRASER

B.S., Worcester Polytechnic Institute, 1994

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1997

Urbana, Illinois

AN OBJECT-ORIENTED FRAMEWORK FOR SECURITY POLICY REPRESENTATION

Timothy Jon Fraser, M.S.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1997
Professor Roy Campbell, Advisor

Traditionally, trusted operating systems are designed to enforce one particular security policy. This policy is usually a static form of mandatory access control (MAC) based on a hierarchy of sensitivity labels, which is strongly oriented towards defense applications. [3] Unfortunately, this sort of policy is generally not well suited for use in commercial environments. The fact that the policy is a non-configurable aspect of the operating system prevents commercial organizations from adapting these systems for their own use. Historically, this has led to a lack of useful trusted operating systems in the commercial sector.

According to the Department of Defense Goal Security Architecture (DGSA), the key to designing commercially useful trusted operating systems is to provide them with security policy decision functions (SPDFs) which take descriptions of policies as input. [2] Instead of being limited to enforcing a single static policy, such an operating system would be capable of dynamically changing the policy it enforces by taking in new policy descriptions during run-time. These descriptions could encompass a wide variety of policy forms, allowing commercial organizations to specify policies tailored to their specific operational needs.

The process of taking a policy description as input requires some sort of interpretable policy description mechanism. This thesis describes an object-oriented framework which is well suited for this purpose, due to its generality and extensibility. It discusses the architecture of the framework, its external interfaces to an SPDF, and its role in the larger process of policy description. It also documents a number of policy examples represented with a prototype implementation of the framework and tested with a simple SPDF simulator.

I would like to thank Dr. David Gomberg and Jody Heaney of the MITRE corporation, and Professor Roy Campbell of the University of Illinois for their invaluable advice and support during the course of this project.

Chapter

1	Security Policy Representation	1
1.1	The Need for a General Security Policy Representation Mechanism	2
1.2	Policy Development and Management	3
1.3	The Security Policy Representation Framework	5
1.4	Safety of the Framework Approach	7
1.5	Generality of the Framework Representation Mechanism	8
1.6	Working with Dynamic Policy Behavior	8
2	Framework Architecture	10
2.1	Primitives	12
2.2	Operating System Entities	13
2.3	External Interfaces	16
2.4	Generic Discretionary Access Control Framework	18
2.5	Generic Non-Discretionary Access Control Framework	24
2.6	Generic Device Specific Policy	31
2.7	Device-aware Non-discretionary Access Control Policy	35
2.8	Domain-oriented Non-discretionary Access Control	36
2.9	Support for Security Policy Decision Functions	38
3	Using the Framework	39
3.1	The PrarieSoft DSP Scenario	39
3.2	The Aerospace Collaboration DONDAC Scenario	43
3.3	Alternate Uses for the Framework	49
4	Analysis	50

Appendix

A	Framework Representation State for PrarieSoft DSP	52
B	Framework Representation State for Aerospace Collaboration DON- DAC policy	55
	Bibliography	63

List of Tables

3.1	DSP rules for PrarieSoft's Routers	42
3.2	Employees of SOT, and the domains they may access.	43

List of Figures

1.1	Conceptual View of an Interpreting SPDF	4
1.2	Conceptual View of the SMDE Policy Development Process	5
2.1	Component-level Map of the Framework	11
2.2	Key for Class Hierarchy Diagrams	12
2.3	Basic Primitives Class Hierarchy Diagram	13
2.4	Operating System Entity Class Hierarchy Diagram	14
2.5	External Interface Class Hierarchy Diagram	16
2.6	Generic Discretionary Access Control Class Hierarchy Diagram	19
2.7	Flowchart of the DAC Decision-making Process Highlighting the Role Played by External Mechanisms	22
2.8	Generic Non-discretionary Access Control Class Hierarchy Diagram	25
2.9	Generic Device Specific Policy Class Hierarchy Diagram	32
3.1	The PrarieSoft Network Topology Showing the Names of Significant Network Interface Devices	40
3.2	The Flow of Information between the DUM, SOT, and ZZZ Infor- maiton Domains	44
3.3	Components of the SOT Distributed System	46
3.4	The SPDF Simulator Waiting for Input	47
3.5	SPDF Simulator Indicating that Davis is not Allowed to Read ZZZ informaiton.	48

Chapter 1

Security Policy Representation

This chapter describes the need for general policy representation mechanisms in trusted operating systems, and shows how an object-oriented framework can be used to fill that need.¹ The philosophy behind the framework approach, and its role in the overall process of policy management are described in detail. Special emphasis is placed on the commercial usefulness of systems equipped with this sort of representation mechanism, since this is a major motivating factor behind the research in this field.

This section deals with the framework approach in an abstract fashion. It leaves the discussion of the actual framework implementation's concrete features and flaws to later sections. Section 2 describes the architecture of the prototype framework implementation, summarizing the functionality of each component and detailing the areas where the design might be improved. This is followed by section 3, which presents a series of policy representation examples constructed with the prototype framework implementation and exercised with a simple SPDF simulator. Finally, section 4 presents an analysis of the practical limits to the usefulness of the framework, and contrasts the framework approach with language-based solutions to the interpretable policy representation problem.

¹The term 'trusted' usually denotes an operating system which has been evaluated to at least a B1 rating according to the Department of Defense's Orange Book criteria. This rating has a direct bearing on the government's procurement procedures, and even commercial organizations consider it to be analogous to the approval of the Underwriters Laboratories in the world of consumer electronics.

1.1 The Need for a General Security Policy Representation Mechanism

Most modern operating systems are designed to enforce some sort of security policy. In order to accomplish this, a particular operating system must contain mechanisms which can make policy decisions, and mechanisms which can enforce them. Depending on its design, these mechanisms may be spread throughout several of the system's modules. However, for the sake of discussion, it is often helpful to imagine that all of these mechanisms are gathered into one component called a 'reference monitor'. [1] The part of the reference monitor that is responsible for making decisions may be referred to as the 'security policy decision function', or SPDF. [2]

Typically, the details of the policy enforced by an operating system are hard-coded into its SPDF. This means that a given operating system will only be able to enforce one kind of policy. Traditional implementations of UNIX, for example, support a kind of discretionary access control (DAC). More complex versions exist which also support forms of mandatory access control (MAC).² The MAC-supporting versions are the most interesting from a security perspective, since DAC support by itself is not sufficient for most organization's security needs. [1]

MAC is the policy form described by the Bell-LaPadula model. Generally speaking, MAC policies divide information into a collection of hierarchically related domains, each identified by a label. These labels are usually derived from the Department of Defense's document sensitivity labeling scheme ("Top Secret", "Secret", etc.). The rules of the policy order the domains from 'highest' to 'lowest', and ensure that information cannot flow 'downward'.

Although this hierarchical arrangement is natural for the purposes of the military intelligence community, it is not well suited for use in commercial organizations. [2] The information resources of corporations are most easily described as a collection of separate and largely unrelated domains. For example, a company might have a domain for each department ("finance", "human resources", etc.). It is difficult to order all of these domains into a single chain where each is "higher" or "lower" than another.

²Examples include Trusted Information System's Trusted XENIX and Trusted Mach, and Argus Systems Group's Trusted Solaris.

As described above, MAC is the most common policy among the ‘trusted’ operating systems. With the details of MAC hard-coded into its SPDF, configuring an operating system to enforce a different policy is not an option. This leaves commercial organizations with little choice but to make do with MAC, or look to stop-gap measures outside of the operating system to provide security. Most choose the latter option. [4] This is a tragic situation, since the need for secure operating systems in today’s commercial world is great, and is apt to become greater with time.

According to the Department of Defense Goal Security Architecture (DGSA), the key to solving this problem is to avoid the practice of hard-coding policy details into the SPDF. Instead, SPDFs should be designed to take policy descriptions as input. These descriptions would contain all of the details necessary for the SPDF to make decisions. The resulting SPDF would be much like a traditional interpreter, and the policy descriptions might be represented in an interpreted language specialized for policy representation.

This design would allow a wide variety of organizations to make use of the same operating system. In order to tailor the operating system’s policy enforcing behavior to meet their individual needs, each organization could provide the SPDF with a description of their particular security policy. Since new policies can be given to the SPDF as the system runs, the system’s policy enforcing behavior could be modified dynamically. Policies for each of an organization’s information domains could be described separately. This would allow new domains and policies to be added and old ones removed without shutting down the system for reconfiguration. Figure 1.1 contains a diagram describing the conceptual layout of this sort of SPDF. Due to this ability to enforce organization-specific policies, operating systems equipped with this sort of SPDF would be useful in a wide variety of commercial and military environments.

1.2 Policy Development and Management

The task of describing policies in a form that may be interpreted by an SPDF is not trivial. As mentioned above, solutions to this problem are apt to resemble specialized interpreted programming languages. This is because both problems share the same need for a human-readable high-level description language, a low-level machine-interpretable form, and some sort of automated translator to convert between them.

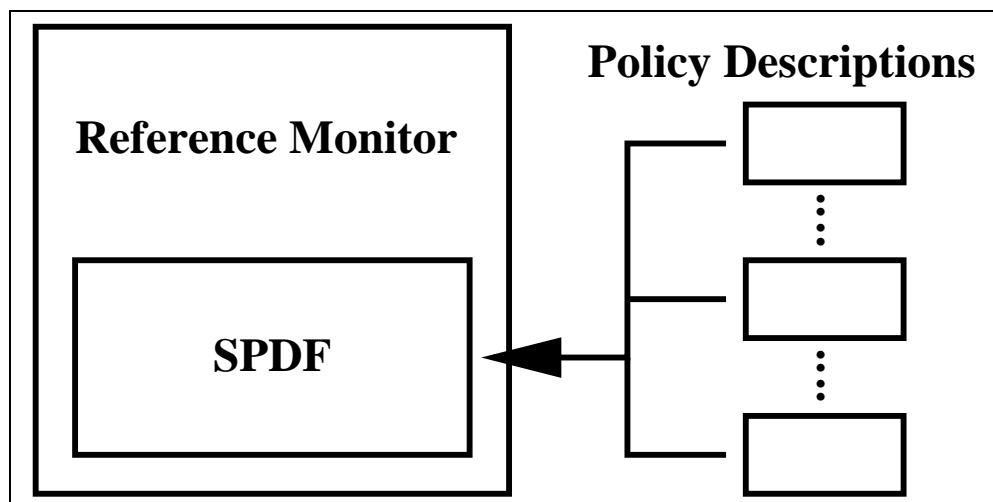


Figure 1.1: Conceptual View of an Interpreting SPDF

Ideally, an operating system with an interpreting SPDF would be accompanied by an integrated environment for policy development and description analogous to those provided for general programming. Such an environment would be required to ease the task of converting an organization's overall information security policy as expressed on paper into a form the SPDF can interpret. Fortunately, an example of such an environment already exists. It is called the Security Model Development Environment, or SMDE. [5]

As shown in figure 1.2, the SMDE divides the overall policy description process into three parts. The topmost is the 'conceptual level', where an organization's information security policy is initially developed and described in natural language on paper.

Below this is the 'modeling level' where the relevant parts of the policy expressed in natural-language are translated into a high-level policy description language. This language serves the same purpose as a high-level programming language - it is a formal notation capable of avoiding the ambiguities of natural language, but is still at a high enough level of abstraction to allow easy manipulation by people. The language used in SMDE is called the Common Notation, or CN, and is loosely based on the Ada general programming language.

The lowest level is the 'rule level', which describes the policy in a low-level SPDF-interpretable form referred to as a 'rule base'. The translation from CN to rule base

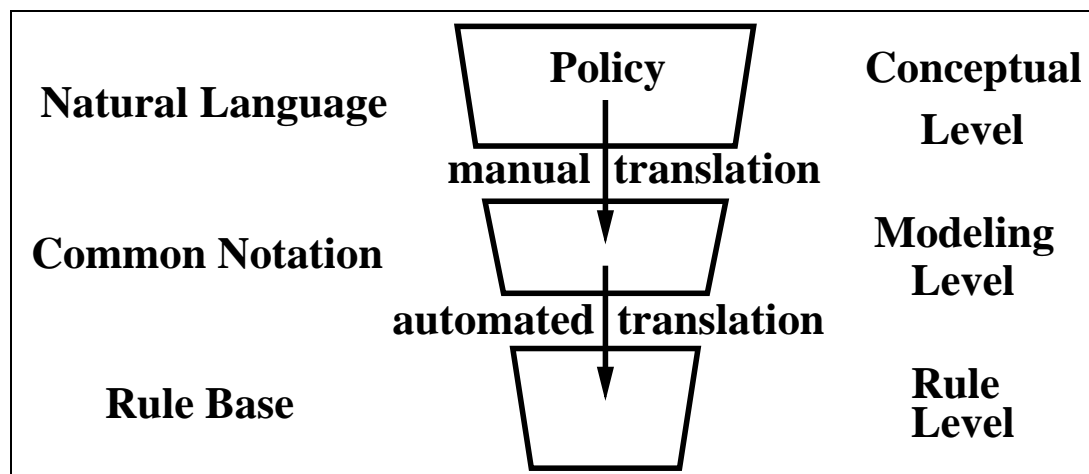


Figure 1.2: Conceptual View of the SMDE Policy Development Process

is a fully automated process handled by a ‘Model Translator Tool’ (MTT) whose purpose is analogous to a compiler. A number of support tools are provided with the MTT to form something that might be described as an integrated debugging environment for policy descriptions.

With an environment similar to SMDE, a reasonably competent programmer would be capable of expressing the relevant aspects of an organization’s information security policy in a high-level description language. The environment would then translate this into a form that an operating system’s SPDF could interpret. This would allow organizations to take advantage of the policy configuration features of the operating systems described above without having to deal with overwhelming complexity. Alternately, organizations might purchase ready-made policy descriptions from third-party vendors and adjust them for their own purposes.

1.3 The Security Policy Representation Framework

The security policy representing framework described by this thesis is intended to serve the same purpose as the SMDE rule base. It also provides a considerable amount of the functionality normally required of an SPDF. Conceivably, it could serve as a component of a complete policy development environment similar to SMDE with the addition of a higher level description language and some translation and debugging facilities.

The framework is interesting both because of its approach to the interpretable policy description problem, and its expressive capabilities. The most straightforward approach to the description problem is to create a low-level interpreted language and an interpreter to evaluate it. This interpreter would be made part of the SPDF. This approach requires a considerable amount of software development.

The framework approach avoids the necessity of creating a new language to describe policy and an interpreter to evaluate it. Instead, it makes use of an existing interpreted object-oriented programming language. The framework provides a hierarchy of useful policy-representing classes. Policies are described by writing short programs which make use of these classes. The programming language's interpreter can be used to execute these programs, and thus evaluate the policy description. This approach leads to a policy description mechanism that is neither more nor less expressive than one based on a new specialized language. However, it is significantly easier to develop, since it makes use of the software already provided for the programming language - most notably the interpreter.

The framework itself is a hierarchy of classes. The classes at the bottom of the framework are mostly abstract, and are mainly used to represent mathematical concepts such as sets and mappings. These classes form the basis for a hierarchy of successively more and more specialized classes representing concepts such as labels and access control lists. Finally, at the top of the framework are classes which can be used to represent a variety of generic policy forms, including (but not limited to) forms of discretionary and non-discretionary access control.

The framework's classes not only encapsulate the state required to represent policies and evaluate policy decisions, but also the methods required to actually accomplish the evaluation. Thus, the highest-level policy representing classes possess methods which accept parameters describing situations requiring a decision, evaluate the policy described by the class, and return a meaningful result. Since the framework provides the lion's share of the functionality required to evaluate policy decisions, SPDFs designed to work with the framework need to implement only a minimal level of functionality themselves.

The design features required to allow an operating system to take advantage of a representation mechanism like the framework are not overwhelming. Given an operation system that is typical by present standards, only two additional features

are required to make it suitable. First, it must be designed with a reference monitor and SPDF component as described above. This is not a difficult requirement to satisfy, since these components are already an integral part of current trusted operating system designs. Second, in order to exploit the dynamic policy reconfiguration capabilities provided by the framework, the operating system would have to support dynamic loading and unloading of components during run-time. Although support for this kind of functionality is currently rare, neither of these requirements are unattainable with current technology.

1.4 Safety of the Framework Approach

At first glance, it may seem that giving the framework responsibility for such a large part of the decision-making process may put the integrity of an SPDF at risk whenever there is the possibility of encountering a maliciously constructed policy description. To some extent, SPDFs based on the specialized policy description language approach are also susceptible to this problem. So, to allay these concerns, it is sufficient to show that the framework approach is merely as safe as the specialized language approach.

An SPDF based on the framework possesses its own local copy of the framework code. Since this code is an integral part of the SPDF, it can be verified and protected from modification in the same fashion as the other components of the operating system kernel. Thus, the framework itself can be considered free of malicious behavior, leaving only the policy descriptions themselves to consider.

In reality, policy descriptions are programs which create instances of classes from the framework. The SPDF can ensure that the only classes available to the programs it interprets are those that are part of its framework. This eliminates most of the troublesome functionality that stems from the framework's basis in an unmodified general programming language.

The remaining risks concern the parameters passed to the various framework methods. A malicious policy author might attempt to confuse the SPDF by passing semantically incorrect parameters to methods (such as null parameters, or parameters of unexpected types). Or, he might pass semantically correct parameters which are carefully chosen to take advantage of some unintended feature of the SPDF. Both of these vulnerabilities are shared by the specialized language approach. So the framework approach is at least as safe as a specialized language. In addition, careful

use of a type-safe programming language can help the framework prevent abuses based on semantically incorrect parameters.

1.5 Generality of the Framework Representation Mechanism

In order to be useful in a commercial environment, an operating system must be capable of enforcing a wide variety of policy forms. Thus, a good policy representation mechanism must be similarly versatile. The framework described by this thesis provides direct support for forms of DAC which are based on access control lists. In addition, a significant amount of the framework is devoted to representing a number of generic non-discretionary access control policy forms. This includes support for MAC. The framework also is capable of expressing policies which simply require certain mechanisms to be invoked under certain circumstances. (This policy is described more fully in section 2.6.)

Unfortunately, since the universe of policy forms is countably infinite, it is not possible to say that the framework is capable of expressing all possible forms, or even that it can express a given fraction of them. However, due to its object-oriented nature, it is not difficult to extend the framework to support policy forms that were overlooked in the original design. Section 2.7 discusses how two of the policy forms supported by the framework were combined to produce a third, providing a practical example of the framework's extensibility. Section 3 demonstrates the framework's applicability to a variety of commercial environments by presenting a number of sample policies represented with the framework.

1.6 Working with Dynamic Policy Behavior

The inclusion of an interpreting SPDF in a reference monitor increases the complexity of the operating system kernel. Since it allows the policy enforcing behavior of the system to vary, it makes the task of reasoning about the system's security properties much more difficult. Since it is often desirable to prove theorems about a system's security-relevant behavior in order to show that it can be trusted to enforce a given security policy, this increase in complexity is an important practical concern.

There are several ways to deal with the adverse effects of dynamic behavior. The most direct approach might be to restrict the system to enforcing a finite set of

well-known policies which have all been subjected to formal analysis. New policies might be added to the set only after careful examination. Alternately, in situations requiring a greater degree of dynamic behavior, the SPDF itself might provide some assistance. For instance, the SPDF might be capable of declaring that a certain subset of the policies it enforces are not unloadable during run-time. This would still allow organizations to initially configure the behavior of their systems, but once running, part of their behavior would be essentially static. Conceivably, this approach might be generalized to include support for meta-policies which govern how the behavior of the SPDF may be changed.

Ultimately, the cost of the additional complexity is far outweighed by the value provided by dynamic policy behavior. The needs of commercial industry demand operating systems whose policy enforcing behaviors are dynamically configurable. Current operating systems which enforce a single static security policy cannot hope to meet the individual needs of various organizations. The first operating system to effectively support dynamic policy enforcing behavior will be the first to be truly useful in a commercial environment.

In order to be practical, this dynamic policy enforcing behavior must be supported by a general policy representation mechanism which is capable of representing whatever organization-specific policies are necessary. The framework described in the following chapters is just such a mechanism.

Chapter 2

Framework Architecture

This chapter describes the architectural details of the prototype framework implementation. The prototype framework is a hierarchy of useful policy-representing classes implemented as a Java package. Different parts of the framework are specialized to represent different forms of policy. The hierarchy can be divided into eight logical components based on this policy-specific specialization. Figure 2.1 contains a diagram showing each of these components and their interrelationships.

The following sections deal with each component in turn. Each section begins by describing the overall structure of the component, and then details the specific function of each of the component's classes. Each class is given its own subsection. The headings for these subsections reflect the name of the class being described, and the name of its superclass. Most sections also provide a diagram of the relationships between the component's classes. Figure 2.2 explains the meanings of the symbols used in these diagrams.

As with most prototypes, the implementation not only serves to prove the general concept of the design, but also draws attention to the weak areas where the design was flawed. In recognition of this fact, most of the following sections conclude with a discussion of design alternatives concerning the component they describe, with an emphasis on how the component might be improved in a later prototype.

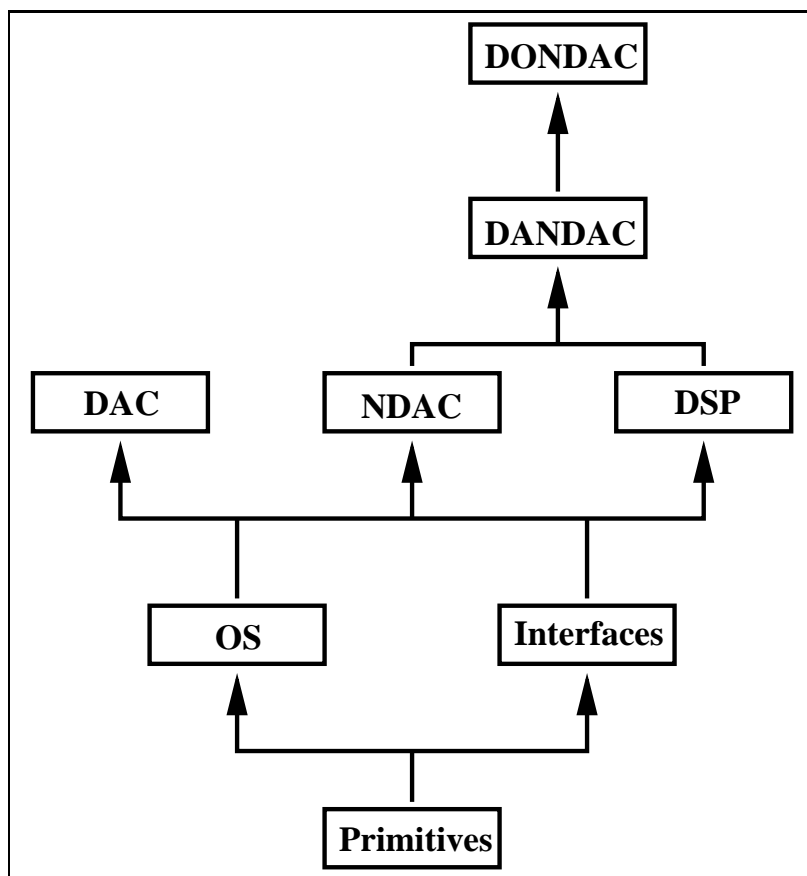


Figure 2.1: Component-level Map of the Framework

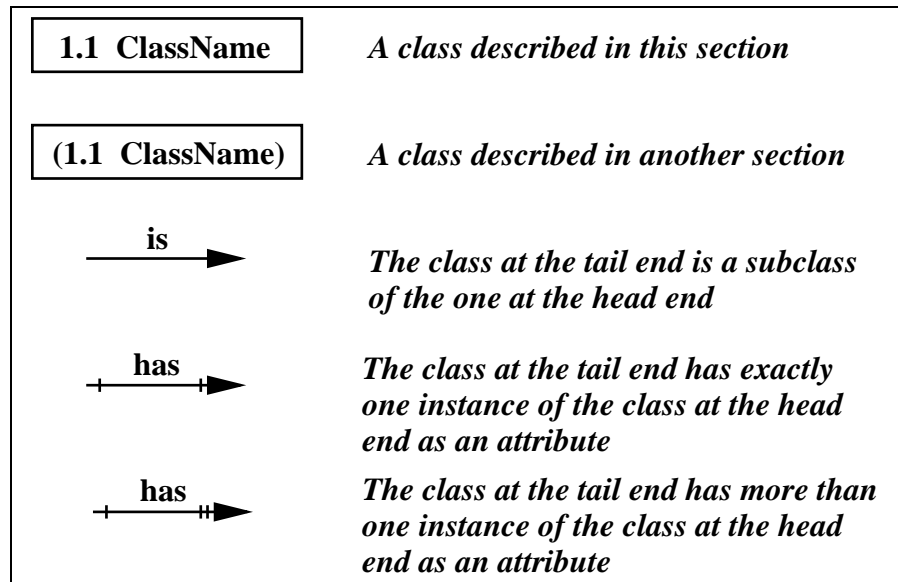


Figure 2.2: Key for Class Hierarchy Diagrams

2.1 Primitives

This section describes the most basic primitives provided by the framework. These classes are the atoms from which the rest of the framework is built. A diagram of the relationships between these classes is shown in figure 2.3.

2.1.1 PolicyRepresentationObject

This is the parent of all of the objects in the framework. Although it provides no policy-representing functionality, it does contain some abstract utility methods. The most important of these methods is an equality predicate. Each child of this class implements this method, which is used to determine if the attributes of two objects of the same class have the same values.

2.1.2 Enumeration : PolicyRepresentationObject

This abstract class describes the concept of an enumerated type - that is, a sequence of unique identifiers. Specialized subclasses of this class provide unique identifiers for users and objects, and lists of application-specific operations.

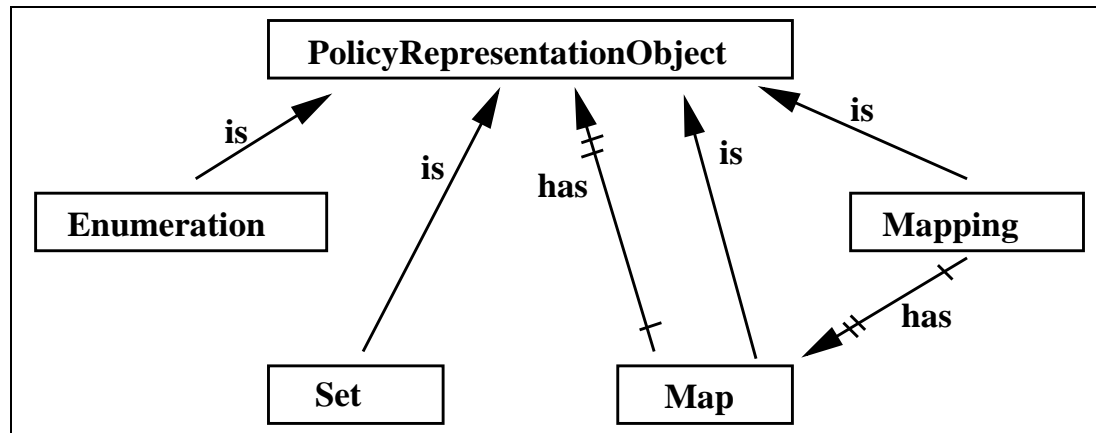


Figure 2.3: Basic Primitives Class Hierarchy Diagram

2.1.3 Set : PolicyRepresentationObject

This class represents the mathematical concept of a set. It is actually implemented as an ordered list of PolicyRepresentationObjects, so that it can provide methods to support iteration. This allows the framework to support ‘there exists’ and ‘for all’ expressions over the membership of sets.

2.1.4 Map : PolicyRepresentationObject

Instances of the Map class are used to associate pairs of PolicyRepresentationObjects. Lists of Maps are used by the Mapping class to represent the mathematical concept of a mapping.

2.1.5 Mapping : PolicyRepresentationObject

The Mapping class represents the mathematical concept of a Mapping. Instances of the Map class are used to link individual PolicyRepresentationObjects from the ‘domain’ to individual PolicyRepresentationObjects from the ‘codomain’. Instances of the Mapping class contain lists of these objects, which together describe the entire mapping.

2.2 Operating System Entities

This section of the framework provides classes which may be used to represent various operating system entities, such as labels and processes. A diagram of the relationships between these classes is shown in figure 2.4.

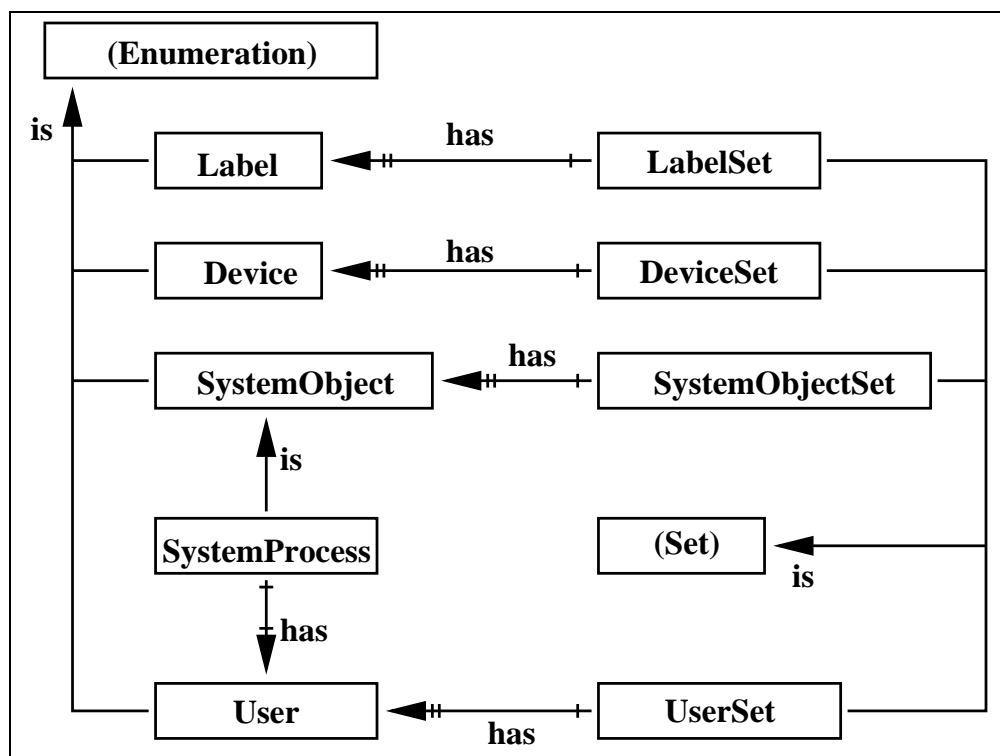


Figure 2.4: Operating System Entity Class Hierarchy Diagram

2.2.1 Label : Enumeration

The Label class provides the ability to represent enumerated sequences of labels. Instances of this class may be used to describe groups of information domain names or sensitivity label schemes.

2.2.2 LabelSet : Set

The LabelSet class represents a set of Labels.

2.2.3 SystemObject : Enumeration

The SystemObject class represents policy-relevant system objects, including files and servers. It represents them as simple unique identifiers.

2.2.4 SystemObjectSet : Set

The SystemObjectSet class represents a set of SystemObjects.

2.2.5 SystemProcess : SystemObject

The SystemProcess class represents the concept of a process. SystemProcesses are a form of SystemObject, which means they may be uniquely identified. Each System-Process is also associated with a particular user.

2.2.6 User : Enumeration

The User class represents the identities of individual users as simple unique identifiers.

2.2.7 UserSet : Set

The UserSet class represents a set of Users.

2.2.8 Device : Enumeration

The Device class represents hardware devices as simple unique identifiers. Policy-relevant devices might include various forms of stable storage media, printers, and network interfaces.

2.2.9 DeviceSet : Set

The DeviceSet class represents a set of Devices.

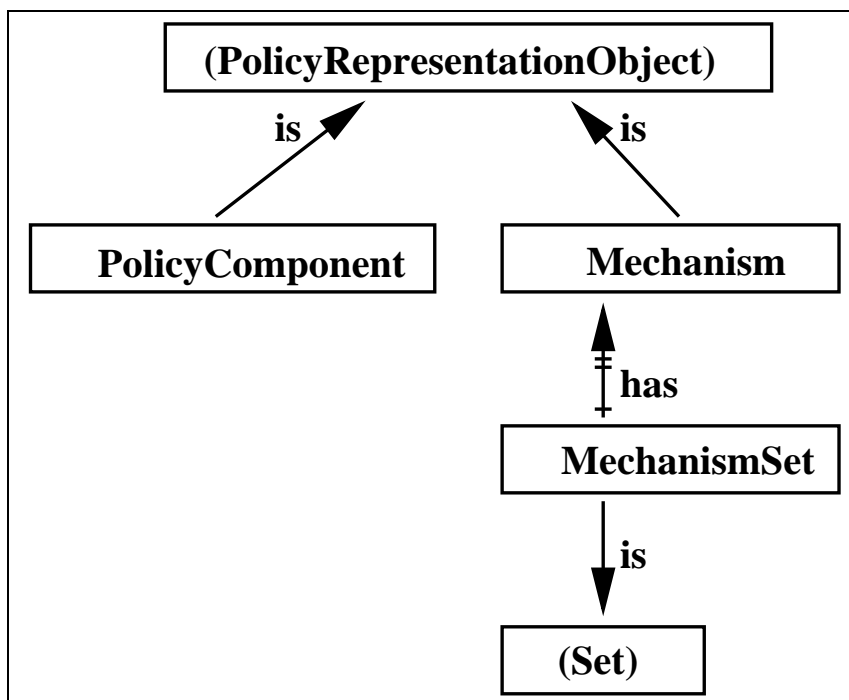


Figure 2.5: External Interface Class Hierarchy Diagram

2.3 External Interfaces

Policy representations based on the framework have two interfaces with the rest of the system. One interface is with the SPDF that interprets them. This interface is encapsulated by the **PolicyComponent** class. The other interface is with various enforcement mechanisms external to the SPDF. This is encapsulated by the **Mechanism** class. Both of these classes are described below.

2.3.1 PolicyComponent : PolicyRepresentationObject

Instances of the **PolicyComponent** class encapsulate a single policy, such as an instance of MAC or DAC. It provides an interface between the policy and the SPDF.

The framework does not specify how the SPDF should be designed. However, it is likely that the SPDF would contain a class called "Policy" which would be capable of encapsulating several instances of **PolicyComponent**. This would allow the **Policy** class to represent a composite policy made up of several specialized components. The **Policy** class might contain a method called "Evaluate" which would take in parameters describing a situation requiring a policy decision, and would invoke the

proper methods of the PolicyComponent instances it contains to make the decision. It might also contain methods to support the dynamic loading and unloading of PolicyComponents, which would allow the SPDF to be dynamically reconfigured during run-time.

Each subclass of PolicyComponent implements a method named 'Evaluate'. In the subclasses, the Evaluate method is responsible for evaluating a policy decision when given the proper parameters. Different policies require different parameters. The primary challenge of implementing an SPDF using this framework is ensuring that it passes the proper parameters to the Evaluate methods of each of the PolicyComponent instances it contains.

2.3.2 Mechanism : PolicyRepresentationObject

The Mechanism class encapsulates the framework's interface to various external policy-enforcing mechanisms. These mechanisms might include various forms of encryption, methods to manage digital signatures, or the like. Mechanisms may return a Boolean value after they have been invoked, which may be used as a criterion in an access control decision.

Each subclass of Mechanism implements a method named 'Invoke'. This method is basically a stub function that is responsible for passing the proper parameters provided by the framework to the actual external mechanism. Like the Evaluate method of the PolicyComponent class, the Invoke method is apt to require different parameters for different forms of policy. In all of the policy forms supported by the prototype version of the framework, the parameters passed to Invoke are the same as those passed to Evaluate. That is, external mechanisms are also given enough parameters to uniquely identify the situation requiring a policy decision.

2.3.3 MechanismSet : Set

The MechanismSet class simply represents a set of Mechanisms.

2.3.4 Design Issues

As described above, the Evaluate method of PolicyComponent and the Invoke method of Mechanism require different parameters in each of their specializations. This resulted in a considerable number of overloaded methods in the prototype implementation. Since one of the goals of the framework is to be as safe as possible, a considerable amount of care was necessary to avoid situations where an inappropriate base method

might be invoked rather than an overridden version with different parameters.

Since both the Evaluate and Invoke methods are apt to take the same parameters, an improved implementation might include a class named ‘Situation’ which would encapsulate all of the parameters for these methods. Sections of the framework devoted to different policy forms could specialize this class to contain the particular parameters they need. Safety could be assured by the Java’s type checking system, which would prevent the passing of inappropriate specializations of Situation to Evaluate and Invoke methods.

2.4 Generic Discretionary Access Control Framework

This section describes the part of the framework devoted to representing forms of discretionary access control, or DAC. The DAC classes can be used to represent policies similar to the access controls found in the UNIX and Windows NT file systems.¹ They can also be used to represent policies which govern access to services provided by server applications or the operating system kernel itself.

Figure 2.6 provides a diagram of the DAC component of the framework. Despite its cluttered appearance, the hierarchy the diagram represents is not too complicated. The hierarchy is made up of two parts, one shown on the right topped by the AccessControlList class, and one shown on the left topped by the GenericDACPolicy class. The AccessControlList class represents the concept of an access control list. Access control lists are the primary data structure for DAC policy description. Each instance of AccessControlList is capable of describing all of the DAC policy rules which apply to a single SystemObject (such as a file or server).

The GenericDACPolicy class is a child of PolicyComponent. It encapsulates the entire DAC policy concept by specifying the mapping between SystemObject instances and their instances of AccessControlList. It also provides an Evaluate method which evaluates the contents of AccessControlLists and makes DAC policy decisions. Evaluate requires three parameters to uniquely identify a DAC situation: a User, a SystemObject, and an Operation.

¹UNIX is a trademark of AT&T. Windows NT is a trademark of the Microsoft corporation.

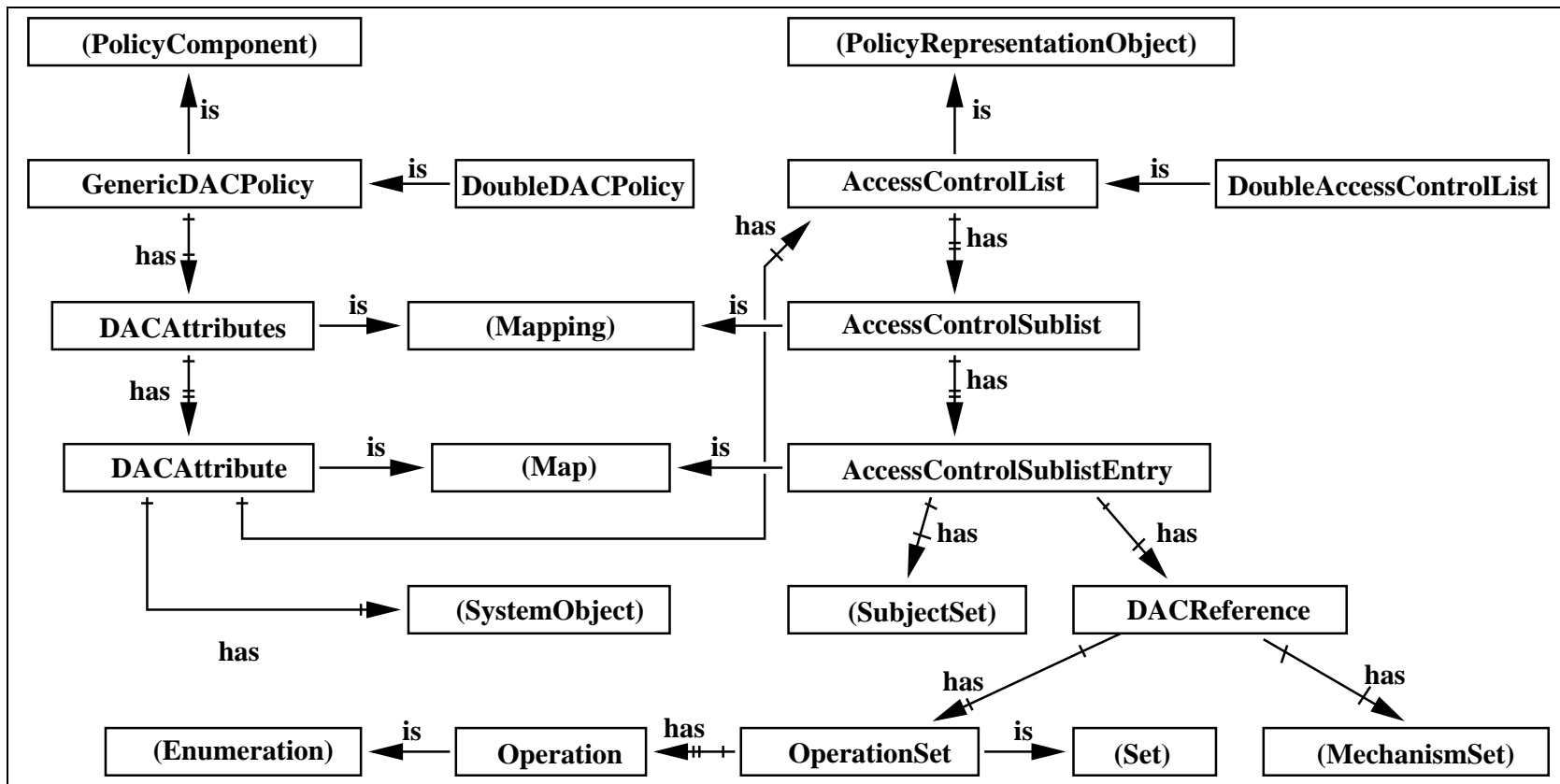


Figure 2.6: Generic Discretionary Access Control Class Hierarchy Diagram

The DAC component of the framework is capable of representing DAC policy forms that are more complex than those found in UNIX and Windows NT. As described below, the framework allows each entry in an access control list to specify sets of external mechanisms (see section 2.3.3). These mechanisms may be used to augment the SPDF's decision-making ability, or to trigger the invocations of various security mechanisms whenever certain types of references occur.

Because of this mechanism interface, the DAC part of the framework is capable of representing a policy statements like “only administrators are allowed to shut down router machines, and only when they are logged in locally, at the console.” In this example, when a user attempts to access the shutdown utility, the list can be checked to determine if they are an “administrator”, according to the normal DAC procedure. Because of the inclusion of a Mechanism interface, a function external to the SPDF can be invoked to determine whether or not they are logged in locally. This information can then be used as a decision-making criterion.

2.4.1 GenericDACPolicy : PolicyComponent

The GenericDACPolicy encapsulates the entire DAC policy concept, including the mapping between SystemObjects and their AccessControlLists, and the methods needed to evaluate DAC decisions. This class is capable of handling UNIX-style DAC, where instances of AccessControlList only specify operations which are allowed. In cases where AccessControlList instances must specify both allowed operations and a blacklist of disallowed operations, the DoubleDACPolicy class may be used.

2.4.2 AccessControlList : PolicyRepresentationObject

The AccessControlList is a generalized form of the traditional access control list concept. It may be associated with any form of SystemObject (section 2.2.3), from files to servers. Each instance of AccessControlList contains an instance of AccessControlSublist which can be used to list allowed operations. If a secondary blacklist of disallowed operations is also required, then the DoubleAccessControlList class should be used instead of AccessControlList.

2.4.3 AccessControlSublist : Mapping

The AccessControlSublist class corresponds roughly to the traditional idea of an access control list. It is a list of AccessControlSublistEntry instances, each of which describes a User-Operation pair.

2.4.4 AccessControlSublistEntry : Map

Instances of the AccessControlSublistEntry class associate a set of users with an instance of the DACReference class. The set of Users corresponds to the UNIX/NT concept of a group - that is, a collection of users. If the set contains only one User, then it corresponds directly to one user. The DACReference class specifies a set of operations. Thus, the AccessControlSublistEntry class can be used to represent the traditional user-operation pairing needed for access control lists.

2.4.5 DACReference : PolicyRepresentationObject

Each instance of DACReference contains a set of operations. This set describes the set of operations in the UserSet-OperationSet pairs maintained by each instance of AccessControlSublistEntry. The DACReference class also encapsulates two instances of MechanismSet (section 2.3.3), which is a departure from the traditional DAC policy form found in UNIX and Windows NT.

The first MechanismSet specifies a group of mechanisms which may be used as predicates to augment the SPDF's decision-making abilities. All of the mechanisms in this set must be invoked by the SPDF before it makes a decision. All of the mechanisms must return a Boolean result of true in order for the operation to be allowed. Useful predicates might include mechanisms which consult some state of the system external to the SPDF, such as file system quotas, security logs, or perhaps some system-wide security alertness level.

The second MechanismSet specifies a group of mechanisms which must be invoked by the SPDF after a decision has been made to allow the operation. Since their invocation occurs after the decision has been made, these mechanisms cannot be used as predicates. However, this facility may be used to ensure that particular enforcement mechanisms are invoked whenever an operation described by a particular DACReference instance occurs. Appropriate mechanisms might include invocations of encryption or auditing functions.

Figure 2.7 contains a flowchart which details the sequence of events from the initial DAC-relevant reference made by a user's process, through the SPDF's decision-making process, to the point where control is returned to the process. It illustrates the role of external mechanisms described above. Mechanisms play a similar role in the other parts of the framework devoted to forms of access control, such as NDAC (section 2.5).

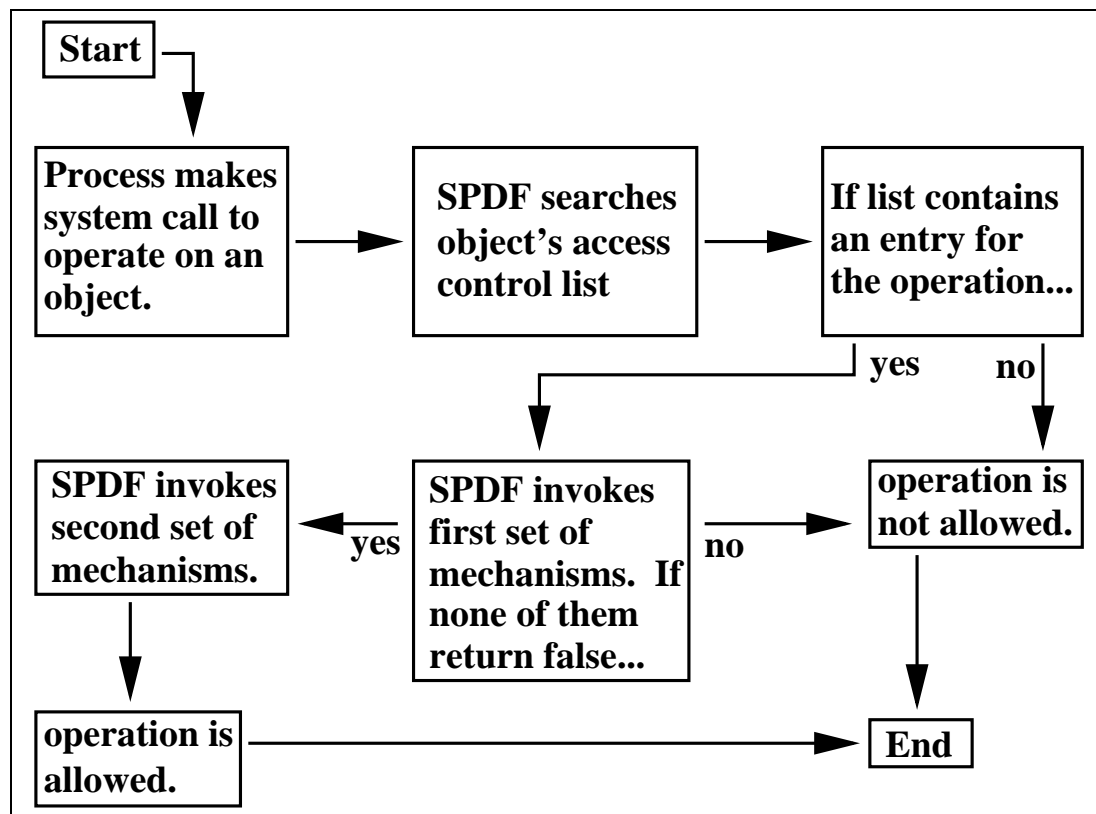


Figure 2.7: Flowchart of the DAC Decision-making Process Highlighting the Role Played by External Mechanisms

2.4.6 Operation : Enumeration

Instances of Operation can be used to represent collections of operations, ranging from “read, write, execute” for files, to “log in remotely” for rlogin servers.

2.4.7 OperationSet : Set

The OperationSet class simply represents a set of operations.

2.4.8 DACAttribute : Map

Instances of the DACAttribute class may be used to associated an instance of AccessControlList with an instance of SystemObject.

2.4.9 DACAttributes : Mapping

Each instance of the DACAttributes class encapsulates a list of DACAttribute instances, each of which associates an AccessControlList with a SystemObject. An SPDF may use this class to maintain the system-wide mapping of AccessControlLists to SystemObjects if no other adequate facility is available elsewhere in the system.

2.4.10 DoubleDACPolicy : GenericDACPolicy

The DoubleDACPolicy class is a specialization of the GenericDACPolicy class that has the added ability to handle AccessControlList instances which specify both allowed operations and a blacklist of disallowed operations. This allows the framework to represent DAC policy forms similar to the one found in Windows NT. [6]

2.4.11 DoubleAccessControlList : AccessControlList

The DoubleAccessControlList class is a specialization of the AccessControlList class which is encapsulates two instances of class AccessControlSublist. The first instance is used to list allowed operations, as in the AccessControlList class. The second is used as a blacklist of operations that are not allowed, even if the first AccessControlSublist indicates that they should be allowed.

2.4.12 Design Issues

The addition of Mechanisms to the traditional DAC scheme raises some interesting issues concerning the Evaluate methods of the GenericDACPolicy and DoubleDACPolicy classes. As mentioned above, both methods take a User, SystemObject, and Operation as parameters.

The algorithm used by `GenericDACPolicy`'s `Evaluate` is relatively straightforward. It begins retrieving the `AccessControlList` instance associated with the specified `SystemObject`. It then searches this list for an entry matching the specified `User` and `Operation`. The operation is allowed if and only if an appropriate entry is found, and any external mechanisms it specifies are invoked by the SPDF and return a favorable result. If the operation is allowed, a set of further mechanisms may be returned to the SPDF for later invocation.

The algorithm actually implemented in the prototype framework makes the unrealistic assumption that at most one entry will apply to a given situation. In a scheme that does not involve mechanisms, this assumption does no harm. But when using a framework-based scheme, it is a potential problem. If two entries apply to a given situation, it is possible that the external mechanisms specified by one will lead to the operation being allowed, while the other will lead to denial. Depending on which entry occurs first in the list, the behavior of the algorithm will be different. Furthermore, the prototype implementation will only cause the SPDF to invoke the external mechanisms found in the first applicable entry. In some situations, it may be desirable to invoke the external mechanisms specified by all of the applicable entries, instead of just those specified by the first one.

The `Evaluate` method of the `DoubleDACPolicy` class uses a somewhat more complicated algorithm that searches through two lists per `SystemObject` rather than one. The first list is used as a blacklist, and may be used to deny operations that are allowed by the second list. The same external-mechanism-related problems exist with this algorithm as with the simpler `GenericDACPolicy` algorithm, except that they occur in two searches rather than one. The usefulness of the DAC component of the framework might be significantly increased by the correct implementation of evaluation algorithms that address these issues.

2.5 Generic Non-Discretionary Access Control Framework

This section describes the part of the framework devoted to representing forms of non-discretionary access control (NDAC). NDAC policies associate labels with every policy-relevant object in the system. These labels divide the universe of objects into several disjoint sets. Some forms of NDAC refer to these sets as “information domains” [2] Once all of the objects in the system have been organized into separate

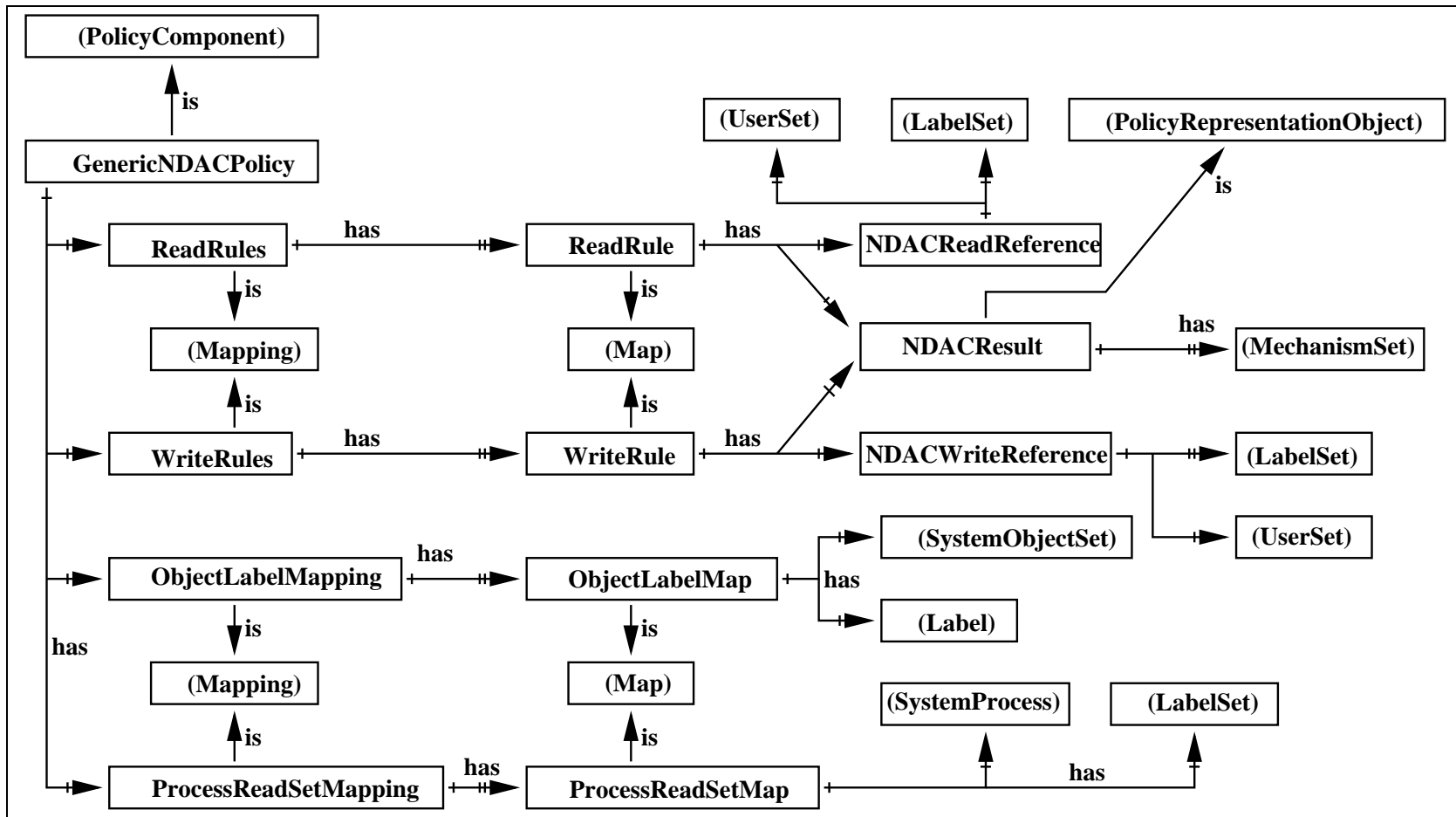


Figure 2.8: Generic Non-discretionary Access Control Class Hierarchy Diagram

information domains, the NDAC policy can specify rules governing which users are allowed read and write access to which information domains, and how objects may be transferred from one domain to another. The Mandatory Access Control policy described briefly in section 1.1 is a common example of an NDAC policy.

In order to make NDAC decisions, the SPDF needs access to a certain amount of information on the dynamic state of the system. Specifically, at any given time, it must be able to determine which information domains have been accessed by a given processes since the beginning of its lifetime. (In this context, “process” refers to the idea of a UNIX process, or a Mach task.) The framework provides classes which an SPDF may use to keep track of this state. Alternately, these classes can be ignored in favor of some pre-existing state-tracking mechanism located elsewhere in the operating system.

As with the DAC policy form, the SPDF also needs access to some relatively static state information, including the mapping of labels to policy-relevant objects in the system, and the mapping of users to processes.

The structure of the NDAC part of the framework is diagrammed in figure 2.8. Although this diagram is even more crowded than figure 2.6, it also represents a relatively simple hierarchy. `GenericNDACPolicy` class is the topmost class in this part of the framework hierarchy. It contains four mappings. The first two describe the rules of the policy, and the second two maintain information on the state of the system.

The first mapping is for the `ReadRules` (section 2.5.5), which indicate which process are allowed to read objects from which information domains. This decision is based on the identity of the user who owns the process, and the label associated with the object being read. The second mapping is for the `WriteRules` (section 2.5.7), which indicate the allowable writing behavior for processes after they have read objects from certain combinations of information domains. The third mapping is the `ObjectLabelMapping` (section 2.5.3), which indicates which labels are associated with which objects. Finally, the fourth mapping is the `ProcessReadSetMapping` (section 2.5.12), which dynamically keeps track of which labels have been read by each process in the system.

As with the discretionary access control part of the framework (section 2.4), Mechanisms (section 2.3.2) have been made part of the decision process. Each in-

stance of `ReadRule` or `WriteRule` may specify a pair of `MechanismSet` instances. One set describes a group of external mechanisms that must be invoked before the NDAC decision is made, and the other describe a group which must be invoked after. These mechanisms serve the same purpose as the mechanisms in the DAC part of the framework. They allow the NDAC part of the framework to represent policy statements such as “The public relations officer may write data from the ‘Project Albatross’ domain into the ‘Public Information’ domain if and only if he has digitally signed the data using the system’s digital signature management application.” An external mechanism might provide a predicate-like interface to the system’s digital signature management application which handles the signature checking prescribed by the policy.

2.5.1 **GenericNDACPolicy : PolicyComponent**

The `GenericNDACPolicy` encapsulates the entire NDAC concept, including the four mappings described above and the methods needed to evaluate decisions. All situations requiring an NDAC policy decision can be uniquely identified by specifying an instance of `SystemProcess`, an instance of `SystemObject`, and an operation consisting of ‘read’ or ‘write’. NDAC policies are only concerned with these two operations, since they are the fundamental operations provided by hardware memory. [1] Given the above three parameters, the `GenericNDACPolicy` class’s `Evaluate` method can evaluate the NDAC policy and return a Boolean result indicating that the read or write operation should be allowed (true), or not allowed (false).

The `Evaluate` method begins its decision-making algorithm by consulting the system state to determine which `User` is associated with its `SystemProcess` parameter, and which `Label` is associated with its `SystemObject` parameter. Its subsequent behavior depends on whether the operation specified by its third parameter is a read or a write.

In the case of a read operation, the `Evaluate` method’s parameters indicate that a process working for a particular `User` is trying to read an object with a particular `Label`. The method searches through the `ReadRules` mapping looking for a `ReadRule` instance that applies to this particular combination of `User` and `Label`. If no such rule is found, then the method returns false.

If an applicable rule is found, then it may require the SPDF to invoke a number of external mechanisms before it can make the decision. In this case, the default

decision is true, but Evaluate will return false if any of the external mechanisms invoked by the SPDF return false. If the result of the decision does turn out to be true, the Evaluate method will also provide the SPDF with a list of additional external mechanisms to invoke after it returns.

The behavior of the Evaluate method is somewhat different in the case of a write operation. Before it can make a decision, it must consult the system state to determine the set of Labels which the SystemProcess has read since the beginning of its life. (This set of Labels corresponds conceptually to a set of information domains.) It then searches the WriteRules mapping for a WriteRule instance corresponding to the situation described by this set, the User corresponding to the System process attempting the write, and the Label of the SystemObject being written.

From this point on, the decision algorithm is quite similar to the one used in the read situation described above. As before, the write operation is not allowed if an appropriate rule is not found. If a rule is found, the decision may depend on the results of some external mechanism invocations.

2.5.2 ObjectLabelMap : Map

Instances of the ObjectLabelMap class associate an instance of the Label class (section 2.2.1) with an instance of SystemObjectSet (section 2.2.4). The ObjectLabelMapping class maintains a list of ObjectLabelMap objects.

2.5.3 ObjectLabelMapping : Mapping

The ObjectLabelMapping class encapsulates a list of ObjectLabelMap objects. If no other facility exists in the operating system, the SPDF may use instances of ObjectLabelMapping to maintain the system-wide mapping of Labels to SystemObjects.

2.5.4 ReadRule : Map

Instances of the ReadRule class associate an instance of the NDACReadReference class with an instance of NDACResult. The NDACReadReference is used to describe a class of read situations requiring an NDAC policy decision. The NDACResult instance is used to describe the pair of MechanismSets indicating which mechanisms should be invoked before and after a decision is made. Together, they describe an allowable class of NDAC read situations, and the external mechanisms that must be invoked to handle them.

2.5.5 ReadRules : Mapping

The ReadRules class encapsulates a list of ReadRule objects. Instances of ReadRules can be used to represent the policy concerning the reading behavior of all processes in the system.

2.5.6 WriteRule : Map

Instances of the WriteRule class associate an instance of the NDACWriteReference class with an instance of NDACResult. The NDACWriteReference is used to describe a class of write situations requiring an NDAC policy decision. The NDACResult instance is used to describe the pair of MechanismSets indicating which mechanisms should be invoked before and after a decision is made. Together, they describe an allowable class of NDAC write situations, and the external mechanisms that must be invoked to handle them.

2.5.7 WriteRules : Mapping

The WriteRules class encapsulates a list of WriteRule objects. Instances of WriteRules can be used to represent the policy concerning the writing behavior of all processes in the system. As described above, write situations that are not explicitly allowed by some WriteRule object in the list are implicitly disallowed.

This arrangement allows a concise representation of policies which specify information domains that are largely independent. However, a very large number of WriteRule instances are required to represent MAC-like policies whose information domains are related by a complicated hierarchy. Fortunately, the efficient case is apt to be more common in commercial environments.

2.5.8 NDACReadReference : PolicyRepresentationObject

Instances of NDACReadReference are used by instances of ReadRule to describe classes of NDAC read situations. The NDACReadReference class encapsulates a UserSet and a LabelSet. The read situations described by each NDACReadReference instance are the cross-product of the memberships of these two sets.

2.5.9 NDACWriteReference : NDACReadReference

The NDACWriteReference class extends the NDACReadReference class to include an additional LabelSet. Instances of NDACWriteReference describe classes of NDAC write situations. The membership of the UserSet describes the Users who own the

writing process, the first LabelSet describes the Labels they may write, and the second LabelSet describes the set of Labels that the process has read previously.

2.5.10 NDACResult : PolicyRepresentationObject

The NDACResult class encapsulates a pair of MechanismSet instances (section 2.3.3). Instances of this class are used by ReadRule and WriteRule instances to specify the external mechanisms the SPDF must invoke in the course of evaluating the rule. The first MechanismSet specifies Mechanisms which the SPDF must invoke before it makes its decision. The second set specifies Mechanisms which the SPDF must invoke after it has made its decision.

2.5.11 ProcessReadSetMap : Map

An instance of ProcessReadSetMap associates a SystemProcess (section 2.2.5) with a LabelSet (section 2.2.2). This set contains all of the labels the process has read during its lifetime.

2.5.12 ProcessReadSetMapping : Mapping

The ProcessReadSetMapping maintains a dynamic list of ProcessReadSetMap objects. This class may be used by the SPDF to keep track of the labels read by every process in the system.

2.5.13 Design Issues

Two issues with the design of the framework's NDAC component deserve to be mentioned in some detail. The first concerns the rule-based representation used by the framework to express NDAC policies. The second deals with the distribution of essential functionality between the NDAC component of the framework and the DONDAC component which extends it. Both issues are addressed below.

2.5.13.1 Rules vs. Static Constraints The NDAC component of the framework describes policies in terms of rules governing the dynamic behavior of processes. This is a convenient form for SPDFs which must make decisions as the system runs. Unfortunately, NDAC policies are most naturally described in terms of static constraints, such as invariant conditions on the state of a finite-state-machine representation of a system. Although the two forms are essentially equivalent, the static constraint form may be more comfortable for policy developers, and may be more tractable for formal theorem proving.

This issue might be addressed in a complete policy representation environment such as the SMDE. [5] The CN high-level policy description language provided by the SMDE allows developers to describe NDAC policies in terms of static constraints. Meanwhile, the low-level translation produced for the SPDF presents the policy in terms of rules. Thus both the developers and the SPDF see the policy in a form they prefer. The representations constructed with this framework are meant to be SPDF-interpretable, so the framework uses a rule-based form.

2.5.13.2 Distribution of Domain-specific Functionality The NDAC component of the framework provides the functionality necessary to represent policy rules governing the reading and writing behavior of processes. The structure of these rules implicitly gives each process the same rights as the user who owns it. In some NDAC policy forms, it is desirable to associate an information domain with each process. Processes are then limited to activities in that domain only, rather than being free to access any domain to which their owner has access.

Representation for this form of NDAC policy is provided by the Domain-oriented NDAC (DONDAC) component of the framework (section 2.8). The DONDAC component is quite simple. It consists of only one class, and inherits nearly all of its NDAC functionality from the NDAC component of the framework. Unfortunately, encapsulating the domain-oriented functionality in a separate component greatly reduces the usefulness of the NDAC component. The generality of the framework could be significantly increased by integrating the DONDAC functionality into the NDAC component, and removing DONDAC component entirely.

2.6 Generic Device Specific Policy

This section describes the part of the framework devoted to representing the Device-specific Policy form (DSP). Unlike DAC and NDAC, DSP is not a form of access control. It does not describe how to decide whether or not a given operation should be allowed. Instead, it is used to represent policy rules which simply state that certain external mechanisms should be invoked when certain operations occur. The operations themselves are the same read and write used in the NDAC component of the framework (section 2.5). DSP also makes use of the same information domain structures employed by the NDAC policies.

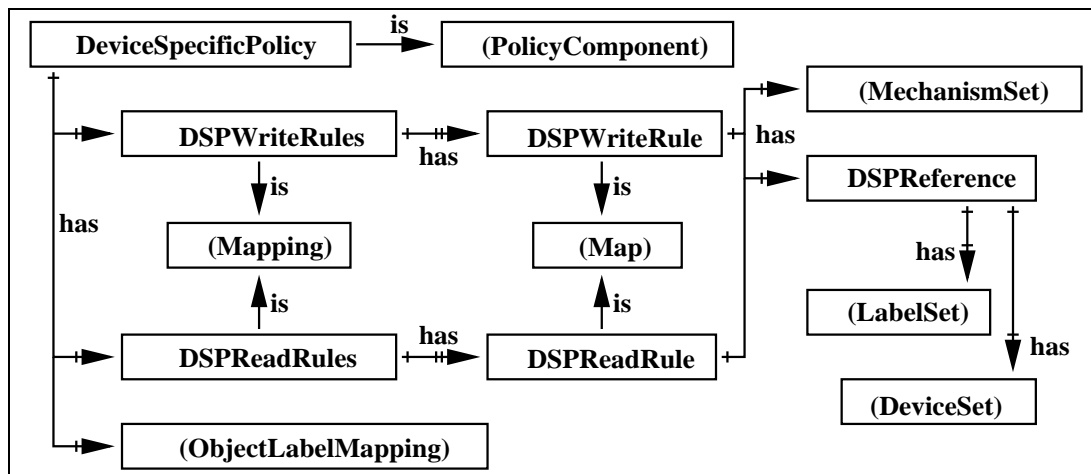


Figure 2.9: Generic Device Specific Policy Class Hierarchy Diagram

In addition, the DSP component of the framework is capable of expressing rules which deal with specific hardware devices. Although this seems unusual, it is necessary to enable the representation of policy statements such as “When information from the ‘foo’ domain is written to the public WAN, it must be encrypted using the ‘bar’ encryption package. However, when ‘foo’ domain information is written to any LAN, no encryption is necessary.” This ability to represent statements involving specific hardware devices gives the DSP policy form its name.

As can be seen from the diagram in figure 2.9, the structure of the DSP component of the framework is quite similar to the NDAC component. The DeviceSpecificPolicy class is at the top of the DSP hierarchy. Like the GenericNDACPolicy class (section 2.5.1), the DeviceSpecificPolicy class encapsulates mappings for rules governing read and write operations. These rules are represented by the DSPReadRules and DSPWriteRules classes, respectively. Each instance of the DeviceSpecificPolicy class also encapsulates an instance of the ObjectLabelMapping class (section 2.5.3) to maintain the mapping of Labels to SystemObjects.

The Evaluate method of the DeviceSpecificPolicy class resembles a simplified version of GenericNDACPolicy’s Evaluate method. The DSP version of evaluate takes a SystemObject, a Device, and an operation consisting of read or write as parameters. The DSP policy is not concerned with users or processes. When its Evaluate method is invoked, it is only concerned with the fact that a SystemObject with a certain Label is being read or written on a certain Device. Since it does not make any access

control decisions, the DSP Evaluate method it does not return a meaningful Boolean value.

As with NDAC's Evaluate, the evaluation algorithm consists of searching for rules matching the given situation. Unlike NDAC, if a relevant rule is not found, the SPDF does not move to prevent the action. The lack of a DSP rule for a given situation indicates only that no external mechanisms need to be invoked. If a rule is found, the set of Mechanisms it prescribes is passed to the SPDF for invocation. DSP rules specify only one MechanismSet each. Since DSP's Evaluate method makes no decisions, it is not necessary to specify one set of external mechanisms which must be invoked before deciding, and another which must be invoked after.

2.6.1 DeviceSpecificPolicy : PolicyComponent

The DeviceSpecificPolicy class encapsulates the entire generic Device Specific Policy concept, including the rules for reading and writing, and the instance ObjectLabelMapping (section 2.5.3).

2.6.2 DSPReference : PolicyRepresentationObject

Instances of the DSPReference class are used by instances of DSPReadRule and DSPWriteRule to describe the DSP-relevant situations to which they apply. In order to accomplish this, the DSPReference class encapsulates a LabelSet and a DeviceSet. The class of situations described by an instance of DSPReference is the cross product of the memberships of these two sets. That is, a given DSPReference describes every situation where a SystemObject with a Label in its LabelSet is read from (or written to) any Device in its DeviceSet.

2.6.3 DSPWriteRule : Map

An instance of the DSPWriteRule class represents a rule which describes a set of Mechanisms (section 2.3.3) that must be applied when writing data with certain Labels to certain Devices. An instance of DSPReference is used to describe the Labels and Devices.

2.6.4 DSPWriteRules : Mapping

An instance of the DSPWriteRules class encapsulates a list of DSPWriteRule instances, which together describe the entire DSP policy for writing. The purpose of the DSPWriteRules class is to describe a list of mechanisms which must be invoked

whenever particular kinds of write operations occur. The absence of a rule for a particular situation means only that no mechanisms need to be invoked, not that the situation should not be allowed.

2.6.5 DSPReadRule : Map

An instance of the DSPReadRule class represents a rule which describes a set of Mechanisms (section 2.3.3) that must be applied when reading data with certain labels from certain devices. An instance of DSPReference is used to describe the labels and devices.

2.6.6 DSPReadRules : Mapping

An instance of the DSPReadRules class encapsulates a list of DSPReadRule instances, which together describe the entire DSP policy for reading. The purpose of the DSPReadRules class is to describe a list of mechanisms which must be invoked whenever certain kinds of read operations occur. The absence of a rule for a particular situation means only that no mechanisms need to be invoked, not that the situation should not be allowed.

2.6.7 Design Issues

At first glance, allowing a security policy to make statements that refer to specific hardware devices may seem strange. However, a representation for hardware devices is necessary in order to express statements like the network policy example cited at the beginning of this section. Fortunately, support for this feature is not impossible to implement, since SPDFs which reside in operating system kernels will have access to knowledge about hardware devices.

It is likely that a high-level policy description language provided by a complete policy development environment would hide most of the device-specific details. Consider a policy for a distributed system containing many nodes (hosts, end systems). The nodes in the system might be connected to each other by several LANs, and the system as a whole might have a single connection to a public WAN through a router node (relay system). The high-level language representation of the policy for the entire system might state that information from the 'foo' domain sent to the WAN must be encrypted. This high-level statement does not specifically mention any hardware devices. However, if it were translated into a low-level representation based on the framework, it would resemble the network device example cited above

with its references to the LAN and WAN network adapter devices.

This example also serves to illustrate another point. Although the high-level policy statement described above applies to every node in the distributed system, the low-level rules that result from it only need to be given to the router node. Since it controls the only connection to the WAN, it is the only node that needs to be concerned with encryption. In some situations, every node in a given distributed system may deserve a different policy description which has been tailored for the node's individual role in the system-wide policy.

2.7 Device-aware Non-discretionary Access Control Policy

The DGSA describes a form of NDAC that is capable of handling what it calls Multidomain objects. [2] These are objects which appear to be composite objects made up of normal objects from different information domains. They are useful for situations in which users of a system must bring information from several domains together in order to accomplish some task. Even though Multidomain objects give the appearance of being the combination of objects from several different domains, the policy-enforcing system still maintains the separation between their component parts. This arrangement requires support for some interesting policy statements.

The Device-aware Non-discretionary Access Control Policy (DANDAC) component of the framework extends the NDAC component to support to Multidomain object policies. It extends several of the NDAC classes by adding an instance of Device to the list of parameters needed to describe a policy-relevant situation. DANDAC's version of the Evaluate method takes this extra parameter in addition to the ones required by NDAC's Evaluate. The classes used to represent DANDAC read and write rules are also extended versions of the NDAC rules, with the added ability to specify sets of Devices.

The most interesting policies that govern Multidomain objects are those that involve printing them or transferring them over networks. Both of these situations require the ability to specify hardware devices in NDAC policy rules. The DANDAC component of the framework might be used to combine rules governing Multidomain object printing with a normal NDAC policy. For example, it could represent a rule which prohibits a process which has read from more than one domain from writing, except when it writes to the printer device. The NDAC component by itself has no

way of representing the “except when it writes to the printer” clause.

2.7.1 DANDACPolicy : GenericNDACPolicy

The DANDACPolicy class extends the GenericNDACPolicy class (section 2.5.1) to handle NDAC rules and situations involving hardware devices. The majority of the NDAC functionality is provided by GenericNDACPolicy.

2.7.2 DANDACReadReference : NDACReadReference

The DANDACReadReference class extends the NDACReadReference class (section 2.5.8) to describe situations involving hardware devices. It accomplishes this by adding a DeviceSet to the sets already encapsulated by NDACReadReference.

2.7.3 DANDACWriteReference : NDACWriteReference

The DANDACWriteReference class extends the NDACWriteReference class (section 2.5.9) to describe situations involving hardware devices. It accomplishes this by adding a DeviceSet to the sets already encapsulated by NDACWriteReference.

2.7.4 DANDACReadRules : NDACReadRules

The DANDACReadRules class extends the NDACReadRules class (section 2.5.5) to handle rules which employ DANDACReadReferences instead of NDACReadReferences. The functionality provided by the two classes is otherwise identical.

2.7.5 DANDACWriteRules : NDACWriteRules

The DANDACWriteRules class extends the NDACWriteRules class (section 2.5.7) to handle rules which employ DANDACWriteReferences instead of NDACWriteReferences. The functionality provided by the two classes is otherwise identical.

2.8 Domain-oriented Non-discretionary Access Control

The Domain-oriented Non-discretionary Access Control (DONDAC) component of the framework contains only one class: DONDACPolicy. It extends the DANDAC component (section 2.7) to allow the representation of NDAC policies which associate each process with a domain, as described in section 2.5.13.2. The NDAC policies described by the DGSA can be represented with the DONDACPolicy class.

2.8.1 DONDACPolicy : DANDACPolicy

The DONDACPolicy class extends DANDACPolicy to allow the representation of DONDAC policy forms. The extension is quite simple. The DONDACPolicy's Evaluate method begins its decision-making process by determining the information domain (represented by a Label) with which the acting SystemProcess is associated. It then checks to see if the SystemObject being read or written is associated with the same domain. If it is, the evaluation process continues according to the standard DANDAC algorithm. If the SystemProcess and SystemObject are not associated with the same information domain, the operation is not allowed.

SystemProcesses are associated with Information domains (represented by instances of the Label class) using the same ObjectLabelMap instance used for SystemObjects. Since SystemProcesses are a kind of SystemObject, this requires no additional specialization in the DONDAC component of the framework.

The DONDACPolicy class also supports the concept of a special set of 'trusted' information domains. This facility allows the specification of policies which have special domains set aside for operating system kernel or trusted server information. SystemProcesses associated with a 'trusted' information domain are not bound by the usual DONDAC restrictions. Like SystemProcesses in the NDAC policy, they are allowed to access any information domain that their associated User may access. This feature is useful for handling kernel processes and server processes which the kernel trusts not to violate the spirit of the security policy.

2.8.2 DONDACPolicy : DANDACPolicy

The DONDACPolicy class extends the DANDACPolicy class to support the representation of NDAC policy forms which associate processes with information domains. The vast majority of the functionality of this class is inherited from DANDACPolicy.

2.8.3 Design Issues

As explained in section 2.5.13.2, the experience gained from building the prototype of the framework shows that the functionality provided by the DONDAC component of the framework should be integrated into the NDAC component. This would provide DONDAC functionality to both the NDAC and DANDAC components, and significantly increase the usefulness of the framework.

2.9 Support for Security Policy Decision Functions

The prototype implementation of the framework is provided as a Java package. This package is made up of sub-packages, with one sub-package for each of the framework components described above. An SPDF can import whichever sub-packages are necessary to represent the policies it needs to interpret.

Since a simple SPDF simulator was required to test policies represented with the framework, the framework package is accompanied by a smaller package of useful SPDF-related classes. This SPDF package contains a variety of classes for keeping track of the state of the system. These classes may be used to encapsulate instances of `DACAttributes` (section 2.4.9), `ObjectLabelMapping` (section 2.5.3) and `ProcessReadSetMapping` (section 2.5.12). This arrangement allows all of the policy instances contained in a SPDF to share this state information.

Chapter 3

Using the Framework

This chapter describes how the framework might be used in real-world systems. First, it provides two examples of security policies represented with the prototype framework. These examples are based in corporate settings, and demonstrate both the utility of the framework and the commercial applicability of their policy forms. The two examples are followed by a short summary of the areas other than operating system kernels where a policy representation mechanism like the framework might be useful.

3.1 The PrarieSoft DSP Scenario

This scenario describes the information-system-relevant aspects of the PrarieSoft corporate information security policy. The PrarieSoft corporation is a fictitious software development company based in the adjacent cities of Urbana and Champaign, IL. PrarieSoft's developers are distributed between two separate sites, one located in each city. At each site, the developer's workstations are linked by private LANs. The two sites communicate with each other using a public WAN. This arrangement is pictured in figure 3.1

All of PrarieSoft's workstations are nodes (end systems) in a single distributed system. File servers at one site routinely serve nodes at the other. Thus, the volume of data flowing over the public WAN between the two sites is quite high. To complicate matters, due to the modern design of PrarieSoft's distributed file system, it is quite difficult for the average user to determine which of his actions result in file transfers over the network and which do not.

If PrarieSoft chose to operate without taking any security precautions, the risk of a competing corporation obtaining access to valuable PrarieSoft data as it traversed the WAN would be quite high. Fortunately, PrarieSoft's distributed system enforces a security policy which mandates cryptographic protection for any sensitive data

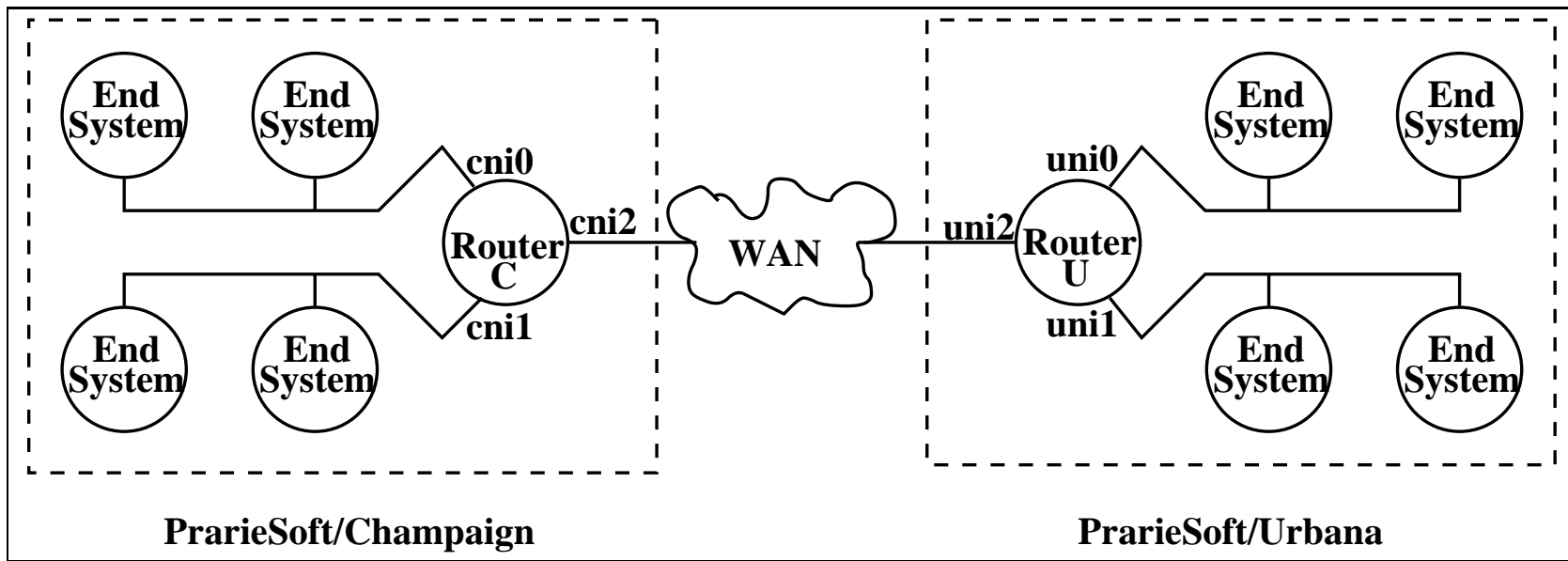


Figure 3.1: The PrarieSoft Network Topology Showing the Names of Significant Network Interface Devices

placed on the WAN. This policy is described below.

The PrarieSoft security policy is represented with the DSP component of the framework. The operating systems on each of the nodes in PrarieSoft's distributed system automatically enforce this policy. This allows the developers to go about their normal tasks without having to worry about security issues.

PrarieSoft divides its data into three information domains. The first is the 'PrarieSoft' domain, which contains all of the administrative information required for the day-to-day operation of the company. The second is the 'CornCobProprietary' domain. This contains all of the proprietary information concerning PrarieSoft's latest development project, codenamed 'CornCob'. The third information domain is the 'CornCobShared' domain. This contains information concerning the CornCob project that has been released to contractors outside the corporation, or to the general public. The information in the PrarieSoft and CornCobProprietary domains is considered sensitive and its secrecy and integrity is valuable to PrarieSoft.

The PrarieSoft DSP policy is summarized by the following high-level rules:

1. All information from the PrarieSoft and CornCobProprietary domains passing over the public WAN must be encrypted using the 'foo' encryption package.
2. All information from the CornCobShared domain passing over the public WAN must be digitally signed using the 'foo' encryption package to ensure its integrity and authenticate the identity of its originator.
3. No encryption is required when passing information over one of the PrarieSoft LANs, since these LANs are physically secure.

Table 3.1 summarizes the DSP read and write rules for the two routers shown in figure 3.1. The other nodes in the PrarieSoft distributed system do not need to be concerned with the DSP policy, since they may only access the various LANs. Appendix A presents a parse-tree-like description of the hierarchy of objects used to represent the Urbana router's version of this policy with the prototype framework.

DSP Rules for Reading:		
LabelSet	DeviceSet	Mechanism
For Router C:		
{ YokelSoft, CornCobProprietary }	{ cni2 }	Decrypt
{ CornCobShared }	{ cni2 }	Verify Signature, Integrity
For Router U:		
{ YokelSoft, CornCobProprietary }	{ uni2 }	Decrypt
{ CornCobShared }	{ uni2 }	Verify Signature, Integrity
DSP Rules for Writing:		
LabelSet	DeviceSet	Mechanism
For Router C:		
{ YokelSoft, CornCobProprietary }	{ cni2 }	Encrypt
{ CornCobShared }	{ cni2 }	Sign
For Router U:		
{ YokelSoft, CornCobProprietary }	{ uni2 }	Encrypt
{ CornCobShared }	{ uni2 }	Sign

Table 3.1: DSP rules for PrarieSoft's Routers

Employee	Employer	Domains
Bates	Boering	SOT, <i>ZZZ</i>
Benson	Boering	SOT, <i>ZZZ</i>
Davis	Duckweed-Martian	SOT,DUM
Devlin	Duckweed-Martian	SOT,DUM

Table 3.2: Employees of SOT, and the domains they may access.

3.2 The Aerospace Collaboration DONDAC Scenario

This section describes a scenario involving two fictitious aerospace giants: Duckweed Martian and Boering. These two corporations are usually competitors, but in this case, they are collaborating on a contract to build a new Sub-Orbital Transport. The two have formed a third corporation to support this joint venture, called SOT.

Half of SOT's employees are on loan from Duckweed Martian, and half are from Boering. Both corporations are allowing SOT to make use of a limited amount of their proprietary corporate data in their research. Thus, a given workstation at the SOT site may contain proprietary data from both Duckweed Martian and Boering, as well as shared SOT data.

This raises two related issues: First, how can the use of proprietary data of the parent corporations be controlled, and second, how can the proprietary data of one parent corporation be protected from the employees of the other while it is stored at the SOT site? A security policy based on the framework's DONDAC component is capable of addressing both of these concerns.

3.2.1 DONDAC Policy Information Domains

In order to keep the parent companies' data separated, the information on SOT's computer systems is divided into three information domains, one for each corporation. There is a DUM domain for Duckweed Martian data, a *ZZZ* domain for Boering data, and an SOT domain for SOT data. Each SOT employee is granted access to two of these domains, depending on their parent corporation of origin. (Access in this case refers to the ability to both read and write data of the domain in question.) This example describes a representative sample of four SOT employees, named Bates, Benson, Davis and Devlin. The parent company which employs each of them and the domains to which they have access are shown in table 3.2.

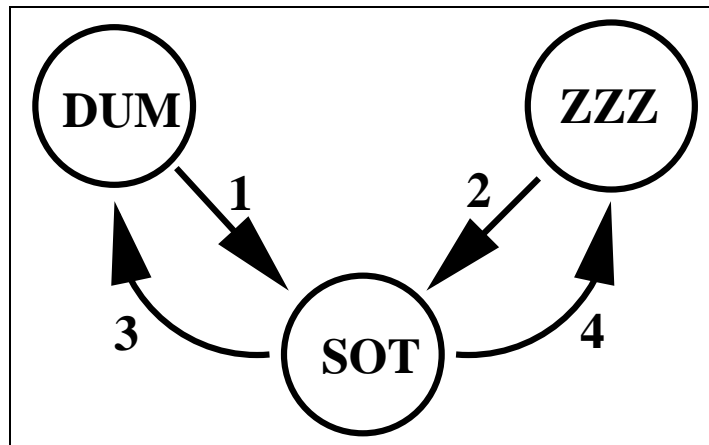


Figure 3.2: The Flow of Information between the DUM, SOT, and ZZZ Information Domains

In addition to the three corporate information domains described above, the policy includes a fourth domain called ‘System’. This is the domain used by the operating system kernels on each node in SOT’s distributed system, as well as a variety of trusted servers. Some of these servers provide support for Multidomain Objects. The operating system kernels provide the ability to transfer information from one domain to another as a system call. The use of this system call is closely controlled by the system’s security policy.

3.2.2 Interdomain Information Transfer Policy

Bates and Davis have a special role to play in the security policy. They are the liaison officers for their respective parent companies. The policy makes them responsible for controlling the flow of information between domains, as described below.

SOT’s security policy defines a strict set of rules which governs how information can move from one domain to another. The diagram in figure 3.2 shows the three corporate information domains. The arrows represent the ways in which information may move from one domain to another. The numbers on the arrows correspond to the policy statements listed below:

1. Only the Duckweed Martian liaison officer may transfer information from the DUM domain to the SOT domain, and only if that specific data has been ‘released’ by the Duckweed Martian management. The system uses a software application to keep track of which items have and have not been released, so

both the process of releasing information and checking the status of a particular piece of material is fully automated. Transfers can't occur over the network.

2. Only the Boering liaison officer may transfer information from the *ZZZ* domain to the SOT domain, and only if that specific data has been 'released' by the Boering management. The system uses a software application to keep track of which items have and have not been released, so both the process of releasing information and checking the status of a particular piece of material is fully automated. Transfers can't occur over the network.
3. The Duckweed Martian liaison officer may transfer any SOT information into the DUM domain, but the action must be logged so that each parent corporation can keep track of what SOT information the other is taking. Transfers can't occur over the network.
4. The Boering liaison may transfer any SOT information into the DUM domain, but the action must be logged so that each parent corporation can keep track of what SOT information the other is taking. Transfers can't occur over the network.

3.2.3 DSP-like Policy Rules and Multidomain Objects

Like the PrarieSoft corporation described in section 3.1, the SOT corporation must also deal with the problem of passing its sensitive information over public networks. As with the PrarieSoft sites, the router (relay system) bears the majority of the responsibility for enforcing network-related policies. Figure 3.3 contains a diagram describing the physical layout of SOT's distributed system, paying particular attention to the hardware devices connected to the SOT corporate router. The following list summarizes the DSP rules which apply to this router.

1. When DUM or *ZZZ* data is written to the local file system, it must be encrypted. When it is read, it must be decrypted. This is intended to thwart *ZZZ* employees who might be tempted to remove physical storage media containing DUM data from the site, as well as DUM employees who might try the same trick with *ZZZ* data. SOT data is shared, and does not need to be encrypted in this case.
2. When SOT, DUM, or *ZZZ* data is written to the WAN, it must be encrypted to protect it on its journey through the public networks. When SOT, DUM,

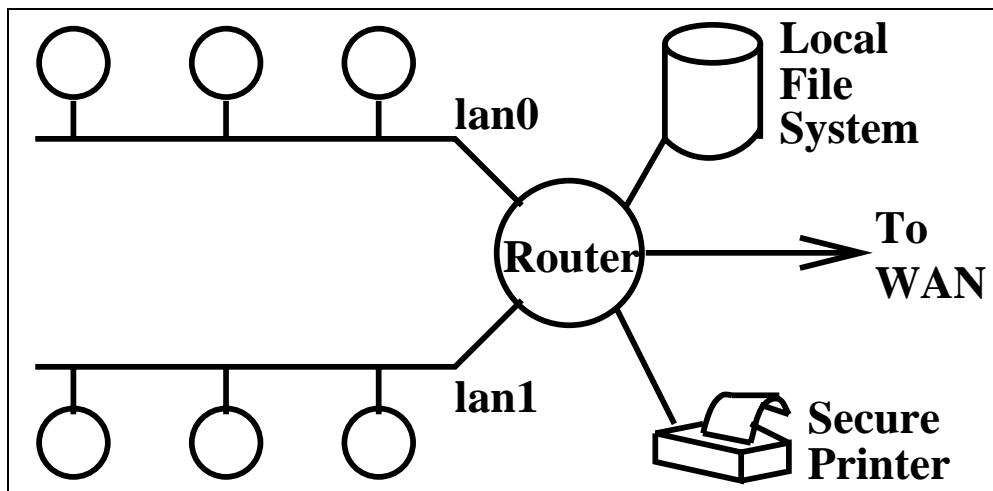


Figure 3.3: Components of the SOT Distributed System

or *ZZZ* data is read from the WAN, it must be decrypted. It is not necessary to encrypt data written to either of the SOT LANs, since they exist inside of the physically secure SOT site.

3. Users may employ the system's trusted Multidomain object server to construct reports made up of information from more than one domain. These reports may only be written to the printer in the secure printing room, and only when the operator is on duty.

Appendix B shows a parse-tree-like description of the hierarchy of objects used to represent this policy with the prototype framework.

3.2.4 SOT Security Policy Representation and the SPDF Simulator

Figures 3.4 and 3.5 present images of the SPDF simulator exercising the SOT policy on a Sun SPARCstation running Solaris 5.4. The simulator allows a user to interactively describe a series of situations requiring DONDAC policy decisions. The simulator then evaluates the policy for the given situation and presents the user with the result. Whenever the policy requires the invocation of an external mechanism, the simulator informs the user and asks what the result of the invocation should be. As the simulation proceeds, the user plays the role of all the various external mechanisms.

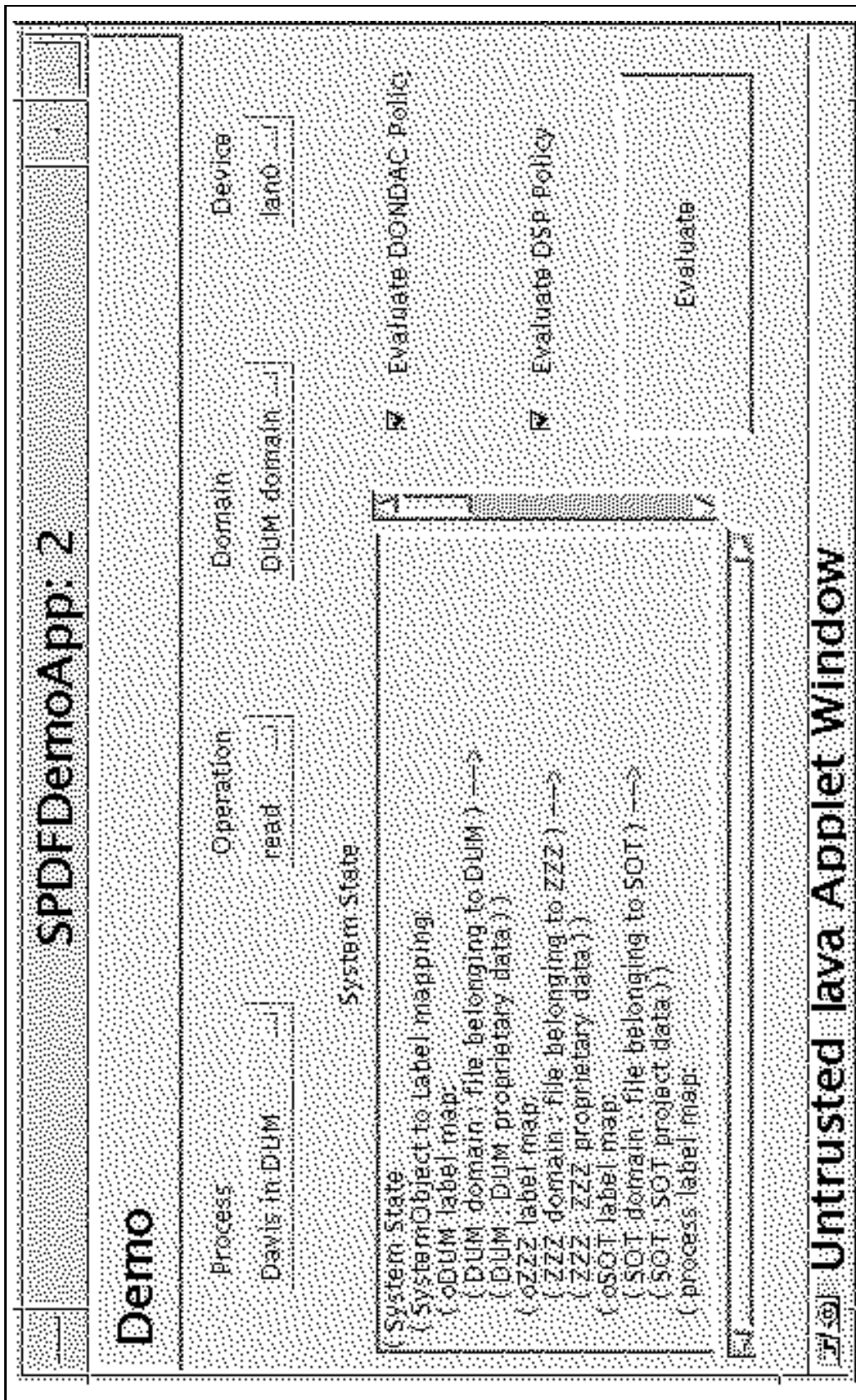


Figure 3.4: The SPDF Simulator Waiting for Input

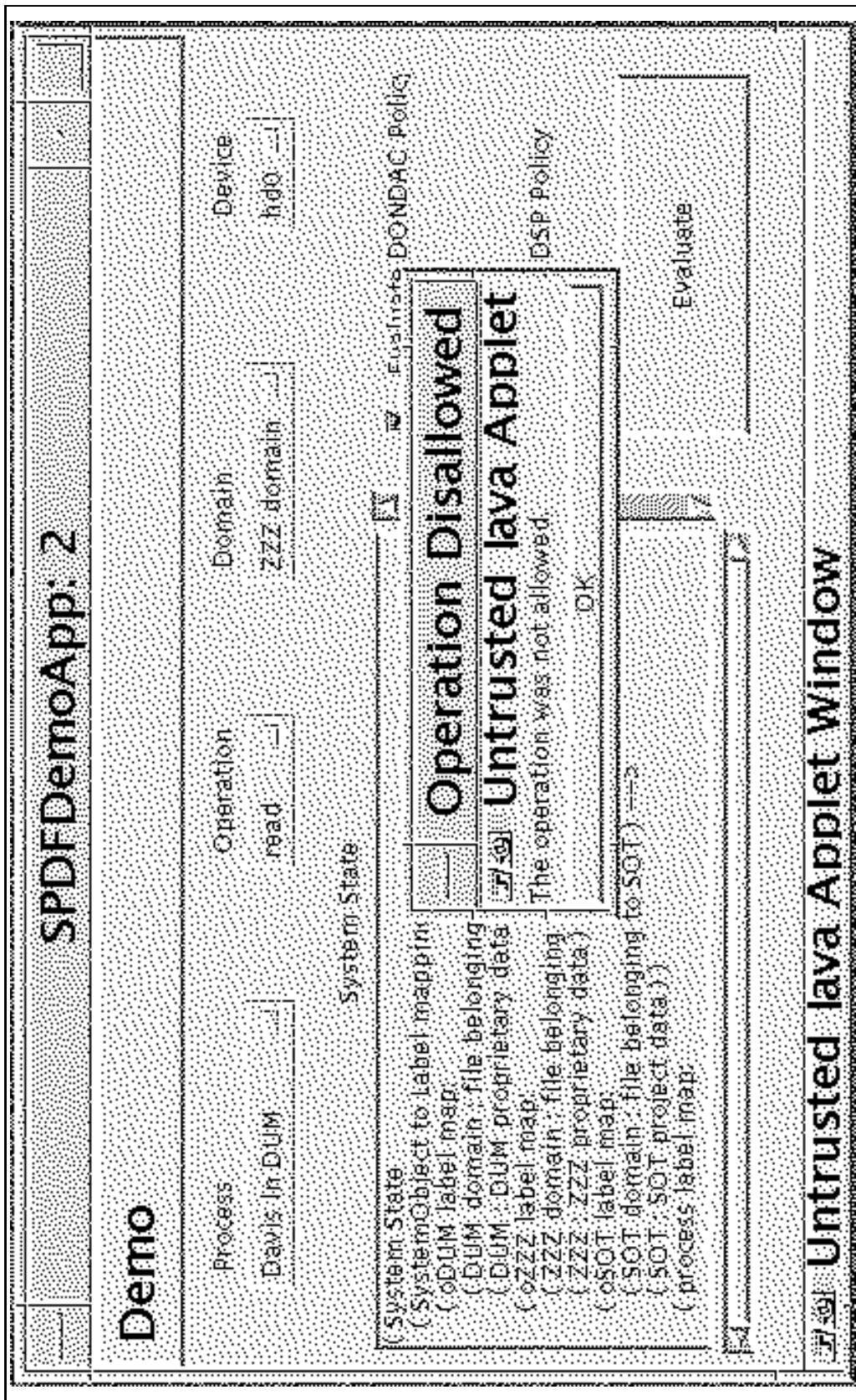


Figure 3.5: SPDF Simulator Indicating that Davis is not Allowed to Read ZZZ information.

3.3 Alternate Uses for the Framework

A security policy representation framework might be useful in a variety of contexts other than an operating system's SPDF. Any server that must enforce some sort of security policy could be designed to incorporate an interpreting SPDF and make use of the framework. Framework-based policy representations could be used to communicate policy information between such servers. A good example of this application might be a cryptographic key server. Users might submit keys to this server for distribution. They might accompany the keys with DAC policy descriptions based on the framework. These descriptions might represent the policies governing the key's release to the server's various clients.

Framework-based policy representations might also be used to describe local system resources available to various types of scripts or applets received over a network and executed in a limited environment. Or, Framework-based descriptions of policy might be sent between nodes in a distributed system in order to implement remote policy management and reconfiguration. In general, any situation requiring an interpretable representation of security policy could benefit from the framework's functionality.

Chapter 4

Analysis

The prototype implementation of the framework demonstrates that the general policy-representing framework concept is viable. As described in section 1.4, the safety and expressive qualities of the framework approach are equal to those of a specialized interpreted language designed especially for policy representation. The framework approach may be considered superior in the sense that framework implementations take less time to develop. With the addition of a complete policy development environment, the framework could form the basis of policy descriptions interpreted by SPDFs in a variety of policy enforcing applications, from operating system kernels to cryptographic key servers. The framework also demonstrates how mechanisms external to the SPDF may be integrated into the decision-making process, and how NDAC policies that are aware of individual hardware devices may support rules governing Multidomain objects.

On the other hand, there are some practical limits to the usefulness of the framework. Although the framework-based form of representation is easily interpreted by SPDFs, it is not particularly convenient for formal analysis. Automated theorem provers might find the rule-oriented representation of NDAC policies particularly hard to digest. Similarly, the framework-based representation does not facilitate practical analysis of policies. For example, there is no easy way to determine if a given NDAC or DSP rule contradicts another, or makes another redundant. Furthermore, there is no way to take two policies represented with the framework and determine in a meaningful way what they have in common. This ability would simplify the process of developing new policies based on a combination of old ones. It would also be helpful when attempting to determine if one policy subsumes another.

Still, as a general and extensible policy representation mechanism, the framework approach is both viable and practical. It's basis in an existing interpreted general programming language simplifies the development process considerably, making it the ideal tool for prototypes and experimental systems. The framework provides exactly the general policy representation mechanism needed to make interpreting SPDFs viable. In turn, this SPDF technology enables the production of operating systems capable of enforcing a wide range of commercially useful security policies. Although the initial prototype framework implementation is far from optimal, it shows that future research in this direction could provide the general policy representation mechanism required to eventually bring trusted operating systems into the commercial mainstream.

Appendix A

Framework Representation State for PrarieSoft DSP

Section 3.1 describes a security policy for a fictitious corporation called ‘PrarieSoft’. This was one of the test cases represented with the prototype framework. The policy’s representation is actually a short Java program that instantiates a number of the classes provided by the prototype framework’s DSP component. Together, these instances represent the policy and provide methods to help evaluate decisions based on it.

Each class in the framework implements a method called ‘AddDescription’ which may be used to produce a short text description of a given instance of that class. The description associated with each instance must be provided as a string parameter to the constructor used to create the instance. In classes which encapsulate instances of other classes, the AddDescription method also invokes the AddDescription methods of the encapsulated objects. Thus, it is possible to invoke the AddDescription method of a top-level object like DeviceSpecificPolicy (2.6.1) to produce a text description of all of the objects it encapsulates. If a programmer (or automated translator) is careful enough to provide each object in a policy description with a meaningful text description, this method may be used to produce a structured text translation of the policy.

As described in section 2.6, the DeviceSpecificPolicy object encapsulates several other objects, each of which encapsulates several others, and so on. A diagram of these encapsulation relationships might resemble a tree with the DeviceSpecificPolicy object at the root. The invocation of this object’s AddDescription method triggers an in-order traversal of all the objects in the tree. Each object adds its text description to the overall representation’s description as it is passed in the traversal.

The actual structured text description of the PrarieSoft policy is provided below. Indentation and parenthesis are used to describe its tree-like structure, following the

same practice traditionally used to format LISP or Scheme programs. This practice places the root node of the tree on the first line with no indentation. A child of a given node occurs on a line below its parent, and is indented one space more than the parent node. Nodes at the same level in the tree are indented equally.

The first line of the text description was produced by the root DeviceSpecificPolicy object. It describes itself with the text "PrarieSoft DSP:". According to the indentation, it encapsulates two mapping objects, one of class DSPReadRules and one of class DSPWriteRules. These objects describe themselves with the strings "Rules for reading:" and "Rules for writing:", respectively. The objects encapsulated by these two mappings are listed below them in a similar fashion. The overall structured text description contains a considerable amount of redundant information, since each object that is referenced more than once in the tree adds its entire description each time.

```
( PrarieSoft DSP:
  ( Rules for reading:
    ( Rule for reading sensitive data:
      ( sensitive read situations:
        ( Sensitive PrarieSoft Labels:
          ( PrarieSoft : PrarieSoft Corporation Proprietary Data )
          ( CornCobProprietary : PrarieSoft Proprietary CornCob Data ) )
        ( Public Nets:
          ( uni2 : Connection to Public WAN ) ) ) -->
      ( decryption mechanisms:
        ( decrypt : Decryption Mechanism ) ) )
    ( Rule for reading non-sensitive data:
      ( non-sensitive read situations:
        ( Non-sensitive PrarieSoft Labels:
          ( CornCobShared : Public CornCob Project Data ) )
        ( Public Nets:
          ( uni2 : Connection to Public WAN ) ) ) -->
      ( signature verification mechanisms:
        ( verify : Digital Signature Verification Mechanism ) ) ) ) )
```

```

( Rules for writing:
  ( Rule for writing sensitive data:
    ( sensitive write situations:
      ( Sensitive PrarieSoft Labels:
        ( PrarieSoft : PrarieSoft Corporation Proprietary Data )
        ( CornCobProprietary : PrarieSoft Proprietary CornCob Data ) )
      ( Public Nets:
        ( uni2 : Connection to Public WAN ) ) ) -->
    ( encryption mechanisms:
      ( encrypt : Encryption Mechanism ) ) )
  ( Rule for writing non-sensitive data:
    ( non-sensitive write situations:
      ( Non-sensitive PrarieSoft Labels:
        ( CornCobShared : Public CornCob Project Data ) )
      ( Public Nets:
        ( uni2 : Connection to Public WAN ) ) ) -->
    ( signing mechanisms:
      ( sign : Digital Signature Creation Mechanism ) ) ) ) )

```

Appendix B

Framework Representation State for Aerospace Collaboration DONDAC policy

This appendix presents the actual structured text description of the policy representation produced for the SOT example of section 3.2. It is formatted in the same fashion as the structured text description provided in appendix A.

```
( System Policy:
  ( Demo DONDAC Policy:
    ( System Domains:
      ( System : System Domain ) )
    ( Rules for reading:
      ( Rule for reading DUM:
        ( reading DUM data:
          ( DUM employess on SOT project:
            ( Davis : DUM employee on SOT project, DUM liason )
            ( Devlin : DUM employee on SOT project ) )
          ( DUM labels:
            ( DUM : DUM proprietary data ) )
          ( all devices:
            ( lan0 : LAN 0 )
            ( lan1 : LAN 1 )
            ( wan : WAN )
            ( hd0 : local fixed disk )
            ( lpr : printer in secure printing room ) ) ) -->
        ( no mechanisms:
          ( no mechanisms: )
          ( no mechanisms: ) ) ) )
```

```

( Rule for reading ZZZ:
  ( reading ZZZ data:
    ( ZZZ employess on SOT project:
      ( Bates : ZZZ employee on SOT project, ZZZ liason )
      ( Benson : ZZZ employee on SOT project ) )
    ( ZZZ labels:
      ( ZZZ : ZZZ proprietary data ) )
    ( all devices:
      ( lan0 : LAN 0 )
      ( lan1 : LAN 1 )
      ( wan : WAN )
      ( hd0 : local fixed disk )
      ( lpr : printer in secure printing room ) ) ) -->
  ( no mechanisms:
    ( no mechanisms: )
    ( no mechanisms: ) ) )
( Rule for reading SOT:
  ( reading SOT data:
    ( all employees on SOT project:
      ( Davis : DUM employee on SOT project, DUM liason )
      ( Devlin : DUM employee on SOT project )
      ( Bates : ZZZ employee on SOT project, ZZZ liason )
      ( Benson : ZZZ employee on SOT project ) )
    ( SOT labels:
      ( SOT : SOT project data ) )
    ( all devices:
      ( lan0 : LAN 0 )
      ( lan1 : LAN 1 )
      ( wan : WAN )
      ( hd0 : local fixed disk )
      ( lpr : printer in secure printing room ) ) ) -->
  ( no mechanisms:
    ( no mechanisms: )

```

```

    ( no mechanisms: ) ) ) )
( Rules for writing:
  ( Rule for writing DUM data:
    ( writing DUM data:
      ( DUM employess on SOT project:
        ( Davis : DUM employee on SOT project, DUM liason )
        ( Devlin : DUM employee on SOT project ) )
      ( DUM labels:
        ( DUM : DUM proprietary data ) )
      ( DUM labels:
        ( DUM : DUM proprietary data ) )
      ( all devices:
        ( lan0 : LAN 0 )
        ( lan1 : LAN 1 )
        ( wan : WAN )
        ( hd0 : local fixed disk )
        ( lpr : printer in secure printing room ) ) ) -->
    ( no mechanisms:
      ( no mechanisms: )
      ( no mechanisms: ) ) )
  ( Rule for writing ZZZ data:
    ( writing ZZZ data:
      ( ZZZ employess on SOT project:
        ( Bates : ZZZ employee on SOT project, ZZZ liason )
        ( Benson : ZZZ employee on SOT project ) )
      ( ZZZ labels:
        ( ZZZ : ZZZ proprietary data ) )
      ( ZZZ labels:
        ( ZZZ : ZZZ proprietary data ) )
      ( all devices:
        ( lan0 : LAN 0 )
        ( lan1 : LAN 1 )
        ( wan : WAN )

```

```

        ( hd0 : local fixed disk )
        ( lpr : printer in secure printing room ) ) ) -->
( no mechanisms:
  ( no mechanisms: )
  ( no mechanisms: ) ) )
( Rule for writing SOT data:
  ( writing SOT data:
    ( all employees on SOT project:
      ( Davis : DUM employee on SOT project, DUM liason )
      ( Devlin : DUM employee on SOT project )
      ( Bates : ZZZ employee on SOT project, ZZZ liason )
      ( Benson : ZZZ employee on SOT project ) )
    ( SOT labels:
      ( SOT : SOT project data ) )
    ( SOT labels:
      ( SOT : SOT project data ) )
    ( all devices:
      ( lan0 : LAN 0 )
      ( lan1 : LAN 1 )
      ( wan : WAN )
      ( hd0 : local fixed disk )
      ( lpr : printer in secure printing room ) ) ) ) -->
  ( no mechanisms:
    ( no mechanisms: )
    ( no mechanisms: ) ) )
( Rule for printing Multidomain object with DUM and SOT labels:
  ( printing Multidomain object with DUM and SOT labels:
    ( DUM employess on SOT project:
      ( Davis : DUM employee on SOT project, DUM liason )
      ( Devlin : DUM employee on SOT project ) )
    ( DUM and SOT labels:
      ( DUM : DUM proprietary data )
      ( SOT : SOT project data ) ) )

```

```

( DUM and SOT labels:
  ( DUM : DUM proprietary data )
  ( SOT : SOT project data ) )
( secure printing room printers:
  ( lpr : printer in secure printing room ) ) -->
( mechanisms for Multidomain object printing:
  ( external predicates for printing Multidomain objects:
    ( isSpoOnDuty : secure printer operator on duty predicate ) )
  ( policy-mandated mechanisms for printing Multidomain objects:
    ( coverSheetGenerator : labelled cover sheet generator ) ) ) )
( Rule for printing Multidomain object with ZZZ and SOT labels:
  ( printing Multidomain object with ZZZ and SOT labels:
    ( ZZZ employess on SOT project:
      ( Bates : ZZZ employee on SOT project, ZZZ liason )
      ( Benson : ZZZ employee on SOT project ) )
    ( ZZZ and SOT labels:
      ( ZZZ : ZZZ proprietary data )
      ( SOT : SOT project data ) )
    ( ZZZ and SOT labels:
      ( ZZZ : ZZZ proprietary data )
      ( SOT : SOT project data ) )
    ( secure printing room printers:
      ( lpr : printer in secure printing room ) ) ) -->
  ( mechanisms for Multidomain object printing:
    ( external predicates for printing Multidomain objects:
      ( isSpoOnDuty : secure printer operator on duty predicate ) )
    ( policy-mandated mechanisms for printing Multidomain objects:
      ( coverSheetGenerator : labelled cover sheet generator ) ) ) )
( Rule for transferring DUM data to SOT:
  ( transferring DUM data to SOT:
    ( DUM liason officers:
      ( Davis : DUM employee on SOT project, DUM liason ) )
    ( DUM labels:

```

```

    ( DUM : DUM proprietary data ) )
  ( SOT labels:
    ( SOT : SOT project data ) )
  ( storage devices for local file system:
    ( hd0 : local fixed disk ) ) ) -->
( mechanisms for DUM to SOT transfer:
  ( DUM release-checking mechanisms:
    ( releasedByDUM : released by DUM liason predicate ) )
  ( no mechanisms: ) ) )
( Rule for transferring ZZZ data to SOT:
  ( transferring ZZZ data to SOT:
    ( ZZZ liason officers:
      ( Bates : ZZZ employee on SOT project, ZZZ liason ) )
    ( ZZZ labels:
      ( ZZZ : ZZZ proprietary data ) )
    ( SOT labels:
      ( SOT : SOT project data ) )
    ( storage devices for local file system:
      ( hd0 : local fixed disk ) ) ) ) -->
  ( mechanisms for ZZZ to SOT transfer:
    ( ZZZ release-checking mechanisms:
      ( releasedByZZZ : released by ZZZ liason predicate ) )
    ( no mechanisms: ) ) )
( Rule for transferring SOT data to DUM:
  ( transferring SOT data to DUM:
    ( DUM liason officers:
      ( Davis : DUM employee on SOT project, DUM liason ) )
    ( SOT labels:
      ( SOT : SOT project data ) )
    ( DUM labels:
      ( DUM : DUM proprietary data ) )
    ( storage devices for local file system:
      ( hd0 : local fixed disk ) ) ) ) -->

```

```

( mechanisms for SOT to DUM transfer:
  ( no mechanisms: )
  ( Information Transfer Logging mechanisms:
    ( TranferLogger : logs information transfers ) ) )
( Rule for transferring SOT data to ZZZ:
  ( transferring SOT data to ZZZ:
    ( ZZZ liason officers:
      ( Bates : ZZZ employee on SOT project, ZZZ liason ) )
    ( SOT labels:
      ( SOT : SOT project data ) )
    ( ZZZ labels:
      ( ZZZ : ZZZ proprietary data ) )
    ( storage devices for local file system:
      ( hd0 : local fixed disk ) ) ) -->
  ( mechanisms for SOT to ZZZ transfer:
    ( no mechanisms: )
    ( Information Transfer Logging mechanisms:
      ( TranferLogger : logs information transfers ) ) ) ) )
( Demo DSP Policy:
  ( Rules for reading:
    ( Rule for reading DUM, ZZZ or SOT from WAN:
      ( reading DUM, ZZZ or SOT from WAN:
        ( DUM, ZZZ, and SOT labels:
          ( DUM : DUM proprietary data )
          ( ZZZ : ZZZ proprietary data )
          ( SOT : SOT project data ) )
        ( adaptors for WANS:
          ( wan : WAN ) ) ) -->
      ( decryption mechanisms:
        ( decrypt : Decryption Mechanism ) ) )
    ( Rule for reading DUM or ZZZ from LFS:
      ( reading DUM or ZZZ from LFS:
        ( DUM and ZZZ labels:

```

```

        ( DUM : DUM proprietary data )
        ( ZZZ : ZZZ proprietary data ) )
    ( storage devices for local file system:
      ( hd0 : local fixed disk ) ) ) -->
  ( decryption mechanisms:
    ( decrypt : Decryption Mechanism ) ) ) )
( Rules for writing:
  ( Rule for writing DUM, ZZZ or SOT to WAN:
    ( writing DUM, ZZZ or SOT to WAN:
      ( DUM, ZZZ, and SOT labels:
        ( DUM : DUM proprietary data )
        ( ZZZ : ZZZ proprietary data )
        ( SOT : SOT project data ) )
      ( adaptors for WANS:
        ( wan : WAN ) ) ) ) -->
    ( encryption mechanisms:
      ( encrypt : Encryption Mechanism ) ) )
  ( Rule for writing DUM or ZZZ to LFS:
    ( writing DUM or ZZZ to LFS:
      ( DUM and ZZZ labels:
        ( DUM : DUM proprietary data )
        ( ZZZ : ZZZ proprietary data ) )
      ( storage devices for local file system:
        ( hd0 : local fixed disk ) ) ) ) -->
    ( encryption mechanisms:
      ( encrypt : Encryption Mechanism ) ) ) ) ) ) )

```

Bibliography

- [1] Abrams, et. al. *Information Security: An Integrated Collection of Essays*. IEEE Computer Society Press, 1995. ISBN 0-8186-3662-9.
- [2] Center for Information System Security. *Department of Defense (DoD) Goal Security Architecture (DGSA) (version 3.0 draft)*. September 1995.
- [3] Department of Defense. *Department of Defense Trusted Computer System Evaluation Criteria*. DoD 5200.28-STD, December 1995.
- [4] Grego, Adolfo. *Network Security: Current Mechanisms and Future Trends*. Masters Thesis, University of Illinois, 1996.
- [5] Heaney, et. al. *An Environment for Security Model Development*. IEEE 5th Annual Computer Security Applications Conference, December 1989.
- [6] Reichel, Rob. *Inside Windows NT Security, Part 1*. Windows/DOS Developer's Journal, May 1993.