NIST Interagency Report xxxx

National Institute of Standards and Technology
U.S. Department of Commerce

Policy Machine: Features, Architecture, and Specification

David Ferraiolo
Serban Gavrila
Wayne Jansen

NIST IR-xxxx

Policy Machine: Features, Architecture, and Specification

David Ferraiolo
Serban Gavrila
Wayne Jansen

C O M P U T E R   S E C U R I T Y

Computer Security Division
Information Technology Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899-8930

December 2012

Reports on Computer Systems Technology

National Institute of Standards and Technology Interagency Report xxxx
105 pages (December 2012)

Abstract

The ability to control access to sensitive data in accordance with policy is perhaps the most fundamental security requirement. Despite over four decades of security research, the limited ability for existing access control mechanisms to enforce a comprehensive range of policy persists. While researchers, practitioners and policy makers have specified a large variety of access control policies to address real-world security issues, only a relatively small subset of these policies can be enforced through off-the-shelf technology, and even a smaller subset can be enforced by any one mechanism. This report describes an access control framework, referred to as the Policy Machine (PM), which fundamentally changes the way policy is expressed and enforced. The report gives an overview of the PM and the range of policies that can be specified and enacted. The report also describes the architecture of the PM and the properties of the PM model in detail.

Keywords: *Access Control; Authorization; Privilege Management; Computer Security*

Acknowledgements

The authors, David Ferraiolo and Serban Gavrila of the National Institute of Standards and Technology, and Wayne Jansen of Booz Allen Hamilton, wish to thank colleagues at NIST and elsewhere who reviewed drafts of this document and contributed to its technical content. Key improvements to this document would not have been possible without the feedback and valuable suggestions of those individuals.

Table of Contents

1. Introduction

Access control as it pertains to a computing environment is the ability to allow or prevent an entity from using a computing resource in some specific manner. A common example of resource use is reading a file. The access control has two distinct parts: policy definition where access authorizations to resources are specified, and policy enforcement where attempts to access resources are intercepted, and allowed or disallowed. An access control policy is a comprehensive set of access authorizations that govern the use of computing resources system wide. Controlling access to sensitive data in accordance with policy is perhaps the most fundamental security requirement that exists. Yet, despite more than four decades of security research, existing access control mechanisms have a limited ability to enforce a wide, comprehensive range of policies, and instead enforce a specific type of policy.

Most, if not all, significant information systems employ some means of access control. The main reason is that without sufficient access control, the service being provisioned would likely be undermined. Many types of access control policies exist. An enforcement mechanism for a specific type of access control policy is normally inherent in any computing platform. Applications built upon a computing platform typically make use of the access control capabilities available in some way to suit its needs. An application may also institute its own distinct layer of access controls for the objects formed and manipulated at the level of abstraction it provides. A common example of an application abstraction layer is a database application that implements a role-based access control mechanism, while operating on a host computer that implements a more elementary discretionary access control mechanism.

When composing different computing platforms to implement an information system, a policy mismatch can occur. A policy mismatch arises when the narrow range of policies supported by the various access control mechanisms involved have differences that make them incompatible for meeting a specific need. In some cases, it is possible to working around limitations in the ability for all platforms to express a consistent access control policy, by mapping equivalences between the available access control constructs to effect the intended policy. For example, a traditional multi-level access control system that supports information flow policies has been demonstrated as capable of effecting role-based access control policies through carefully designed and administered configuration options [Kuh98]. However, such mappings require that the correct semantic context is used consistently when administering policy, which can be mentally taxing and error inducing, and prevent the desired policy from being maintained correctly in the information system.

NIST has devised a general-purpose access control mechanism, referred to as the Policy Machine (PM), which can express and enforce arbitrary, organization-specific, attribute-based access control policies through policy configuration settings. The PM is defined in terms of a fixed set of configurable data relations and a fixed set of functions that are generic to the specification and enforcement of combinations of a wide set of attribute-based access control policies. The PM offers a new perspective on access control in terms of a fundamental and reusable set of data abstractions and functions. The goal of the PM is to provide a unifying framework that supports commonly known and implemented access control policies, as well as combinations of common policies, and policies for which no access control mechanism presently exists.

Access control policies typically span numerous systems and applications used by an organization. However, when users need to access resources that are protected under different control mechanisms, the differences in the type and range of policies supported by each mechanism can differ vastly, creating policy mismatches. If a PM mechanism were present in every computing platform, obvious benefits would be not only the elimination of policy mismatches, but also the ability to meet organizational security requirements readily, since a wider range of arbitrary policies could be expressed uniformly throughout the platforms that comprise an information system. The PM can arguably be viewed as a dramatic shift in the way policy can be specified and enforced. But more importantly, it can also be viewed as a way to develop applications more effectively by taking advantage of the underlying control mechanism available and extending policy seamlessly to meet the access control needs for objects within the layer of abstraction the application provides.

1.1 Purpose and Scope

The purpose of this Interagency Report is to provide an overview of the PM and guidelines for its implementation. The report explains the basics of the PM framework and discusses the range of policies that can be specified and enacted. It also describes the architecture of the PM and the details of key functional components.

The intended audience for this document includes the following categories of individuals:

- Computer security researchers interested in access control and authorization frameworks
- Security professionals, including security officers, security administrators, auditors, and others with responsibility for information technology security
- Executives and technology officers involved in decisions about information technology security products
- Information technology program managers concerned with security measures for computing environments.

This document, while technical in nature, provides background information to help readers understand the topics that are covered. The material presumes that readers have a basic understanding of computer security and possess fundamental operating system and networking expertise.

1.2 Document Structure

The remainder of this document is organized into the following chapters:

- Chapter 2 provides background information on access control models, including several examples of popular, well-known models.

- Chapter 3 explains the framework of the policy machine model, including key elements, relationships, and abstractions of the model, the notation for expressing policies, and some introductory examples of policy.

- Chapter 4 examines various aspects of the policy model regarding the administration of policy.

- Chapter 5 reviews ways to apply the framework to specify various types of policies.

- Chapter 6 looks at issues that arise with the integration of multiple policies and ways to apply the framework.

- Chapter 7 provides an overview of the key architectural components and interactions of the PM.

- Chapter 8 contains a list of references.

Sidebars containing auxiliary material related to the main discussion appear in gray text boxes throughout the main body of the document. At the end of the document, there are also appendices that contain supporting material. Appendix A provides a list of acronyms used in the report and Appendix B provides explanations about some of the mathematical notation used. Appendix C provides a list of core functions and commands for the PM model and their semantic description. Appendix D outlines two approaches for supporting personas within the PM model.

2. Background

Classical access control models and mechanisms are defined in terms of subjects (S), access rights (A), and named objects (O). Users represent individuals who directly interact with a system and have been authenticated and established their identities. A user identity is unique and maps to only one individual. A subject represents a user and any system process or entity that acts on behalf of a user. A user is unable to access objects directly, and instead must perform accesses through a subject (e.g., a system process that operates on behalf of the user). Subjects represent the active entities of a system that can cause a flow of information between objects or change the security state of the system.

Objects are system entities that must be protected. Each object has a unique system-wide identifier. The set of objects may pertain to processes, files, ports, and other system abstractions, as well as system resources such as printers. Subjects may also be included in the set of objects. In effect, this allows them to be governed by another subject. That is, the governing subject can administer the access of such subjects to objects under its control. The selection of entities included in the set of objects is a matter of choice determined by the protection requirements of the system.

Subjects operate autonomously and may interact with other subjects. Subjects may be permitted modes of access to objects that are different from those other subjects. When a subject attempts to access an object, a reference mediation function determines whether the subject's assigned permissions adequately satisfy policy before allowing the access to take place. In addition to carrying out user accesses, a subject may maliciously (e.g., through a Trojan horse) or inadvertently (e.g., through a coding error) make requests that are unknown to and unwanted by its user.

An access matrix provides a simple representation of the access modes to an object for which a subject is authorized [Gra72, Har76]. Figure 1 provides a simple illustration of an access matrix. Each row of the matrix represents a subject, S_i, while each column represents an object, O_i. Each entry, $A_{i,j}$, at the intersection of a row and column of the matrix, contains the set of access rights for the subject to the object. The access matrix model, while simple, can express a broad range of policies, because it is based on a general form of an access rule (i.e., subject, access mode, object), and imposes little restriction on the rule itself.

Since, in most situations, subjects do not need access rights to most objects, the matrix is typically sparse. Several, more space-efficient representations have been proposed as alternatives. An authorization relation, for example, represents an access matrix as a list of triples of the form (S_i, $A_{i,j}$, O_j). Each triple represents the access rights of a subject to an object and this representation is typically used in relational database systems [San94].

Access control and capability lists are two other forms of representation. An access control list (ACL) is associated with each object in the matrix and corresponds to a column of the access control matrix. Each access entry in the ACL contains the pair (S_i, $A_{i,j}$), which specifies the subjects that can access the object, along with each subject's rights or modes of access to the object. ACLs are widely used in present-day operating systems. Similarly, a capability list is associated with each subject and corresponds to a row of the matrix. Each entry in a capability list is the pair ($A_{i,j}$, O_j), which specifies the objects the subject can access, along with its access rights to each object. A capability list can thus be thought of as the inverse of an access control list. Capability lists, when bound with the identity of the subject, have use in distributed systems.

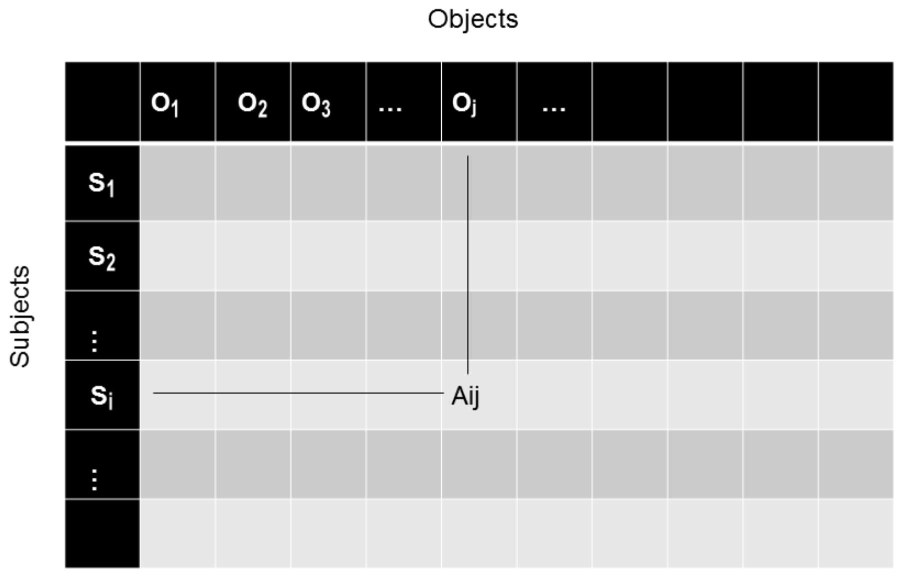

Figure 1: Access Matrix

A key difference between the capability list and access control list is the subject's ability to identify objects. With an access control list, a subject can identify any object in the system and attempt access; the access control mechanism can then mediate the access attempt using the object's access list to verify whether the subject is authorized the request mode of access. In a capability system, a subject can identify only those objects for which it holds a capability. Possessing a capability for the object is a requisite for the subject to attempt access to an object, which is then mediated by the reference mediation function. Both the contents of access control and capability lists, as well as the access control mechanism itself, must be protected from compromise to prevent unauthorized subjects from gaining access to an object.

2.1 Access Control Models

An access matrix and its various representations are a type of discretionary access control model. Discretionary in this context means that subjects, which represent users as opposed to administrators, are allowed some freedom to manipulate the authorizations of other subjects to access objects [Hu06]. Discretionary models form a broad class of access control models. Non-discretionary models are the complement of discretionary models, insofar as they establish controls that can be changed only through the actions of subjects representing administrators, and not by those representing users [Hu06]. With non-discretionary models, subjects and objects are typically classified into or labeled with distinct categories. Category-sensitive access rules that are established through administration completely govern the access of a subject to an object and are not modifiable at the discretion of the subject.

Many different access control models, both discretionary and non-discretionary, have been developed to suit a variety of purposes. Models are often developed or influenced by well-conceived organizational policies for controlling access to information. However, models typically differ from organizational policy in several ways. For instance, models deal with abstractions that involve a formal or semi-formal definition, from which the presence or lack of certain properties may be demonstrated. Policy on the other hand is usually a more informally stated set of high-level guidelines that provide a rationale for the way accesses are to be controlled, and may also give decision rules about permitting or denying certain types of access. Policies may be also incomplete, include statements at variable levels of discourse, and contain self-contradictions, while models typically involve only essential conceptual artifacts, are composed at a uniform level of discourse, and provide a consistent set of logical rules for access control.

Organizational objectives and policy for access control may not align well with those of a particular access control model. For example, some models enforce a strict policy that may too restrictive for some organizations to carry out their mission, but essential for others. Even if alignment between the two is strong, in general, the organizational access control policy may not be satisfied fully by the model. For example, different federal agencies can have different conformance directives regarding privacy that must be met, which affect the access control policy. Nevertheless, access control models can provide a strong baseline from which organizational policy can be satisfied.

Well-known models include Discretionary Access Control, Mandatory Access Control, Role Based Access Control, One-directional Information Flow, Chinese Wall, Clark-Wilson, and N-person Control. Several of these models are discussed below to give an idea of the scope and variability between models. They are also used later in the report to demonstrate how seemingly different models can be expressed using the PM model.

It is important to keep in mind that models are written at a high conceptual level, which stipulates concisely the scope of policy and the desired behavior between defined entities, but not the security mechanisms needed to reify the model for a specific computational environment, such as an operating system or database management system. While certain implementation aspects may be inferred from an access control model, such models are normally implementation free, insofar as they do not dictate how an implementation and its security mechanisms should be organized or constructed. These aspects of security are addressed through information assurance processes.

2.2 Discretionary Access Control

The access matrix discussed in the previous section is a discretionary access control (DAC) model. Many other DAC models have been derived from the access matrix and share common characteristics. In addition to an administrator’s authorized control over access to objects, DAC leaves a certain amount of control to the discretion of the object's owner. Ownership of an object is typically conferred to the subject that created the object, along with the capabilities to read and write the object. For example, it is the owner of the file who can control other subjects' accesses to the file. Control then implies possession of administrative capabilities to create and modify access control entries associated with a set of other subjects, which pertain to owned objects. Control may also involve the transfer of ownership to other subjects. Only those subjects specified by the owner may have some combination of permissions to the owner's files.

DAC policy tends to be very flexible and is widely used in the commercial and government sectors. However, DAC potentially has two inherent weaknesses [Hu06]. The first is the inability for an owner to control access to an object, once permissions are passed on to another subject. For example, when one user grants another user read access to a file, nothing stops the recipient user from copying the contents of the file to an object under its exclusive control. The recipient user may now grant any other user access to the copy of the original file without the knowledge of the original file owner. Some DAC models have the ability to control the propagation of permissions. The second weakness is vulnerability to Trojan horse attacks, which is common weakness for all DAC models. In a Trojan horse attack, a process operating on behalf a user may contain malware that surreptitiously performs other actions unbeknownst to the user.

2.3 Mandatory Access Control

Mandatory Access Control (MAC) is a prime example of a non-discretionary access control model. MAC requires that access control policy decisions are regulated by a central authority, not by the individual owner of an object. MAC has its origins with military and civilian government security policy, where individuals are assigned clearances and messages, reports, and other forms of data are assigned classifications [San94]. The security level of user clearances and of data classifications govern whether an individual can gain access to data. For example, an individual can read a report, only if the security level of the report is classified at or below his or her level of clearance.

Defining MAC for a computer system requires assignment of a security level to each subject and each object. Security levels form a strict hierarchy such that security level x dominates security level y, if and only if, x is greater than or equal to y within the hierarchy. The U.S. military security levels of Top Secret, Secret, Confidential, and Unclassified are a good example of a strict hierarchy. Access is determined based on assigned security levels to subjects and objects and the dominance relation between the subject's and object's assigned security.

The security objective of MAC is to restrict the flow of information from an entity at one security level to an entity at a lesser security level. Two properties accomplish this. The simple security property specifies that a subject is permitted read access to an object only if the subject's security level dominates the object's security level. The ★-property specifies that a subject is permitted write access to an object only if the object's security level dominates the subject's security level. Indirectly, the ★-property, also referred to as the confinement property, prevents the transfer of data from an object of a higher level to an object of a lower classification and is required to maintain system security in an automated environment.

These two properties are supplemented by the tranquility property, which can take either of two forms: strong and weak. Under the strong tranquility property, the security level of a subject or object does not change while the object is being referenced. The tranquility property serves two purposes. First, it associates a subject with a security level. Second, it prevents, a subject from

reading data with a high security level, storing the data in memory, switching its level to a low security level, and writing the contents of its memory to an object at that lower level.
Under the weak tranquility property labels are allowed to change, but never in a way that can violate the defined security policy. It allows a session to begin in the lowest security level, regardless of the user's security level, and increased that level only if objects at higher security levels are accessed. Once increased, the session security level can never be reduced, and all objects created or modified take on the security level held by the session at the time when the object was created or modified, regardless of its initial security level. This is known as the high water mark principle.

Because of the constraints placed on the flow of information, MAC models prevent software infected with Trojan horse from violating policy. Information can flow within the same security level or higher, preventing leakage to a lower level. However, information can pass through a covert channel in MAC, where information at a higher security level is deduced by inference, such as assembling and intelligently combining information of a lower security level.

2.4 Chinese Wall

The Chinese Wall policy evolved to address conflict-of-interest issues related to consulting activities within banking and other financial disciplines [Bre89]. The stated objective of the Chinese Wall policy and its associated model is to prevent illicit flows of information that can result in conflicts of interest.

The Chinese Wall policy is application-specific in that it applies to a narrow set of activities that are tied to specific business transactions. Consultants or advisors are naturally given access to proprietary information to provide a service for their clients. When a consultant gains access to the competitive practices of two banks, for instance, the consultant essentially obtains insider information that could be used to profit personally or to undermine the competitive advantage of one or both of the institutions.

The Chinese Wall model establishes a set of access rules that comprises a firewall or barrier, which prevents a subject from accessing objects on the wrong side of the barrier. It relies on the consultant's dataset to be organized such that each company dataset belongs to exactly one conflict of interest class, and objects within the same company dataset also belong to the same conflict of interest class. A subject can have access to at most one company dataset in each conflict of interest class. However, the choice of dataset is at the subject's discretion. Once a subject accesses (i.e., reads or writes) an object in a company dataset, the only other objects accessible by that subject lie within the same dataset or within the datasets of a different conflict of interest class. In addition, a subject can write to a dataset only if it does not have read access to an object that contains unsanitized information (i.e., information not treated to prevent discovery of a corporation's identity) and is in a different company dataset to the one for which write access is requested.

The following limitations in the formulation of the Chinese Wall model have been noted [San92]: a subject that has read objects from two or more company datasets cannot write at all, and a subject that has read objects from exactly one company dataset can write only to that dataset. These limitations occur because subjects include both users and processes acting on behalf of users, and can be resolved by interpreting the model differently to differentiate users from subjects [San92].

2.5 Role Based Access Control

The Role Based Access Control (RBAC) model governs the access of a user to information through roles for which the user is authorized to perform. RBAC is a more recent access control model than those described above, which is based on several entities: users (U), roles (R), permissions (P), sessions (S), and objects (O). A user represents an individual or an autonomous entity of the system. A role represents a job function or job title that carries with it some connotation of the authority held by a members of the role. Access authorizations on objects are specified for roles, instead of users. A role is fundamentally a collection of permissions to use resources appropriate to conduct a particular job function, while a permission represents a mode of access to one or more objects of a system. Objects represent the protected resources of a system.

Users are given authorization to operate in one or more roles, but must utilize a session to gain access to a role. A user may invoke one or more sessions, and each session relates a user to one or more roles. The concept of a session within the RBAC model is equivalent to the more traditional notion of a subject discussed earlier. When a user operates within a role, it acquires the capabilities assigned to the role. Through this function, the RBAC model supports the principle of least privilege, which requires that a user be given no more privilege than necessary to perform a job.

Another important feature RBAC is role hierarchies, whereby one role at a higher level can acquire the capabilities of another role at a lower level, through an explicit inheritance relation. A user assigned to a role at the top of a hierarchy, also is indirectly associated with the capabilities of roles lower in the hierarchy and acquires those capabilities as well as those assigned directly to the role. Standard RBAC also provides features to express policy constraints involving Separation of Duty (SoD) and cardinality. SoD is a security principle used to formulate multi-person control policies in which two or more roles are assigned responsibility for the completion of a sensitive transaction, but a single user is allowed to serve only in some distinct subset of those roles (e.g., not allowed to serve in more than one of two transaction-sensitive roles). Cardinality limits a role's capacity to a fixed number of users.

Two types of SoD relations exist: static separation of duty (SSD) and dynamic separation of duty (DSD). SSD relations place constraints on the assignments of users to roles, whereby membership in one role may prevent the user from being a member of another role, and thereby presumably forcing the involvement of two or more users in performing a sensitive transaction that would involve the capabilities of both roles. Dynamic separation of duty relations, like SSD relations, limit the capabilities that are available to a user, while adding operational flexibility, by placing constraints on roles that can be activated within a user's sessions. As such, a user may be a member of two roles in DSD, but unable to execute the capabilities that span both roles within a single session. Similarly, static and dynamic forms of cardinality exist to prevent respectively a limited number of users from being assigned to a role or from being active in a role simultaneously.

Certain access control models may be simulated or represented by another. For example, MAC can simulate RBAC if the role hierarchy graph is restricted to a tree structure rather than a partially ordered set [Kuh98]. RBAC is also policy neutral, and sufficiently flexible and powerful enough to simulate both DAC and MAC [Osb00]. Prior to the development of RBAC, MAC and DAC were considered to be the only classes of models for access control; if a model was not MAC, it was considered to be a DAC model, and vice versa.

3. Policy Machine Framework

The policy machine (PM) model is a redefinition of access control in terms of a standardized and generic set of relations and functions that are reusable in the expression and enforcement of policies. Its objective is to provide a unifying framework to support a wide range of attribute-based policies or policy combinations through a single mechanism. The PM can be thought of as a logical ‘‘machine” comprised of a fixed set of relations and functions between policy elements, which are used to render access control decisions.

Policy elements not only represent the users and objects of a system, but also attributes of those elements that have an effect on access control decisions. Several key relations provide a frame of reference for defining and interpreting a system policy in terms of the policy elements specified. These relations include assignments that link together policy elements into a meaningful structure, associations that are used to define authorizations for classes of users, prohibitions that are used to define what essentially are negative authorizations, and obligations that are used to perform administrative actions automatically based on event triggers. Several key functions also aid in making access control decisions and enforcing expressed policies. The remaining sections of this chapter discuss in detail core policy elements, relations, and functions that comprise the PM model.

An important characteristic of the PM is that it is inherently policy neutral. That is, no particular security policy is embodied in the PM model. Instead, the model serves a vehicle for expressing a wide range of security polices and enforcing them for a specific system through a precise specification of policy elements and relationships. Policies are composed through administrative operations that allow the expression and enforcement of both non-discretionary and discretionary policies, and also the expression and simultaneous enforcement of multiple policies. The structure of the PM is based on the concept that all enforcement can be fundamentally characterized as either static, dynamic, or historical.

3.1 Core Policy Elements

The basic data elements of the PM include authorized users (U), processes (P), system operations (SysOp), and objects (O). Users are individuals that have been authenticated by the system. Objects are system entities that are subject to control under one or more defined policies. Both users and objects have unique identifiers within the system. The set of objects reflect environment-specific entities needing protection, such as files, ports, clipboards, email messages, records and fields. The selection of entities included in this set is a matter of choice based on the protection requirements of the system. Included in the set of objects are also policy elements and relations needed by the PM to represent the authorization structure.

Operations are unique actions that can be performed on the contents of objects. Some of these operations are specific to the environment for which the PM is implemented. Common operating system operations on objects include read (r) and write (w), for example.[1] Other operations that pertain to administrative actions involving the creation and deletion of PM data elements and relations are also part of the model. That is, the entire set of system operations, SysOp, is naturally divided into a set of generic input/output operations on objects, Op, and a set of administrative operations on the data elements that represent policy, AOp. Non-administrative input/output operations are covered in the remainder of this chapter and administrative operations are covered in the next chapter.

A process is a system entity, with memory, and operates on behalf of a user. Users submit access requests through processes. Most other access control models treat users and processes uniformly, under the concept of a subject, which is defined as an active entity. The PM is different in this regard by treating users and processes as independent but related entities.

Processes can issue access requests, have exclusive access to their own memory, but none to any other process. Processes communicate and exchange data with other processes through a physical medium, such as the system clipboard or sockets. A user may be associated with one or more processes, while a process is always associated with just one user. The function process_user(p) returns the user $u \in U$ associated with process $p \in P$. A user may create and run various processes from within a session. The PM model permits only one session per user, however.

Other additional important elements of the model include policy classes (PC) and user and object attributes (UA and OA). A policy class is used to organize and distinguish between distinct types of policy being expressed and enforced. A policy class can be thought of as a container for policy elements and relationships that pertain to a specific policy. User and object attributes play a similar role. User and object attributes are policy elements used to organize and distinguish between distinct classes of users and objects respectively. They can also be thought of as containers for users and objects respectively. Every object also serves as an object attribute within the PM model; i.e., O is a subset of OA. The way in which policy elements can be assembled and used to represent policy is covered in subsequent sections.

Notation for Basic Model Elements. The basic elements of the model discussed so far can be defined more formally as shown below.

- **U:** A finite set U of authorized users; u or u_1, u_2, … denote a member of U, unless otherwise specified.

- **P:** A finite set of system processes; p or p_1, p_2, … denote a member of P, unless otherwise specified.

[1] Besides read and write, other generic input/output operations on objects may exist, depending on the computing environment. Examples include write-append, which allows an object to be expanded, but does not allow the previous contents to be changed, and execute, which allows the content of an object to be run as an executable, but does not allow it to be read. For simplicity, the more general and encompassing forms of input/output, read and write, are used exclusively throughout this report.

- **Op:** A finite set of generic input/output operations; op or op_1, op_2, … denote a member of Op, unless otherwise specified.

- **AOp:** A finite set of administrative operations; aop or aop_1, aop_2, … denote a member of AOp, unless otherwise specified.

- **SysOp:** The finite set of administrative and non-administrative operations for a system.
  $SysOp = Op \cup AOp$

- **O:** A finite set of protected objects; o or o_1, o_2, … denote a member of O, unless otherwise specified.
  $O \subseteq OA$

- **PC:** A finite set of policy classes; pc or pc_1, pc_2, … denote a member of PC, unless otherwise specified.

- **UA:** A finite set of user attributes; ua or ua_1, ua_2, … denote a member of U, unless otherwise specified.

- **OA:** A finite set of object attributes; oa or oa_1, oa_2, … denote a member of OA, unless otherwise specified.
  $OA \supseteq O$

- **Process-to-User Mapping:** The function process_user from domain P to codomain U, such that u = process_user(p), iff $p \in P$ is a process operating on behalf of user $u \in U$.
  $\forall p \in P, \exists u \in U$: u = process_user(p)

3.2 Assignments and Relations between Elements

Assignments are the means used to express a relationship between users and user attributes, objects and object attributes, user (object) attributes and user (object) attributes, and user (object) attributes and policy classes. The assignment relationship is a binary relation on the set of policy elements, $PE = U \cup UA \cup OA \cup PC$, where $O \subseteq OA$. The assignment relation is denoted by the arrow symbol "→" and can be expressed as either $(x, y) \in \rightarrow$ or $x \rightarrow y$, on elements x, y of PE. The relation is defined as follows:

$$\rightarrow \subseteq (U \times UA) \cup (UA \times UA) \cup (OA \times OA) \cup (UA \times PC) \cup (OA \times PC)$$

The assignment relation must satisfy the following properties:

- It is irreflexive; i.e., for all x, y in PE, $x \rightarrow y \Rightarrow x \neq y$.

- It is acyclic; i.e., there does not exist a finite sequence of distinct elements $x_1, x_2, ..., x_n$ in PE, such that $n > 1 \land x_i \rightarrow x_{i+1}$ for $i = 1,2,...,n-1 \land x_n \rightarrow x_1$.

- A sequence of assignments (i.e., a path) must exist from every element in U, UA, and OA to some element in PC; i.e., for all elements w in $U \cup UA \cup OA$, there exists a sequence of distinct elements $x_1,x_2,...,x_n$ in PE, such that $n > 1 \land x_1 = w \land x_n \in PC \land x_i \to x_{i+1}$ for $i = 1,2,...,n\text{-}1$.

- An object attribute cannot be assigned to an object; i.e., there does not exist an assignment $x \to y$, such that $x \in OA$ and $y \in O$.

The assignment relation can be represented as a directed graph or digraph G = (PE, →), where PE are the vertices of the graph, and each tuple (x, y) of → represents a direct edge or arc that originates at x and terminates at y. A digraph of policy elements and the assignment among them is also referred to as a policy element diagram within this report and is a key concept underlying the PM model. A policy graph is typically oriented in a top-down fashion with the head of an arrow (i.e., its termination) pointing downward, as shown in the simplified policy element diagram of Figure 2, which illustrates assignments between each type of policy element.

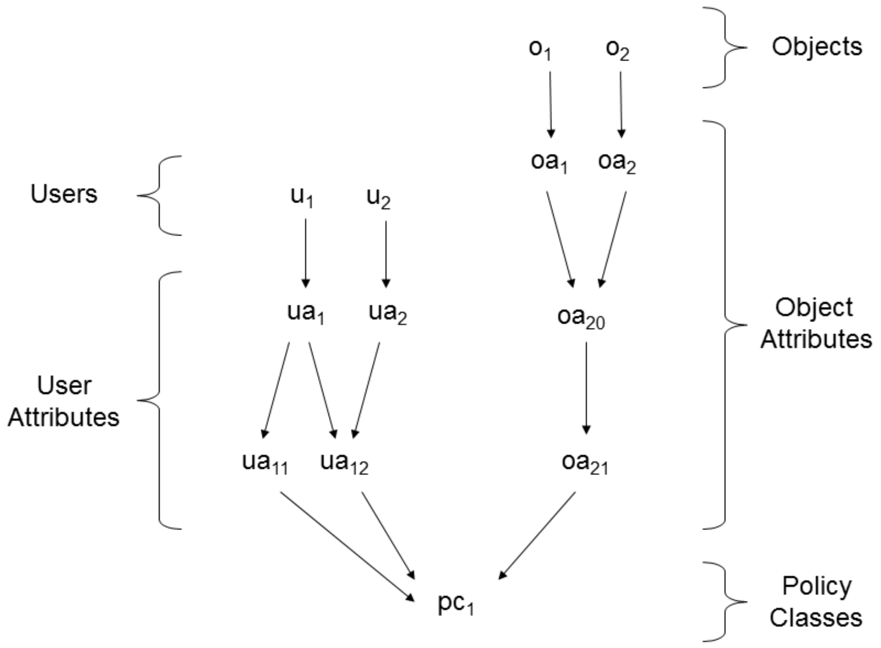

Figure 2: Simplified Policy Element Diagram

The transitive closure of the relation →, denoted as $\to^+$, provides a convenient way to determine whether one element in PE is reachable from another through a series of one or more assignments. The expression $x \to^+ y$ denotes that y is reachable from x. For all x and y in PE, (x, y) is a member of $\to^+$, if and only if (iff) there exists a sequence of distinct elements $x_1,x_2,...,x_n$ in PE, such that $n > 1 \land x_i \to x_{i+1}$ for $i = 1,2,...,n\text{-}1 \land x=x_1 \land y=x_n$. For example, in Figure 2, ua_{12} is reachable from u_1, u_2, ua_1, ua_2, which can be expressed as $u_1 \to^+ ua_{12}$, $u_2 \to^+ ua_{12}$, $ua_1 \to^+ ua_{12}$, $ua_2 \to^+ ua_{12}$. Reachability is related to the concept of containment. For any x and y in PE, x is said to be contained in y, or y is said to contain x, iff $x \to^+ y$. In the previous example involving ua_{12}, ua_{12} can be said to contain u_1, u_2, ua_1, ua_2.

Occasionally, it is useful to express that one element in PE is reachable from another through a series of zero or more assignments. The reflexive and transitive closure of the relation →, denoted as $\rightarrow^{*}$, provides a convenient way to represent this situation. That is, for any x and y in PE, $x \rightarrow^{*} y$ is the equivalent of stating that y contains x or is itself the element x.

3.2.1 User, Object, and Attribute Relationships

A user may be assigned to one or more user attributes. The assignment $u \rightarrow ua$ means that the user is assigned to or contained in the class represented by ua. It also denotes that user u takes on or inherits the properties held or represented by the attribute ua. The properties of a user attribute are capabilities for and prohibitions against accessing certain types of objects.

Similarly, an object may be assigned to one or more object attributes through one or more object-to-attribute assignments, represented as a binary relation from O to OA. The assignment $o \rightarrow oa$ means that that the object o is contained in the class represented by oa and takes on or inherits the properties held by the attribute oa. The properties of an object attribute are capabilities and prohibitions allotted to users, which govern access to contained objects (i.e., access modes allowed and denied to specific users).

3.2.2 Relationships among Attributes

A user (object) attribute may be assigned to one or more other user (object) attributes. Because the assignment relation is acyclic, a hierarchical ordering among attributes can be established through a series of assignments. Assignments between attributes are by definition restricted to attributes of the same type (i.e., either all user attributes or object attributes). Therefore, no members of an object attribute hierarchy can be in common with those of a user attribute hierarchy and vice versa—they are mutually exclusive.

The user attribute hierarchy UH is a subrelation of the relation $\rightarrow$ in $UA \times UA$, which is defined as $(UA \times UA) \cap \rightarrow$, such that every tuple of UH is also a tuple of $\rightarrow$. That is, for all x and y in UA, xUHy, iff $x \rightarrow y$. The object attribute hierarchy OH can be defined similarly as $(OA \times OA) \cap \rightarrow$, a subrelation of $\rightarrow$ in $OA \times OA$, such that, for all x and y in OA, $x \rightarrow y$, iff xOHy. In general, a relation S is a subrelation of relation R, if every tuple of S is a tuple of R. Note that several other subrelations of $\rightarrow$ also exist, including user to user attribute assignments, defined as $(U \times UA) \cap \rightarrow$, user attribute to policy class assignments, defined as $(UA \times PC) \cap \rightarrow$, and object attribute to policy class assignments, defined as $(OA \times PC) \cap \rightarrow$.

Containment is of key importance for an attribute hierarchy. Containment within an attribute hierarchy allows each attribute to inherit the properties held by every attribute that contains it. As mentioned earlier, an attribute or other policy element x is said to be contained in another attribute or policy element y, iff $x \rightarrow^{+} y$. For example, focusing exclusively on the object attributes in Figure 2, the following expressions are true: $oa_1 \rightarrow^{+} oa_{20}$, $oa_2 \rightarrow^{+} oa_{20}$, $oa_1 \rightarrow^{+} oa_{21}$, $oa_2 \rightarrow^{+} oa_{21}$, and $oa_{20} \rightarrow^{+} oa_{21}$. That is, within the object attribute hierarchy, both oa_1 and oa_2 are contained in oa_{20} and inherit the properties of oa_{20}, and oa_1, oa_2, and oa_{20} are contained in oa_{21} and likewise inherit its properties.

Inheritance of properties within an attribute hierarchy also has an effect on the way users and objects contained by those attributes are treated within the PM model. A user x that is contained in user attribute y, can gain the properties that are both assigned to and inherited by attribute y.

Similarly, an object x that is contained in object attribute y, can gain the properties that are both assigned to and inherited by attribute y.

3.2.3 Policy Class Relationships

A user attribute or an object attribute may be assigned to one or more policy classes (e.g., ua→pc or oa→pc). Properties that are assigned to a policy class are inherited by the attributes assigned to it. As mentioned earlier, a policy class can be thought of as a container for policy elements and relationships that pertain to a specific policy; every element is contained in at least one policy class. Unlike attributes, however, a policy class cannot be assigned to any other policy class.

Elements of one policy class can be defined to be mutually exclusive from those of another policy class. That is, if a policy element x is contained in pc_1, it is precluded from being contained in pc_2. Policy elements can also be defined to be inclusive of more than one policy class. An access control policy can be characterized through a single policy class, multiple mutually exclusive policy classes, or multiple non-mutually exclusive policy classes.

Notation for Element Relationships. The relationships among elements of the PM model discussed so far can be defined more formally as shown below.

- **PE:** A finite set of policy elements, where PE $\stackrel{\text{def}}{=}$ U ∪ UA ∪ OA ∪ PC (i.e., {U, UA, OA, PC} is a partition on the set PE); pe or pe_1, pe_2, … denote arbitrary members of PE, unless otherwise specified.

- **Assignment:** The binary relation → in the set PE, such that the following hold:
  - $\rightarrow \subseteq (U\times UA) \cup (UA\times UA) \cup (OA\times OA) \cup (UA\times PC) \cup (OA\times PC)$
  - the relation is irreflexive; i.e., $\forall x,y \in PE: (x \rightarrow y \Rightarrow x \neq y)$
  - the relation is acyclic; i.e., ∄ a finite sequence of distinct elements $x_1,x_2,...,x_n \in PE$, such that $n > 1 \wedge x_i \rightarrow x_{i+1}$ for $i = 1,2,...,n-1 \wedge x_n \rightarrow x_1$
  - a path exists from every element in U, UA, and OA to some element in PC; i.e., $\forall w \in (U \cup UA \cup OA)$, ∃ a finite sequence of distinct elements $x_1,x_2,...,x_n \in PE$, such that $n > 1 \wedge x_1=w \wedge x_n \in PC \wedge x_i \rightarrow x_{i+1}$ for $i = 1,2,...,n-1$
  - assignments to an object from an object attribute are precluded; i.e., $\forall x \in OA: \nexists y \in O$ such that $x \rightarrow y$

- **Policy Element Diagram:** A policy element diagram is an ordered pair (→, PE) where → is an assignment relation in the set PE.

- **Containment:** The binary relation $\rightarrow^+$; i.e., $\rightarrow^+$ is the transitive closure of the assignment relation →.
  - $\rightarrow \subseteq \rightarrow^+$
  - $\forall x, y \in PE$: (x, y) is a member of $\rightarrow^+$, if and only if (iff) ∃ a finite sequence of distinct elements $pe_1,pe_2,...,pe_n \in PE$, such that $n > 1 \wedge pe_i \rightarrow pe_{i+1}$ for $i = 1,2,...,n-1 \wedge x=pe_1 \wedge y=pe_n$
  - x *is contained in* y $\stackrel{\text{def}}{=}$ $x,y \in PE \wedge x \rightarrow^+ y$
  - y *contains* x $\stackrel{\text{def}}{=}$ $x,y \in PE \wedge x \rightarrow^+ y$

The notation $x \to^* y$ is a shorthand expression of the condition that y is reachable from x through a series of zero or more applications of the assignment relation (i.e., the reflexive and transitive closure of $\to$).

User Oriented

- **User-to-User Attribute Assignment:** The binary relation UUA over the policy elements $(U \times UA) \cap \to$ is a subrelation of the binary relation $\to$.
  - $UUA = (U \times UA) \cap \to \subseteq \to$
  - $\forall x \in U, \forall y \in UA$: (x UUA y, iff $x \to y$)

- **User-Attribute-to-Attribute Assignment:** The binary relation UH over the policy elements $(UA \times UA) \cap \to$ is a subrelation of the binary relation $\to$.
  - $UH = (UA \times UA) \cap \to \subseteq \to$
  - $\forall x \in UA, \forall y \in UA$: (x UH y, iff $x \to y$)
  - user attribute x *inherits the properties of* attribute y $\overset{\text{def}}{=} x \in UA \land y \in UA \land x \to^+ y$

- **User-Attribute-to-Policy Class Assignment:** The binary relation UAPC over the policy elements $(UA \times PC) \cap \to$ is a subrelation of the binary relation $\to$.
  - $UAPC = (UA \times PC) \cap \to \subseteq \to$
  - $\forall x \in UA, \forall y \in PC$: (x UAPC y, iff $x \to y$)
  - attribute x *inherits the properties of* policy class y $\overset{\text{def}}{=} x \in UA \land y \in PC \land x \to^+ y$

Object Oriented

- **Object-Attribute-to-Attribute Assignment:** The binary relation OH over the policy elements $(OA \times OA) \cap \to$ is a subrelation of the binary relation $\to$.
  - $OH = (OA \times OA) \cap \to \subseteq \to$
  - $\forall x \in OA, \forall y \in OA$: (x OH y, iff $x \to y$)
  - object attribute x *inherits the properties of* attribute y $\overset{\text{def}}{=} x \in OA \land y \in OA \land x \to^+ y$

- **Object-Attribute-to-Policy Class Assignment:** The binary relation OAPC over the policy elements $(OA \times PC) \cap \to$ is a subrelation of the binary relation $\to$.
  - $OAPC = (OA \times PC) \cap \to \subseteq \to$
  - $\forall x \in OA, \forall y \in PC$: (x OAPC y, iff $x \to y$)
  - attribute x *inherits the properties of* policy class y $\overset{\text{def}}{=} x \in OA \land y \in PC \land x \to^+ y$

3.3 Associations and Privileges

Associations and privileges are used to define and derive additional relationships that involve authorized operations between policy elements. They are closely related to one another, and as shown later in this section, shaped in part by attribute hierarchies that have also been defined to express policy.

3.3.1 Associations

Associations are policy settings that govern which users are authorized to access which objects and exercise which operations. More specifically, associations represent a ternary relation between the policy elements UA, Ops, and OA, where $Ops = 2^{Op} - \{\emptyset\}$ (i.e., the set of all subsets of Op, except for the empty set). Associations are normally formed and rescinded through administrative commands issued with an according interface of the PM.

The ternary relation ASSOC ⊆ UA × Ops × OA defines the set of possible associations within a policy specification. An individual triple (ua, ops, oa) of ASSOC, where ops ∈ Ops represents a set of operations, can be denoted as ua—ops—oa. Within one policy class, an association ua—ops—oa specifies that all users contained in ua possess the authority to perform all operations in ops on all objects contained in oa. Note that associations affecting a user's mode of access to objects can occur at multiple levels within an attribute hierarchy. Similarly, associations that affect an object's accessibility by users can also occur at multiple levels.

Associations can be formed within the PM and interpreted using either an access list or a capability list orientation. That is, an individual association can be represented from the perspective of a user or object attribute, using a pair of binary relations as illustrated in Figure 3. The top of Figure 3(a) illustrates the ternary relation, while the bottom, left side of Figure 3(b) illustrates the inherent access list-oriented representation (i.e., the implicit representation drawn from an object attribute's perspective), and the right side of Figure 3(b) illustrates the inherent capability-oriented representation (i.e., the implicit representation drawn from a user attribute's perspective). The ability to form associations from either orientation allows flexibility when adapting PM model abstractions to a specific system implementation environment. Care should be taken, however, to maintain one orientation consistently throughout.

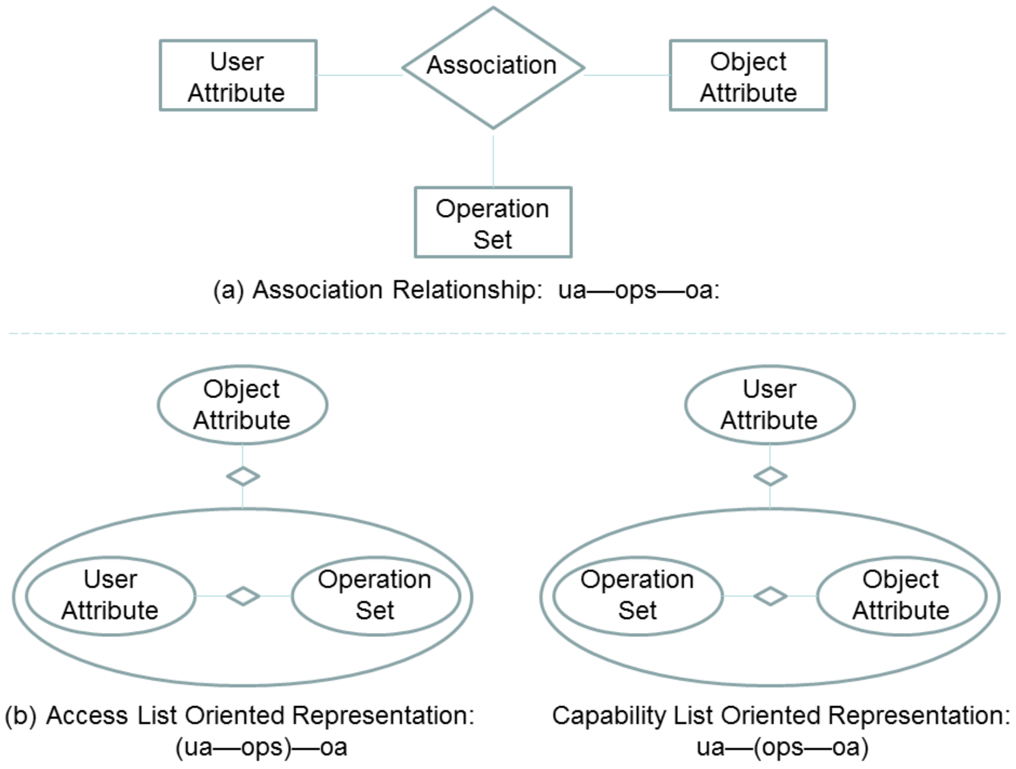
User Attribute
Association
Object Attribute
Operation Set
(a) Association Relationship: ua—ops—oa:
Object Attribute
User Attribute
Operation Set
User Attribute
Operation Set
Object Attribute
(b) Access List Oriented Representation: (ua—ops)—oa
Capability List Oriented Representation: ua—(ops—oa)

Figure 3: Associations and Alternative Representations

3.3.2 Inheritance and Attribute Properties

Attribute hierarchies play an important role in forming associations. The relationships formed through associations between attributes are properties subject to inheritance. As mentioned above, the properties of an attribute include not only those directly held by the attribute, but also the properties inherited from every attribute in which it is contained. Stated slightly differently, the properties of an attribute include not only those directly held by the attribute, but also the properties inherited from every attribute that contains it.

Figure 4 gives a simple example of an authorization graph containing attribute hierarchies, with policy elements U = {u_1, u_2, u_3}, O = {o_1, o_2, o_3}, UA = {Group1, Group2, Division}, OA = {Project1, Project2, Projects}, and PC = {OU}. An authorization graph is simply a policy element diagram annotated with associations and other relationships that exist between policy elements. Associations are illustrated using dotted lines between the elements involved in each association. The following three associations are shown in Figure 4: (Group1, {w}, Project1), (Group2, {w}, Project2), and (Division, {r}, Projects). Looking at the properties of each user attribute in the hierarchy that are assigned or inherited from the defined associations, the following can be determined:

- Group1 is assigned the capability of ({w}, Project1) and inherits the capability of ({r}, Projects) from Division.
- Group2 is assigned the capability of ({w}, Project2) and inherits the capability of ({r}, Projects) from Division.
- Division is assigned the capability of ({r}, Projects), but inherits no capabilities, since it is not contained in any another user attribute.

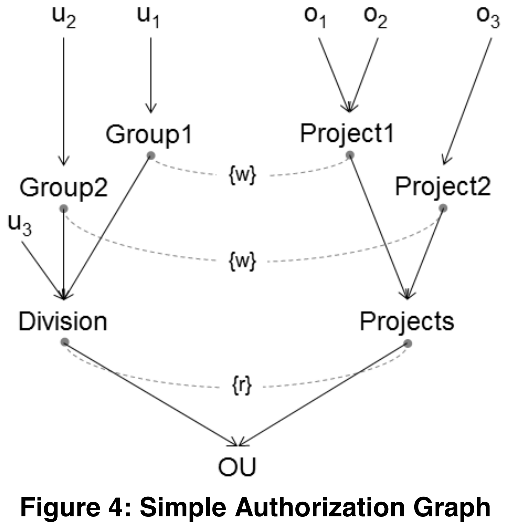

Figure 4: Simple Authorization Graph

For this same example, the properties of each object attribute in the hierarchy that are assigned or inherited from the defined associations can also be determined. That is, rather than a list of inherent capabilities, a list of inherent access entries can be determined for each object attribute.

- Project1 is assigned the access entry (Group1, {w}), and inherits the access entry (Division, {r}) from Projects.
- Project2 is assigned the access entry (Group2, {w}), and inherits the access entry (Division, {r}) from Projects.
- Projects is assigned the access entry (Division, {r}), but inherits no access entries from another object attribute.

While it is relatively easy to determine the assigned and inherited properties of attributes for the simple example given in Figure 4, it would be considerably more difficult to illustrate and analyze a more realistic example. The interactions between vertical assignment relations and horizontal association relations increase in complexity quickly as more elements and their relationships with other elements are added to an authorization graph.

3.3.3 Derived Privileges

A privilege specifies a relationship between a user, an operation, and an object. Privileges are derived from higher level abstractions that define policy, particularly the associations between and assignments among elements of attribute hierarchies discussed in the previous section. That is, every privilege originates from an association and the containment properties of the user and object attributes of that association.

The ternary relation PRIV $\subseteq$ U $\times$ Op $\times$ O defines the set of possible privileges within a policy specification. A generic individual privilege of the form (u, op, o) denotes that user u has the authority to perform operation op on object o. Within a policy consisting of a single policy class, a triple (u, op, o) is a privilege, iff there exists a user attribute ua with an assigned or inherited capability (ops, oa), such that u$\rightarrow$ua, op $\in$ ops, and o$\rightarrow^*$oa.

A privilege can also be derived from the object's perspective. That is, a triple (u, op, o) is a privilege, iff there exists an object attribute oa with an assigned or inherited access entry (ua, ops), such that o$\rightarrow^*$oa, u$\rightarrow^+$ua, and op $\in$ ops. Privileges can also be derived in a more straightforward, perspective-independent fashion for policies consisting of a single policy class. Specifically, the triple (u, op, o) is a privilege, iff there exists an association (ua, ops, oa), such that user u$\rightarrow^+$ua, op $\in$ ops, and o$\rightarrow^*$oa. Policies that involve multiple policy classes require a small adjustment to privilege derivation, which is discussed later in Chapter 6.

Looking again at the example in Figure 4, the entire set of privileges for the authorization graph can be enumerated from the user's perspective and the capabilities of attributes directly assigned to it, as follows:

- u_1 is assigned to Group1, which has the inherent capabilities of ({w}, Project1) and ({r}, Projects). Since Project1 contains o_1 and o_2 and Projects contains o_1, o_2, and o_3, the derived privileges involving u_1 are (u_1, w, o_1), (u_1, w, o_2), (u_1, r, o_1), (u_1, r, o_2), and (u1, r, o_3).

- u_2 is assigned to Group2, which has the inherent capabilities of ({w}, Project2) and ({r}, Projects). Since Project2 contains o_3 and Projects contains o_1, o_2, and o_3, the derived privileges involving u_2 are (u_2, w, o_3), (u_2, r, o_1), (u_2, r, o_2), and (u_2, r, o_3).

- u_3 is assigned to Division, which has the inherent capability of ({r}, Projects). Since Projects contains o_1, o_2, and o_3, the derived privileges involving u_3 are (u_3, r, o_1), (u_3, r, o_2), and (u_3, r, o_3).

As mentioned earlier, the same set of privileges enumerated above can be derived in a similar fashion from the authorization graph, taking the object's perspective and the properties of attributes directly assigned to it. While it is possible to represent the derived privileges of any authorization graph involving associations as an access matrix, the PM model allows groups of users and objects to be organized collectively in a manner intended to facilitate administration.

Similar to inherent properties of associations, both access entry and capability orientations apply also to privileges. A user u may access an object via its capability, (op, o), iff the privilege (u, op, o) exists. Likewise, a user may access an object o via the object's access entry, (u, op), iff a privilege (u, op, o) exists.

Although privileges can be envisaged in terms of user capabilities or object access entries, the reference mediation function controls access in terms of processes. That is, the reference mediation function grants the process p the permission to execute an access request $<op, o>_p$, iff the privilege (u, op, o) exists, where u = process_user(p). It is important to note that the variable op is used for two distinct purposes. In an access request, it designates a single abstract input/output operation on the object, while in a privilege, it designates an access right that authorizes an unlimited number of abstract input/output operations on an object.

Notation for Associations and Privileges. The relationships among elements of the PM model formed through associations and privileges can be defined more formally as shown below.

- **Ops:** A finite set of all subsets of operations defined in Op, excluding the empty set; ops or ops_1, ops_2, … denote a member of Ops, unless otherwise specified.
  $Ops = 2^{Op} – \{\emptyset\}$

- **Associations:** The ternary relation ASSOC from UA to Ops to OA.
  $ASSOC \subseteq UA \times Ops \times OA$

- **Inherent Capabilities:** The partial function ICap from UA to $2^{(Ops \times OA)}$.
  - $ICap \subseteq UA \times 2^{(Ops \times OA)}$
  - $\forall ua \in UA, \forall ops \in Ops, \forall oa \in OA$: ((ops, oa) $\in$ ICap(ua), iff (ua, ops, oa) $\in$ ASSOC)

- **Inherent Access Entries:** The partial function IAE from OA to $2^{(UA \times Ops)}$.
  - $IAE \subseteq OA \times 2^{(UA \times Ops)}$
  - $\forall ua \in UA, \forall ops \in Ops, \forall oa \in OA$: ((ua, ops) $\in$ IAE(oa), iff (ua, ops, oa) $\in$ ASSOC)

- **Privileges:** The ternary relation PRIV from U to Op to O.
  - $PRIV \subseteq U \times Op \times O$

- $\forall u \in U, \forall op \in Op, \forall o \in O$: ((u, op, o) ∈ PRIV, iff $\exists ops \in Ops, \exists ua \in UA, \exists oa \in OA$: ((ua, ops, oa) ∈ ASSOC $\wedge$ u $\rightarrow^{+}$ ua $\wedge$ op ∈ ops $\wedge$ o $\rightarrow^{*}$ oa))

▪ **Access Entries:** The function AE from O to $2^{(U \times Op)}$.
- AE $\subseteq$ O × $2^{(U \times Op)}$
- $\forall u \in U, \forall op \in Op, \forall o \in O$: ((u, op) ∈ AE(o), iff (u, op, o) ∈ PRIV)

▪ **Capabilities:** The function Cap from U to $2^{(Op \times O)}$.
- Cap $\subseteq$ U × $2^{(Op \times O)}$
- $\forall u \in U, \forall op \in Op, \forall o \in O$: ((op, o) ∈ Cap(u), iff (u, op, o) ∈ PRIV)

▪ **Process Capabilities:** The function PCap from P to $2^{(Op \times O)}$.
- PCap $\subseteq$ P × $2^{(Op \times O)}$
  $\forall p \in P, \forall op \in Op, \forall o \in O$: ((op, o) ∈ PCap(p), iff (process_user(p)., u, op, o) ∈ PRIV)

▪ **Access Request:** A finite set of possible process access requests.
- AReq $\subseteq$ P × (Op×O)
- (p, (op, o)) ∈ AReq $\overset{\text{def}}{=}$ $<op, o>_p$

▪ **Reference Mediation:** The function from domain AReq to codomain {grant, deny}.
$\forall p \in P, \forall op \in Op, \forall o \in O$: (reference_mediation($<op, o>_p$) = grant, iff (op, o) ∈ PCap(p));
otherwise, reference_mediation($<op, o>_p$) = deny

3.4 Prohibitions

Two distinct, but related types of fundamental prohibitions exist: user deny and process deny. User and process denies are generally referred to as prohibitions because they override privileges that would otherwise allow access to an object occur. That is, prohibitions denote an effective set of privileges that a specific user or process is precluded from exercising, regardless of whether any of the privileges involved actually can or cannot be derived for the user or process in question. Prohibitions can be formed and rescinded through administrative commands issued with an according interface of the PM, similar to associations.

A couple of notational conventions help to facilitate the discussion of prohibitions. Let $pe^{\blacklozenge}$ denote the set of all objects contained by the policy element pe (i.e., $pe^{\blacklozenge}$ = {x: x∈O and $x \rightarrow^{+} pe$}). The complement of $pe^{\blacklozenge}$ with respect to the set of all objects, O, is denoted by $pe^{\lozenge}$ (i.e., $pe^{\lozenge} = O - pe^{\blacklozenge}$). The notation $pe^{\blacklozenge}$ and $pe^{\lozenge}$ are respectively called the object range and complementary object range of a policy element. They are used below to define two classes of prohibitions that involve disjunctive and conjunctive relationships, and the objects affected by them.

The quaternary relation U_deny_disjunctive $\subseteq$ U × Ops × OAs × OACs, where OAs = OACs = 2^{OA}, defines the set of user-based disjunctive prohibitions for a policy specification. An individual tuple (u, ops, oas, oacs) ∈ U_deny_disjunctive, where u ∈ U, ops ∈ Ops, oas ∈ OAs, oacs ∈ OACs, and oas ∪ oacs ≠ ∅, denotes that any process p executing on behalf of user u (i.e., u = process_user(p)) cannot perform the operations in ops on any object that is contained by at least one of the object attributes in oas (i.e., inclusory object attributes), or not contained by at least one of the object attributes in oacs (i.e., exclusory object attributes). More precisely, the set of objects affected by a disjunctive user deny is the union of $oa_i^{\blacklozenge}$, for all oa_i in oas, and $oac_j^{\lozenge}$, for all oac_j in oacs (i.e., the set $(oa_1^{\blacklozenge} \cup oa_2^{\blacklozenge} \ldots \cup oa_n^{\blacklozenge}) \cup (oac_1^{\lozenge} \cup oac_2^{\lozenge} \ldots \cup oac_m^{\lozenge})$).

A complementary relation to U_deny_disjunctive is also defined for the PM model. An individual tuple (u, ops, oas, oacs) of the quaternary relation U_deny_conjunctive ⊆ U × Ops × OAs × OACs denotes that any process p executing on behalf of user u cannot perform the operations in ops on any object that is contained by all of the attributes in oas, and is also not contained by any of the object attributes in oacs. As specified above, oas ∪ oacs ≠ ∅. Stated more precisely, the set of objects affected by a conjunctive user deny is the intersection of $oa_i^{\blacklozenge}$, for all oa_i in oas, and $oac_j^{\lozenge}$, for all oac_j in oacs (i.e., the set $(oa_1^{\blacklozenge} \cap oa_2^{\blacklozenge} \ldots \cap oa_n^{\blacklozenge}) \cap (oac_1^{\lozenge} \cap oac_2^{\lozenge} \ldots \cap oac_m^{\lozenge})$).

Process-based prohibitions are defined similarly to user-based prohibitions. The relation P_deny_disjunctive ⊆ P × Ops × OAs × OACs defines the set of process-based disjunctive prohibitions. A tuple (p, ops, oas, oacs) ∈ P_deny_disjunctive denotes that the process p cannot perform the operations in ops on any object that is contained by at least one of the object attributes in oas, or not contained by at least one of the object attributes in oacs.

The conjunctive form of a prohibition also exists for process-based prohibitions. The relation P_deny_conjunctive ⊆ P × Ops × OAs × OACs defines the set of process-based conjunctive prohibitions. A tuple (p, ops, oas, oacs) ∈ P_deny_conjunctive denotes that the process p cannot perform the operations in ops on any object that is contained by all of the attributes in oas, and is also not contained by any of the object attributes in oacs. Note that if all existing prohibitions for a user are process-based prohibitions that apply to only a single user process, it may be possible for the user to perform prohibited access operations through another of its processes, presuming that the appropriate associations are defined that would allow such access. This situation can be easily remedied through the use of a user-based prohibition, whose scope is broader than a single process.

Besides user and process-based prohibitions, other types of useful prohibitions can be defined, such as the following pair based on user attribute. The quaternary relation UA_deny_disjunctive ⊆ UA × Ops × OAs × OACs defines the set of user attribute-based disjunctive prohibitions, and the quaternary relation UA_deny_conjunctive ⊆ UA × Ops × OAs × OACs defines the set of user attribute-based conjunctive prohibitions. A tuple (ua, ops, oas, oacs) ∈ UA_deny_disjunctive denotes that any process p, executing on behalf of some user u that is contained in ua, cannot perform the operations in ops on any object that is contained by at least one of the object attributes in oas, or not contained by at least one of the object attributes in oacs. Similarly, a tuple (ua, ops, oas, oacs) ∈ UA_deny_conjunctive denotes that any process p, executing on behalf of some user u that is contained in ua, cannot perform the operations in ops on any object that is contained by all of the attributes in oas, and is also not contained by any of the object attributes in oacs.

The disjunctive and conjunctive forms of prohibitions allow complex expressions to be specified, which delineate the objects targeted by a prohibition. In practice, most policies typically require the use of only simple expressions in prohibitions. For example, the sets oas and oacs may each be a singleton and contain only one member, or one of the sets may be the empty set and the other a singleton. However, the capabilities that are defined are intended to meet the demands of more complex policies that might arise. While the range of expressions is substantial, limitations do exist, which may necessitate slight adjustments to the policy graph to be able to capture a prohibition-related policy requirement adequately.

Prohibitions take precedence over any defined associations and derived privileges during reference mediation. An access request to an object is granted a user or process acting on behalf of the user, iff the appropriate associations are defined that allow such access, and there is not a prohibition for that user or process on the requested object, which countermands the access operation in question. If such a prohibition does exist, access is denied. That is, when prohibitions apply, the reference mediation function grants the process p permission to execute a request $<op, o>_p$, iff for some u = process_user(p), the following conditions hold:

- The privilege (u, op, o) exists.
- There do not exist prohibitions (p, ops, oas, oacs) $\in$ P_deny_disjunctive or (u, ops, oas, oacs) $\in$ U_deny_disjunctive, such that op $\in$ ops and for some member x of oas, $o \in x^{\blacklozenge}$, or for some member y of oacs, $o \in y^{\lozenge}$.
- There do not exist prohibitions (p, ops, oas, oacs) $\in$ P_deny_conjunctive or (u, ops, oas, oacs) $\in$ U_deny_conjunctive, such that op $\in$ ops and for all members x of oas, $o \in x^{\blacklozenge}$, and for all members y of oacs, $o \in y^{\lozenge}$.
- There do not exist prohibitions (ua, ops, oas, oacs) $\in$ UA_deny_disjunctive, such that op $\in$ ops, $u \rightarrow^{+} ua$, and for some member x of oas, $o \in x^{\blacklozenge}$, or for some member y of oacs, $o \in y^{\lozenge}$.
- There do not exist prohibitions (ua, ops, oas, oacs) $\in$ UA_deny_conjunctive, such that op $\in$ ops, $u \rightarrow^{+} ua$, and for all members x of oas, $o \in x^{\blacklozenge}$, and for all members y of oacs, $o \in y^{\lozenge}$.

Otherwise, the requested access is denied.

A user-based prohibition is persistent and remains in existence until it is rescinded through an administrative action. A user-based prohibition cannot be partially rescinded and must be rescinded in its entirety. That is, even if a subset of a prohibition's affected privileges needs to be restored, the entire prohibition still must be rescinded and replaced with new prohibition for the remaining subset that are still in effect. Process-based prohibitions are usually formed through predefined rules known as obligations, which are executed automatically based on event occurrence. A process-based prohibition is less enduring and handled differently than a user-based prohibition; once the process terminates, the prohibition no longer has applicability and is rescinded automatically by the PM.

Notation for Prohibitions. The relationships among elements of the PM model affected by prohibitions can be defined more formally as shown below.

- **OAs:** The finite set of all subsets of object attributes defined in OA; oas or oas_1, oas_2, … denote a member of OAs, unless otherwise specified.
  $OAs = 2^{OA}$

- **OACs:** The finite set of all subsets of object attributes defined in OA; oacs or $oacs_1$, $oacs_2$, … denote a member of OAs, unless otherwise specified.
  $OACs = 2^{OA}$

- **Object Range of a Policy Element:** The set of objects contained by a policy element.
  $\forall pe \in PE: pe^{\blacklozenge} \overset{\text{def}}{=} \{x: x \in O \land x \rightarrow^{+} pe\}$.

- **Complementary Object Range of a Policy Element:** The set of objects not contained by a policy element.
  $\forall pe \in PE: pe^{\lozenge} \overset{\text{def}}{=} O - pe^{\blacklozenge}$.

- **User Deny Disjunctive Prohibition:** The quaternary relation U_deny_disjunctive from U to Ops to OAs to OACs.
  $U\_deny\_disjunctive \subseteq U \times Ops \times OAs \times OACs$
  $\forall p \in P, \forall op \in Op, \forall o \in O: ((op, o) \in PCap(p) \land \exists ops \in OPs, \exists oas \in OAs,$
  $\exists oacs \in OACs: ((process\_user(p), ops, oas, oacs) \in U\_deny\_disjunctive \land$
  $op \in ops \land (\exists oa \in oas: o \in oa^{\blacklozenge} \lor \exists oa \in oacs: o \in oa^{\lozenge}))$
  $\Rightarrow reference\_mediation(<op, o>_p) = deny)$

- **User Deny Conjunctive Prohibition:** The quaternary relation U_deny_conjunctive from U to Ops to OAs to OACs.
  $U\_deny\_conjunctive \subseteq U \times Ops \times OAs \times OACs$
  $\forall p \in P, \forall op \in Op, \forall o \in O: ((op, o) \in PCap(p) \land \exists ops \in OPs, \exists oas \in OAs,$
  $\exists oacs \in OACs: ((process\_user(p), ops, oas, oacs) \in U\_deny\_conjunctive \land$
  $op \in ops \land (\forall oa \in oas: o \in oa^{\blacklozenge} \land \forall oa \in oacs: o \in oa^{\lozenge}))$
  $\Rightarrow reference\_mediation(<op, o>_p) = deny)$

- **Process Deny Disjunctive Prohibition:** The quaternary relation P_deny_disjunctive from P to Ops to OAs to OACs.
  $P\_deny\_disjunctive \subseteq P \times Ops \times OAs \times OACs$
  $\forall p \in P, \forall op \in Op, \forall o \in O: ((op, o) \in PCap(p) \land \exists ops \in OPs, \exists oas \in OAs,$
  $\exists oacs \in OACs: ((p, ops, oas, oacs) \in P\_deny\_disjunctive \land$
  $op \in ops \land (\exists oa \in oas: o \in oa^{\blacklozenge} \lor \exists oa \in oacs: o \in oa^{\lozenge}))$
  $\Rightarrow reference\_mediation(<op, o>_p) = deny)$

- **Process Deny Conjunctive Prohibition:** The quaternary relation P_deny_conjunctive from P to Ops to OAs to OACs.
  $P\_deny\_conjunctive \subseteq P \times Ops \times OAs \times OACs$
  $\forall p \in P, \forall op \in Op, \forall o \in O: ((op, o) \in PCap(p) \land \exists ops \in OPs, \exists oas \in OAs,$
  $\exists oacs \in OACs: ((p, ops, oas, oacs) \in P\_deny\_conjunctive \land$
  $op \in ops \land (\forall oa \in oas: o \in oa^{\blacklozenge} \land \forall oa \in oacs: o \in oa^{\lozenge}))$
  $\Rightarrow reference\_mediation(<op, o>_p) = deny)$

- **User Attribute Deny Disjunctive Prohibition:** The quaternary relation UA_deny_disjunctive from UA to Ops to OAs to OACs.
  $UA\_deny\_disjunctive \subseteq UA \times Ops \times OAs \times OACs$

$\forall p\in P, \forall op\in Op, \forall o\in O: ((op, o) \in PCap(p) \land \exists ua\in UA, \exists ops\in OPs, \exists oas\in OAs,$
$\exists oacs\in OACs: ((ua, ops, oas, oacs)\in \text{UA\_deny\_disjunctive} \land$
$\text{process\_user}(p)\rightarrow^{+} ua \land op\in ops \land (\exists oa\in oas: o\in oa^{\blacklozenge} \lor \exists oa\in oacs: o\in oa^{\lozenge}))$
$\Rightarrow \text{reference\_mediation}(<op, o>_p) = \text{deny})$

▪ **User Attribute Deny Conjunctive Prohibition:** The quaternary relation UA_deny_conjunctive from UA to Ops to OAs to OACs.
$\text{UA\_deny\_conjunctive} \subseteq UA\times Ops\times OAs\times OACs$
$\forall p\in P, \forall op\in Op, \forall o\in O: ((op, o) \in PCap(p) \land \exists ua\in UA, \exists ops\in OPs, \exists oas\in OAs,$
$\exists oacs\in OACs: ((ua, ops, oas, oacs)\in \text{UA\_deny\_conjunctive} \land$
$\text{process\_user}(p)\rightarrow^{+} ua \land op\in ops \land (\forall oa\in oas: o\in oa^{\blacklozenge} \land \forall oa\in oacs: o\in oa^{\lozenge}))$
$\Rightarrow \text{reference\_mediation}(<op, o>_p) = \text{deny})$

▪ **Prohibition Determination:** The relation NoDenys from P to Op to O; the tuple (p, op, o) is a member of NoDenys iff no prohibitions exist that affect the authorization.
$\forall p\in P, \forall op\in Op, \forall o\in O: ((p, op, o) \in \text{NoDenys}, \text{ iff}$
$\forall ua\in UA, \forall ops\in Ops, \forall oas\in OAs, \forall oacs\in OACs: \neg(op\in ops \land$
$(((ua, ops, oas, oacs) \in \text{UA\_deny\_disjunctive} \land \text{process\_user}(p)\rightarrow^{+} ua \land$
$(\exists oa\in oas: o\in oa^{\blacklozenge} \lor \exists oa\in oacs: o\in oa^{\lozenge})) \lor$
$((ua, ops, oas, oacs) \in \text{UA\_deny\_conjunctive} \land \text{process\_user}(p)\rightarrow^{+} ua \land$
$(\forall oa\in oas: o\in oa^{\blacklozenge} \land \forall oa\in oas: o\in oa^{\lozenge})) \lor$
$((p, ops, oas, oacs) \in \text{P\_deny\_disjunctive} \land$
$(\exists oa\in oas: o\in oa^{\blacklozenge} \lor \exists oa\in oacs: o\in oa^{\lozenge})) \lor$
$((\text{process\_user}(p), ops, oas, oacs) \in \text{U\_deny\_disjunctive} \land$
$(\exists oa\in oas: o\in oa^{\blacklozenge} \lor \exists oa\in oacs: o\in oa^{\lozenge})) \lor$
$((p, ops, oas, oacs) \in \text{P\_deny\_conjunctive} \land$
$(\forall oa\in oas: o\in oa^{\blacklozenge} \land \forall oa\in oas: o\in oa^{\lozenge})) \lor$
$((\text{process\_user}(p), ops, oas, oacs) \in \text{U\_deny\_conjunctive} \land$
$(\forall oa\in oas: o\in oa^{\blacklozenge} \land \forall oa\in oas: o\in oa^{\lozenge})))))$

▪ **Reference Mediation (with Prohibitions):** The function from domain AReq to codomain {grant, deny}.
$\forall p\in P, \forall op\in Op, \forall o\in O: (\text{reference\_mediation}(<op, o>_p) = \text{grant},$
$\text{iff } (op, o) \in PCap(p) \land (p, op, o) \in \text{Nodenys})$
$\text{otherwise, reference\_mediation}(<op, o>_p) = \text{deny}$

3.5 Obligations

Automatic changes to policy based on specific conditions related to modes and patterns of access can be accomplished through obligations. Events are the means by which obligations are triggered. An event occurs each time a requested access $<op, o>_p$ executes successfully. Information related to the event is called the event context and is used by the PM to process obligations. The process identifier, identifier of the associated user, access operation, and object identifier of the triggering event are always returned as part of the event pattern. Other information conveyed via the event context varies based on the type of event that occurred and may include items such as the containers containing the targeted object.

Two components are required to define an obligation: an event pattern, ep, and a response, resp. An obligation can be expressed in various ways; the following is used in this report:

When ep **do** resp

The event pattern specifies conditions that if matched with an event context, trigger the execution of the response. The event pattern is a logical expression that can use the information returned via the event context, as well as the policy elements and relations in existence when the event occurs, to specify the triggering conditions. The invocation of an administrative command constitutes the response. Arguments passed to administrative commands include items from the event context or derived from evaluation of the event pattern. Administrative commands are capable of adjusting policy through changes to the prevailing policy element relationships and to individual policy elements. Administrative commands are discussed in detail in the next chapter.

The conditions for an event pattern can be extensive. For example, an event pattern may apply to certain operations or any operation; the processes of a specific user or group of users, or any user; one type of object or any object; or all defined policy classes or a specific set of policy classes. EC.*name* denotes the *name* item of an event context. EC.p, EC.u, EC.o, and EC.op refer respectively to the identifiers of the process, user, object accessed, and access operation, which are conveyed in the event context of every event.[2]

The execution of an administrative routine can potentially create one or more events for which other obligations might apply, and whose response in turn could create events that trigger further obligations. The chain of obligation-triggering events could continue until a point at which all obligations are satisfied, or continue indefinitely and result in a livelock situation. Livelocks may also induce resource starvation and potentially create a deadlock situation. Therefore, caution is required when specifying obligations to avoid creating conditions that lead to livelock situations.

An obligation is typically created by an administrative command. The user that issues the command, normally an administrator, must have sufficient authorization not only to create the obligation, but also to perform the body of the response. When the event pattern of a defined obligation is matched, the associated response is carried out automatically under the authorization of the user that created it, regardless how or by whom the event was triggered. An obligation's response can conceivably be involved in a race condition with administrative actions being taken manually, as well as the responses of other concurrently triggered obligations that remain outstanding.

Obligations provide a powerful means to define within a policy specification, specific circumstances associated with an event. An occurrence of those circumstances precipitates automatic changes to policy without intervention from an administrator. While obligations are not represented on an authorization graph, any changes to the policy specification that occur

[2] It may seem redundant to include both the user and process identifiers in the event context, since the process_user function can be used to obtain the user identifier from the process identifier. The rationale for including both is that at the time the event context is being processed, the process that spawned the event may have already terminated, preventing derivation of the user identifier.

because of an obligation are reflected in the authorization graph, with the exception of any newly created obligations.

The set of possible obligations within a policy specification is defined by the ternary relation OBLIG ⊆ U × Pattern × Response. For a tuple of OBLIG, (u, pattern, response), u represents the user responsible for establishing the obligation and under whose authorization the response is carried out. The pattern and response elements each denote a sentence in a grammar that respectively expresses the conditions of an event pattern and the administrative command invocation of the response. That is, the pattern and response elements represent a sequence of symbols whose syntax is well formed according to its respective grammar, and whose execution occurs during the matching process, in the case of a pattern, or after a match occurs, in the case of a response.

Notation for Obligations. The relationships among elements of the PM model involved in obligations can be defined more formally as shown below.

- **Event Context (EC):** The event context of an event associated with a non-administrative access request, which triggers an obligation. EC.*name* denotes the *name* item for the event context of the spawning event.

- **String:** A finite sequence of symbols over some alphabet Σ.

- **Pattern:** A finite set of strings over the alphabet Σ_C, which represents the logical expression of an event pattern's conditions. Pattern denotes a formal language over the alphabet in question. The alphabet and language grammar used to specify event patterns are an implementation choice.

- **Response:** A finite set of strings over the alphabet Σ_R, which represents the invocation of an administrative command that constitutes an event response. Response denotes a formal language over the alphabet in question. The alphabet and language grammar used to specify responses are an implementation choice.

- **Obligations:** The ternary relation OBLIG from U to Pattern to Response.
  OBLIG ⊆ U×Pattern×Response

4. Administrative Considerations

The PM model distinguishes between administrative operations for the creation and maintenance of policy elements and relations and generic input/output operations on resources represented by objects (i.e., non-administrative operations). The previous chapters focused mainly on the modeling of policies involving non-administrative operations. Specifically, the definition for association and the rules for deriving privileges and mediating access apply strictly to input/output operations on objects. The derivation of privileges from associations involving administrative operations, although similar, is distinct and follows a different set of definitions and rules.

Many operations categorized as administrative operations, such as creating a file and assigning it to a folder, are arguably non-administrative from a usage standpoint, but nevertheless, from a policy specification standpoint (i.e., creating an object and assigning an object to object attribute) are considered administrative. The main difference is that non-administrative actions pertain to input/output activities on protected resources, while administrative actions pertain to the manipulation of a policy comprising the policy elements and relationships defined within and maintained by the PM. This chapter explains the principles involved in specifying administrative operations under the PM model. It also discusses the precepts to follow when conducting administrative activities.

4.1 Administrative Associations and Privileges

The term administrative association refers to an association that involves administrative operations exclusively to designate access authority. Administrative associations are distinct from normal, non-administrative associations, as mentioned above. While administrative associations appear on an authorization graph, as do non-administrative associations, administrative associations can apply to any policy element, not just object attributes. Administrative associations are defined by the ternary relation Admin_ASSOC from UA to AOps to PE (i.e., Admin_ASSOC $\subseteq$ UA×AOps×PE), where AOps = $2^{\text{AOp}} - \{\emptyset\}$.

With administrative associations, any referenced policy element takes on special semantics. As the third term of an association, the policy element serves as a referent or representative for the section of the authorization graph rooted at the policy element. That is, a referent policy element serves as a designator for not only itself, but also for policy elements and relationships contained by the referent, which allows the elements of that subgraph to be treated as objects within the PM framework and manipulated accordingly. The following classes of administrative operations apply within the PM model:

- Authority to create or delete a policy element with respect to an existing element of a policy graph
- Authority to create or delete assignments between policy elements
- Authority to form or rescind an association, prohibition, or obligation.

Figure 5 presents a simple example of an authorization graph involving both administrative and non-administrative associations, which builds on the example presented earlier in Figure 4. The left side of the policy graph has been expanded to accommodate a set of administrators for the policy class, which is designated by the Administrators and OUadmin user attributes. A single user, u_4, is assigned as an administrator for the OU. Administrative associations that specify the actions the administrator is able to carry out are illustrated in blue. Non-administrative associations that apply to common users are illustrated in black, using a different type of connector between the elements of these associations, than that used for the administrative associations. This convention for depicting the two different types of associations with a distinctive type of connector is followed throughout the remainder of this report.

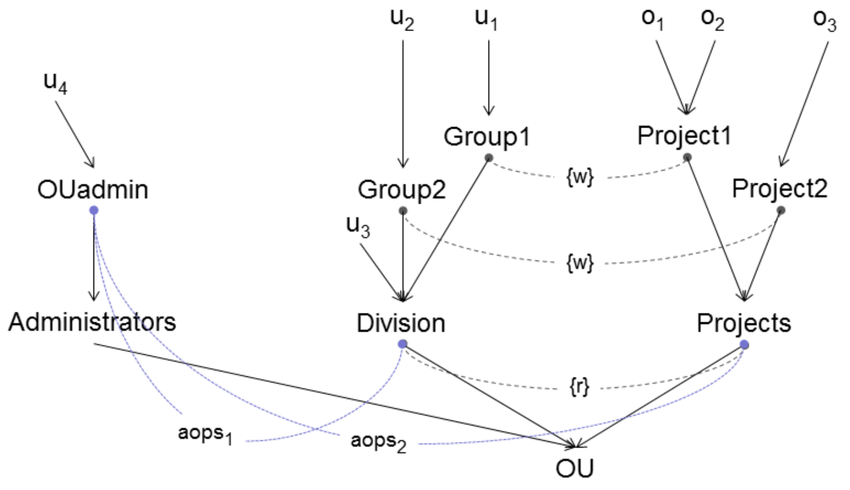

Figure 5: Simple Example Involving Administrative Associations

The authorization graph in Figure 5 contains the following two administrative associations: (OUadmin, $aops_1$, Division) and (OUadmin, $aops_2$, Projects), where $aops_1$ and $aops_2$ each represent a set of administrative operations (i.e., each is a member of AOps). The first association permits the user assigned to OUadmin to create new groups of users and individual users for the OU, to delete existing groups and users, to form new associations for existing and newly created user groups in the OU, and to rescind existing associations involving OU policy elements. The second association permits new groups of projects and individual objects to be created, existing projects and objects to be deleted, new associations to be formed for existing and new projects, and existing associations to be rescinded.

Without administrative associations, a system policy would be very limited. For instance, in this example, new users could not be created or old ones deleted without them, and new objects could not be created; only existing objects could be viewed and modified.

The definition for derived privileges based on administrative associations is similar to that for non-administrative associations. An administrative privilege specifies a relationship between a user, an administrative operation, and a policy element. For a single policy class, the triple (u, aop, pe) is an administrative privilege, iff there exists an administrative association (ua, aops, pe_i), such that user $u \rightarrow^{+} ua$, $aop \in aops$, and $pe \rightarrow^{*} pe_i$.

As mentioned above, each referent, pe_k, represents a policy graph containing all policy elements from which the referent is reachable through one or more assignments, and includes all relationships bound to those elements. Although a referent potentially represents many policy elements and relationships, an administrative operation may apply to only a subset of the policy elements that are represented by the referent. For example, if the referent is a container, an operation might apply to one or more of the policy elements it contains, or the operation might apply only to the container itself—it depends entirely on the operation. In other words, as with non-administrative associations, the properties of a policy element include not only those directly held by the element, but also the properties inherited from every element of the subgraph in which the policy element is contained.

4.2 Administrative Access Requests and Reference Mediation

The access requests and reference mediation are reflected in the PM model differently for administrative actions than for non-administrative actions. Recall that in an access request, $<op, o>_p$, representing a non-administrative action, op designates a single abstract input/output operation on the object, while in a privilege, (u, op, o), op designates an access right that authorizes an unlimited number of abstract input/output operations on an object. For administrative actions, abstract operations are not synonymous with the access rights needed to carry out those operations, and the two aspects require greater delineation. In addition, administrative actions typically involve not just a single object, but multiple policy elements, sets of policy elements, and sets of system operations, which affects the formulation of administrative access requests.

Let AAct represent the set of possible administrative actions and Argseq the set of all finite lists of arguments for administrative actions. An administrative access request AAreq is defined as $<aact, argseq>_p$, where aact∈AAct and argseq∈Argseq. The argument sequence, argseq, is a list of one or more arguments [argseq.1, argseq.2, …, argseq.k], which defines the scope and nature of the action. Each argument can be one of the following items: a distinct policy element, a set of policy elements, a set of non-administrative operations, or a set of administrative operations. That is, an administrative access request comprises an administrative action and a list of enumerated arguments that are dictated by the type of action.

The order of the arguments in an argument sequence for an administrative action is significant, as is the number and type. For instance, the creation of an assignment between two object attributes, $<\text{assign-OAtoOA}, [oa_i, oa_j]>_p$, is completely different from one where the order is reversed, $<\text{assign-OAtoOA}, [oa_j, oa_i]>_p$. Exactly two policy elements are required for assign-OAtoOA: the first corresponding to the tail of the assignment and the second to the head. In contrast, an administrative action to create a read association between a user attribute and an object attribute, $<\text{create-Assoc}, [ua_i, \{r\}, oa_j]>_p$, requires exactly three arguments: a user attribute, a set of operations, and an object attribute.

For an administrative action to be granted, the user on whose behalf the process operates must hold sufficient authority over the policy elements involved, in the form of at least one and possibly more administrative privileges over each of the policy elements involved. Recall that the authority associated with an administrative privilege of the form (u, aop, pe) may apply not only to pe, but also as a referent, to any policy element contained by pe.

To determine the disposition of an access request, <aact, argseq>$_p$, reference mediation requires a mapping from the administrative action and enumerated arguments in question to the set of capabilities that are required over the policy elements referenced in the arguments for the process to carry out the request, barring any prohibitions to the contrary. The mapping Req_ACap(aact, argseq) returns the set of administrative capabilities that are required to carry out the administrative action with the specified arguments. The administrative reference mediation function grants the process p the permission to execute a request <aact, argseq>$_p$, iff process p holds all the capabilities in Req_ACap(aact, argseq).

Notation for Administrative Associations and Privileges. The relationships among elements of the PM model formed through administrative associations and privileges are defined more formally below.

- **AOps:** A finite set of all subsets of administrative operations defined in AOp, excluding the empty set; aops or $aops_1$, $aops_2$, … denote a member of AOps, unless otherwise specified.
  $AOps = 2^{AOp} - \{\emptyset\}$

- **AAct:** A finite set of administrative actions; aact or $aact_1$, $aact_2$, … denote a member of AAct, unless otherwise specified.

- **AActs:** A finite set of all subsets of administrative actions defined in AAct, excluding the empty set; aacts or $aacts_1$, $aacts_2$, … denote a member of AActs, unless otherwise specified.
  $AActs = 2^{AAct} - \{\emptyset\}$

- **Argseq:** A set of all finite lists of the form [arg_1, arg_2,…, arg_n], such that $n \geq 1 \wedge$ for i=1 to n: ($arg_i \in PE \vee arg_i \in PEs \vee arg_i \in OPs \vee arg_i \in AOPs$); argseq denotes a member of Argseq, and argseq.1, argseq.2, … denote the respective element in the list argseq (e.g., argseq.2 is the second element in argseq), unless otherwise specified.

- **Administrative Associations:** The ternary relation Admin_ASSOC from UA to AOps to PE.
  $Admin\_ASSOC \subseteq UA \times AOps \times PE$

- **Inherent Administrative Capabilities:** The partial function IACap from UA to $2^{(AOps \times PE)}$.
  - $IACap \subseteq UA \times 2^{(AOps \times PE)}$
  - $\forall ua \in UA, \forall aops \in AOps, \forall pe \in PE$: ((aops, pe) ∈ IACap(ua), iff (ua, aops, pe) ∈ Admin_ASSOC)

- **Inherent Administrative Access Entries:** The partial function IAAE from PE to $2^{(UA \times AOps)}$.
  - $IAAE \subseteq PE \times 2^{(UA \times AOps)}$
  - $\forall ua \in UA, \forall aops \in AOps, \forall pe \in PE$: ((ua, aops) ∈ IAAE(pe), Iff (ua, aops, pe) ∈ Admin_ASSOC)

- **Administrative Privileges:** The ternary relation Admin_PRIV from U to AOp to PE.
  - $Admin\_PRIV \subseteq U \times AOp \times PE$

- $\forall u \in U, \forall aop \in AOp, \forall pe \in PE$: ((u, aop, pe) $\in$ Admin_PRIV, iff $\exists ua \in UA, \exists aops \in AOps, \exists pe_i \in PE$: ((ua, aops, pe_i) $\in$ Admin_ASSOC $\wedge$ $u \rightarrow^{+} ua$ $\wedge$ $aop \in aops$ $\wedge$ $pe \rightarrow^{*} pe_i$))

- **Administrative Access Entries:** The function AAE from PE to $2^{(U \times AOp)}$.
  - $AE \subseteq PE \times 2^{(U \times AOp)}$
  - $\forall u \in U, \forall aop \in AOp, \forall pe \in PE$: ((u, aop) $\in$ AAE(pe), iff (u, aop, pe) $\in$ Admin_PRIV)

- **Administrative Capabilities:** The function ACap from U to $2^{(AOp \times PE)}$.
  - $ACap \subseteq U \times 2^{(AOp \times PE)}$
  - $\forall u \in U, \forall aop \in AOp, \forall pe \in PE$: ((aop, pe) $\in$ ACap(u), iff (u, aop, pe) $\in$ Admin_PRIV)

- **Administrative Process Capabilities:** The function APCap from P to $2^{(AOp \times PE)}$.
  $APCap \subseteq P \times 2^{(AOp \times PE)}$
  $\forall p \in P, \forall aop \in AOp, \forall pe \in PE$: ((aop, pe) $\in$ APCap(p),
  iff (process_user(p), aop, pe) $\in$ Admin_PRIV)

- **Administrative Access Request:** A finite set AAReq of possible process access requests.
  - $AAReq \subseteq P \times (AAct \times Argseq)$
  - (p, (aact, argseq)) $\in$ AAReq $\overset{def}{=}$ $<aact, argseq>_p$

- **Required Administrative Capabilities:** The partial binary function ReqACap from AAct $\times$ Argseq to $2^{(AOp \times PE)}$, such that $\forall aact \in AAct, \forall argseq \in Argseq$: ((aop,pe) $\in$ ReqACap(aact, argseq), iff (aop,pe) is a requisite authority needed to perform the action aact on argseq).
  $ReqACap \subseteq (AAct \times Argseq) \times 2^{(AOp \times PE)}$

- **Reference Mediation of Administrative Actions:** The function from domain AAReq to codomain {grant, deny}.
  $\forall p \in P, \forall aact \in AAct, \forall argseq \in Argseq$:
  (Admin_reference_mediation($<aact, argseq>_p$) = grant,
  iff $\forall aop \in AOp, \forall pe \in PE$: ((aop, pe) $\in$ ReqACap(aact, argseq) $\wedge$ (aop, pe) $\in$ APCap(p)));
  otherwise, Admin_reference_mediation($<aact, argseq>_p$) = deny

4.3 Administrative Prohibitions and Obligations

Recall that prohibitions act antithetically to privileges, denoting an effective set of restrictions on privileges for a specific user or process, regardless of whether any of the privileges designated actually can or cannot be derived for the user or process in question. Because the set of privileges for administrative operations is distinct from the set of privileges for non-administrative privileges, administrative prohibitions (i.e., prohibitions on administrative privileges) must be defined accordingly.

Notational conventions similar to those for the object range of a policy element, $pe^{\blacklozenge}$, and the complementary object range, $pe^{\lozenge}$, can be defined for administrative prohibitions to accommodate the treatment of policy elements as referents. Let $pe^{\bullet}$ denote the set of all policy elements that can reach element pe (i.e., $pe^{\bullet} = \{x: x \in PE \text{ and } x \rightarrow^{*} pe\}$). The complement of $pe^{\bullet}$ with respect to the set of all policy elements, PE, is denoted by pe° (i.e., $pe^{\circ} = PE - pe^{\bullet}$) and represents those policy elements for which pe is not reachable. The notation $pe^{\bullet}$ and pe° are respectively called the element range and complementary element range of a policy element.

The quaternary relation U_Admin_deny_disjunctive $\subseteq$ U × AOps × PEs × PECs, where PEs = PECs = 2^{PE}, defines the set of user deny prohibitions for a policy specification, which involve administrative operations. The tuple, U_Admin_deny_disjunctive(u, aops, pes, pecs) ∈, where u ∈ U, aops ∈ AOps, pes ∈ PEs, pecs ∈ PECs, and pes ∪ pecs ≠ ∅, denotes that any process p executing on behalf of user u is withheld the authority in aops over any policy element that can reach one of the elements in pes, or cannot reach one of the policy elements in pecs. Similarly, an individual tuple (u, aops, pes, pecs) of the quaternary relation U_Admin_deny_conjunctive $\subseteq$ U × AOps × PEs × PECs, denotes that a process p executing on behalf of user u is withheld the authority in aops over any policy element that can reach all of the policy elements in pes and also cannot reach any of the policy elements in pecs.

The definitions for process-based prohibitions that are needed to accommodate administrative operations are similar to those for user-based prohibitions. The prohibition P_Admin_deny_disjunctive $\subseteq$ P × AOps × PEs × PECs represents a process-based administrative deny relation, where p ∈ P, aops ∈ AOps, pes ∈ PEs, pecs ∈ PECs, and (p, aops, pes) ∈ P_Admin_deny_disjunctive. The meaning of P_Admin_deny_disjunctive(p, aops, pes, pecs) is that the process p is withheld the authority in aops over any policy element that can reach one of the policy elements in pes, or cannot reach one of the policy elements in pecs. The complementary relation for process-based administrative prohibitions also exists. The meaning of P_Admin_deny_conjunctive(p, aops, pes, pecs) is that the process p is withheld the authority in aops over any policy element that can reach all of the policy elements in pes and also cannot reach any of the policy elements in pecs.

User attribute-based prohibitions also apply to administrative operations. The quaternary relation UA_Admin_deny_disjunctive $\subseteq$ UA × AOps × PEs × PECs, defines the set of user deny prohibitions. The tuple UA_Admin_deny_disjunctive(ua, aops, pes, pecs) denotes that any process p, executing on behalf of some user u that is contained in ua, is withheld the authority in aops over any policy element that can reach one of the elements in pes, or cannot reach one of the policy elements in pecs. Similarly, an individual tuple (ua, aops, pes, pecs) of the quaternary relation UA_Admin_deny_conjunctive $\subseteq$ UA × AOps × PEs × PECs, denotes that a process p, executing on behalf of some user u that is contained in ua, is withheld the authority in aops over any policy element that can reach all of the policy elements in pes and also cannot reach any of the policy elements in pecs.

Administrative prohibitions take precedence over any existing administrative associations and the privileges derived from those associations. An access request to a subgraph of an authorization graph is granted a user or process acting on behalf of the user, iff the appropriate administrative associations are defined that allow such access, and there is not an administrative prohibition for that user or process on the requested object, which countermands the administrative access operation in question. If such a prohibition does exist, access is denied. That is, when administrative prohibitions apply, reference mediation grants the process p permission to execute an access request $<aact, argseq>_p$, iff for u = process_user(p), the following conditions hold:

- The administrative privilege (u, aop, pe) exists, for all capabilities (aop, pe) in Req_ACap(aact, argseq);

- There do not exist administrative prohibitions (p, aops, pes, pecs) $\in$ P_Admin_deny_disjunctive or (u, aops, pes, pecs) $\in$ U_Admin_deny_disjunctive, such that aop $\in$ aops, and for some member x of pes, pe $\in x^{\bullet}$, or for some member x of pecs, pe $\in x^{\circ}$;

- There do not exist administrative prohibitions (p, aops, pes, pecs) $\in$ P_Admin_deny_conjunctive or (u, aops, pes, pecs) $\in$ U_Admin_deny_conjunctive, such that aop $\in$ aops, and for all members x of pes, pe $\in x^{\bullet}$, and for all members x of pecs, pe $\in x^{\circ}$.

- There do not exist administrative prohibitions (ua, aops, pes, pecs) $\in$ UA_Admin_deny_disjunctive, such that aop $\in$ aops, u $\rightarrow^{+}$ ua, and for some member x of pes, pe $\in x^{\bullet}$, or for some member x of pecs, pe $\in x^{\circ}$;

- There do not exist administrative prohibitions (ua, aops, pes, pecs) $\in$ UA_Admin_deny_conjunctive, such that aop $\in$ aops, u $\rightarrow^{+}$ ua, and for all members x of pes, pe $\in x^{\bullet}$, and for all members x of pecs, pe $\in x^{\circ}$.

Obligations for administrative access requests are distinct from, but similar to those for non-administrative access requests. Like a non-administrative obligation, an administrative obligation consists of an event pattern and a response, and the response is triggered by events that match the event pattern. The invocation of an administrative command also constitutes the response for an administrative obligation. However, administrative obligations are triggered only by administrative events, which occur each time an administrative access request $<\text{aact, argseq}>_p$ executes successfully.

The event context for administrative events, therefore, is distinct from that for non-administrative events and accordingly must convey different items, namely AEC.p, AEC.u, AEC.aact, and AEC.argseq, where AEC represents the administrative event context, and the suffixes p, u, aact, and argseq represent respectively the process, user, administrative action, and argument sequence elements of the event context. An event pattern and response elements of an administrative obligation use items of the administrative event context to specify the conditions that trigger the execution of the response and to serve as arguments for the response.

The set of possible administrative obligations within a policy specification is defined by the ternary relation Admin_OBLIG $\subseteq$ U $\times$ Pattern $\times$ Response. The tuple (u, pattern, response) $\in$ Admin_OBLIG denotes that the user u is responsible for establishing the obligation consisting of the event pattern and event response elements, pattern and response. It is under the authorization of u that the response is carried out, when the pattern matches an event. The same grammars used to express non-administrative obligations are also presumed to be used to express administrative obligations.

Notation for Administrative Prohibitions and Obligations. The relationships among elements of the PM model affected by administrative prohibitions are defined more formally below.

▪ **PEs:** A finite set of all subsets of policy elements defined in PE; pes or pes_1, pes_2, … denote a member of PEs, unless otherwise specified.
$PEs = 2^{PE}$

▪ **PECs:** A finite set of all subsets of policy elements defined in PE; pecs or $pecs_1$, $pecs_2$, … denote a member of PECs, unless otherwise specified.
$PECs = 2^{PE}$

▪ **Element Range of a Policy Element:** The set of policy elements that can reach a given policy element through 0 or more assignments.
$\forall pe \in PE\text{: } pe^{\bullet} \overset{def}{=} \{x\text{: } x \in PE \wedge x \rightarrow^{*} pe\}$.

▪ **Complementary Element Range of a Policy Element:** The set of policy elements that cannot reach a given policy element.
$\forall pe \in PE\text{: } pe^{\circ} \overset{def}{=} PE - pe^{\bullet}$.

▪ **User Administrative Deny Disjunctive Prohibition:** The quaternary relation U_Admin_deny_disjunctive from U to AOps to PEs to PECs.

$U\_Admin\_deny\_disjunctive \subseteq U \times AOps \times PEs \times PECs$
$\forall p \in P, \forall aact \in AAct, \forall argseq \in Argseq$:
$(\exists aop \in AOp, \exists pe \in PE\text{: } ((aop, pe) \in ReqACap(aact, argseq) \wedge$
$\exists aops \in AOps, \exists pes \in PEs, \exists pecs \in PECs$:
$((process\_user(p), aops, pes, pecs) \in U\_Admin\_deny\_disjunctive \wedge$
$aop \in aops \wedge (\exists x \in pes\text{: } pe \in x^{\bullet} \vee \exists y \in pecs\text{: } pe \in y^{\circ}))) \Rightarrow$
$Admin\_reference\_mediation(<aact, argseq>_p) = deny)$

▪ **User Administrative Deny Conjunctive Prohibition:** The quaternary relation U_Admin_deny_conjunctive from U to AOps to PEs to PECs.

$U\_Admin\_deny\_conjunctive \subseteq U \times AOps \times PEs \times PECs$
$\forall p \in P, \forall aact \in AAct, \forall argseq \in Argseq$:
$(\exists aop \in AOp, \exists pe \in PE\text{: } ((aop, pe) \in ReqACap(aact, argseq) \wedge$
$\exists aops \in AOps, \exists pes \in PEs, \exists pecs \in PECs$:
$((process\_user(p), aops, pes, pecs) \in U\_Admin\_deny\_conjunctive \wedge$
$aop \in aops \wedge (\forall x \in pes\text{: } pe \in x^{\bullet} \wedge \forall y \in pecs\text{: } pe \in y^{\circ}))) \Rightarrow$
$Admin\_reference\_mediation(<aact, argseq>_p) = deny)$

▪ **User Attribute Administrative Deny Disjunctive Prohibition:** The quaternary relation UA_Admin_deny_disjunctive from UA to AOps to PEs to PECs.

$UA\_Admin\_deny\_disjunctive \subseteq UA \times AOps \times PEs \times PECs$
$\forall p \in P, \forall aact \in AAct, \forall argseq \in Argseq$:
$(\exists aop \in AOp, \exists pe \in PE\text{: } ((aop, pe) \in ReqACap(aact, argseq) \wedge$
$\exists ua \in UA, \exists aops \in AOps, \exists pes \in PEs, \exists pecs \in PECs$:
$((ua, aops, pes, pecs) \in UA\_Admin\_deny\_disjunctive \wedge process\_user(p) \rightarrow^{+} ua \wedge$
$aop \in aops \wedge (\exists x \in pes\text{: } pe \in x^{\bullet} \vee \exists y \in pecs\text{: } pe \in y^{\circ}))) \Rightarrow$
$Admin\_reference\_mediation(<aact, argseq>_p) = deny)$

▪ **User Attribute Administrative Deny Conjunctive Prohibition:** The quaternary relation UA_Admin_deny_conjunctive from UA to AOps to PEs to PECs.

$UA\_Admin\_deny\_conjunctive \subseteq UA \times AOps \times PEs \times PECs$

$\forall p \in P, \forall aact \in AAct, \forall argseq \in Argseq$:
$(\exists aop \in AOp, \exists pe \in PE$: $((aop, pe) \in ReqACap(aact, argseq) \wedge$
$\exists ua \in UA, \exists aops \in AOps, \exists pes \in PEs, \exists pecs \in PECs$:
$((ua, aops, pes, pecs) \in UA\_Admin\_deny\_conjunctive \wedge process\_user(p) \rightarrow^{+} ua \wedge$
$aop \in aops \wedge (\forall x \in pes: pe \in x^{\bullet} \wedge \forall y \in pecs: pe \in y^{\circ}))) \Rightarrow$
$Admin\_reference\_mediation(<aact, argseq>_p) = deny)$

- **Process Administrative Deny Disjunctive Prohibition:** The quaternary relation P_Admin_deny_disjunctive from P to AOps to PEs to PECs.
$P\_Admin\_deny\_disjunctive \subseteq P \times AOps \times PEs \times PECs$
$\forall p \in P, \forall aact \in AAct, \forall argseq \in Argseq$:
$(\exists aop \in AOp, \exists pe \in PE$: $((aop, pe) \in ReqACap(aact, argseq) \wedge$
$\exists aops \in AOps, \exists pes \in PEs, \exists pecs \in PECs$:
$((p, aops, pes, pecs) \in P\_Admin\_deny\_disjunctive \wedge aop \in aops \wedge$
$(\exists x \in pes: pe \in x^{\bullet} \vee \exists y \in pecs: pe \in y^{\circ}))) \Rightarrow$
$Admin\_reference\_mediation(<aact, argseq>_p) = deny)$

- **Process Administrative Deny Conjunctive Prohibition:** The quaternary relation P_Admin_deny_conjunctive from P to AOps to PEs to PECs.
$P\_Admin\_deny\_conjunctive \subseteq P \times AOps \times PEs \times PECs$
$\forall p \in P, \forall aact \in AAct, \forall argseq \in Argseq$:
$(\exists aop \in AOp, \exists pe \in PE$: $((aop, pe) \in ReqACap(aact, argseq) \wedge$
$\exists aops \in AOps, \exists pes \in PEs, \exists pecs \in PECs$:
$((p, aops, pes, pecs) \in P\_Admin\_deny\_conjunctive \wedge aop \in aops \wedge$
$(\forall x \in pes: pe \in x^{\bullet} \wedge \forall y \in pecs: pe \in y^{\circ}))) \Rightarrow$
$Admin\_reference\_mediation(<aact, argseq>_p) = deny)$

- **Administrative Prohibition Determination:** The relation NoDeny from P to AOp to PE. The triple (p, aop, pe) is a member of NoDeny, iff no prohibitions exist that affect the authorization aop on the policy element pe for the process p.
$\forall p \in P, \forall aop \in AOp, \forall pe \in PE$: $((p, aop, pe) \in NoDeny$, iff
$\forall ua \in UA, \forall aops \in AOps, \forall pes \in PEs, \forall pecs \in PECs$: $\neg(aop \in aops \wedge$
$(((ua, aops, pes, pecs) \in UA\_Admin\_deny\_disjunctive \wedge$
$process\_user(p) \rightarrow^{+} ua \wedge (\exists x \in pes: pe \in x^{\bullet} \vee \exists y \in pecs: pe \in y^{\circ})) \vee$
$((ua, aops, pes, pecs) \in UA\_Admin\_deny\_conjunctive \wedge$
$process\_user(p) \rightarrow^{+} ua \wedge (\forall x \in pes: pe \in x^{\bullet} \wedge \forall y \in pecs: pe \in y^{\circ})) \vee$
$((p, aops, pes, pecs) \in P\_Admin\_deny\_disjunctive \wedge$
$(\exists x \in pes: pe \in x^{\bullet} \vee \exists y \in pecs: pe \in y^{\circ})) \vee$
$((process\_user(p), aops, pes, pecs) \in U\_Admin\_deny\_disjunctive \wedge$
$(\exists x \in pes: pe \in x^{\bullet} \vee \exists y \in pecs: pe \in y^{\circ})) \vee$
$((p, aops, pes, pecs) \in P\_Admin\_deny\_conjunctive \wedge$
$(\forall x \in pes: pe \in x^{\bullet} \wedge \forall y \in pecs: pe \in y^{\circ})) \vee$
$((process\_user(p), aops, pes, pecs) \in U\_Admin\_deny\_conjunctive \wedge$
$(\forall x \in pes: pe \in x^{\bullet} \wedge \forall y \in pecs: pe \in y^{\circ})))))$

- **Reference Mediation of Administrative Actions (with Prohibitions):** The function from domain AAReq to codomain {grant, deny}.
$\forall p \in P, \forall aact \in AAct, \forall argseq \in Argseq$:
$(Admin\_reference\_mediation(<aact, argseq>_p) = grant$,
iff $\forall aop \in AOp, \forall pe \in PE$: $((aop, pe) \in ReqACap(aact, argseq) \wedge (aop, pe) \in APCap(p) \wedge$
$(p, aop, pe) \in NoDeny))$;
otherwise, $Admin\_reference\_mediation(<aact, argseq>_p) = deny$

▪ **Administrative Event Context (AEC):** The event context for an event associated with an administrative access request, which triggers an obligation. AEC.*name* denotes the *name* item for the event context of the spawning administrative event.

▪ **Administrative Obligations:** The ternary relation Admin_OBLIG from U to Pattern to Response.

Admin_OBLIG ⊆ U×Pattern×Response

4.4 Administrative Commands and Routines

Administrative commands and routines are the means by which policy specifications are formed. Their structure and use are discussed in detail below. The core administrative commands and routines for the PM, along with a description of their semantics, are presented in Appendix C.

4.4.1 Administrative Routines

Administrative routines describe rudimentary operations that can occur on the policy elements and relationships of the PM model. An administrative routine is represented as a parameterized procedure, whose body describes state changes to policy that occur when the routine is executed (e.g., a policy element or relation Y changes state to Y′ when some function f is applied). Administrative routines are specified using the following format:

```
Rtnname ($x_1$, $x_2$, …, $x_k$)
  preconditions: …
    {
    $Y' = f(Y, x_1, x_2, …, x_k)$
    }
  postconditions:
```

The name of the administrative routine, Rtnname, precedes its formal parameters, x_1, x_2, …, x_k ($k \geq 0$). A set of preconditions preface the body of the routine and a set of postconditions follow the body. Preconditions and post conditions are logical expressions that must be respectively satisfied when the routine is invoked and when the routine has completed. No state changes described in the body occur unless the preconditions are satisfied. Preconditions for administrative routines are used to ensure that the arguments supplied to the routine are valid and that policy elements and relationships are maintained consistently with the properties of the model. Postconditions identify alterations to policy that are expected to take place during the execution of the routine.

Consider, as an example, the administrative routine CreateAssoc shown below, which specifies the creation of an association. The preconditions bind x, y, and z parameters respectively to the user attribute, operation set, and object attribute elements of the model. The body describes the addition of the tuple (x, y, z) to the ASSOC relation, which changes the state of the relation to ASSOC′. Finally, the postconditions assert that the tuple (x, y, z) is expected to be a member of ASSOC′ after execution of CreateAssoc. If tuple (x, y, z) was a member of ASSOC to begin with, the relation is unchanged (i.e., ASSOC′ = ASSOC).

```
CreateAssoc (x, y, z)
   preconditions: $x \in UA \land y \in Ops \land z \in OA$
       {
       $ASSOC' = ASSOC \cup \{(x, y, z)\}$
       }
   postconditions: $(x, y, z) \in ASSOC'$
```

Compared to administrative commands, which are discussed in the next section, administrative routines are more primitive. That is, each administrative routine entails a modification to the policy configuration that typically involves either the creation or deletion of a policy element, the creation or deletion of an assignment between policy elements, or the creation or deletion of an association, prohibition, or obligation. Administrative routines provide the foundation for the PM framework and must perform their intended function correctly and without unwanted side effects. Access to these security-critical routines must be restricted.

4.4.2 Administrative Commands

An administrative command consists mainly of a parameterized interface and a sequence of administrative routine invocations. Administrative commands build upon administrative routines to define the protection capabilities of the PM model. The body of an administrative command is executed as an atomic transaction—an error or lack of capabilities that causes any of the constituent routines to fail execution causes the entire command to fail, producing the same effect as though none of the routines were ever executed. Administrative commands are specified using the following format:

```
Cmdname ($x_1, x_2, …, x_k$)
   preconditions: …
       {
       $rtn_1$
       $rtn_2$
       . . .
       $rtn_n$
       }
   postconditions: …
```

The name of the administrative command, Cmdname, precedes the command's formal parameters, $x_1, x_2, …, x_k$ ($k \geq 0$). Each formal parameter of an administrative command can serve as an argument in any of the administrative routine invocations, $rtn_1, rtn_2, …, rtn_n$ ($n \geq 0$), which make up the body of the command. As with administrative routines, the body is bracketed by pre- and postconditions. The preconditions ensure, in general, that the arguments supplied to the command are valid, that the process requesting the execution of the command has sufficient authorization to execute all constituent administrative routines, and that certain properties of the model upon which the command relies prevail.

Administrative commands are used in a variety of ways. Figure 6 gives an overview of the types of usage possible.

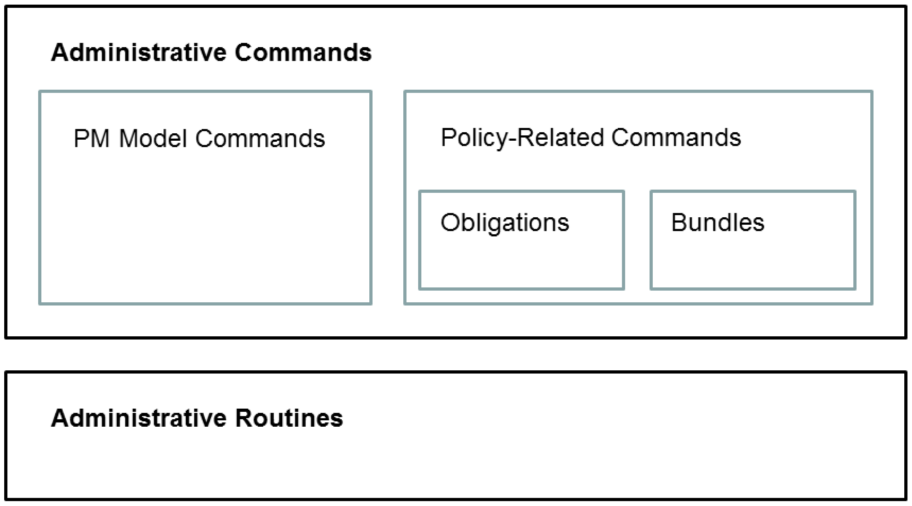

Figure 6: Administrative Commands and Routines

First and foremost, administrative commands are used to define the protection features and services of the PM model. The semantic description of those commands is given in Appendix C. Every administrative access request corresponds to an administrative command of the PM model on a one-to-one basis.

Another common use of administrative commands is in the definition of obligations, as the response to be taken whenever the corresponding event pattern is matched. It is important to note that administrative commands used to define system policy through an obligation response are distinct from those that define the protection features and services of the PM model and are used to fulfill administrative access requests. Although commands defined for use in obligations may carry out the same or similar functions to those of the PM model, they are invoked differently and the authorization requirements for each are also different.[3] Another way of looking at the situation is that PM model commands are incompatible and not usable with obligations. The most common types of administrative commands defined for use in setting policy via obligations involve the creation of assignments or prohibitions. Examples of them are given later in the report.

Administrative commands can also be used to facilitate the administration of system policies. For example, when a new user is created, an administrator typically creates a number of containers, links them together, and grants the authority for the user to access them as its work space. Rather than manually performing each step of this sequence of administrative actions for each new user, the entire sequence of repeated actions can be defined as a single administrative command and executed in its entirety as an atomic action.

[3] The preconditions of administrative commands defined for use in obligations require that the user who defined the obligation holds sufficient authorization to execute all constituent administrative routines of the body, while the preconditions of administrative commands for the PM model require that the process attempting the access holds sufficient authorization to execute all constituent administrative routines of the body.

Taking this idea of bundling a step further, it is possible to combine a lengthy extended sequence of administrative actions together into a single administrative command that is capable of building an entire system policy. This type of bundling would allow an established policy to be instantiated quickly elsewhere, and also allow command libraries containing various kinds of vetted policies or policy enhancements to be assembled and shared on a broad scale.

4.4.3 Administrative Actions

Within the PM framework, several basic precepts govern the actions that a user with administrative authority can take when using administrative commands and routines to specify policy. They are as follows:

- **Initial Conditions.** In the initial state of the PM framework, certain users, designated as administrators, may already hold authority over policy elements pre-established by the framework, via one or more associations. Policy classes serve as the foundation of subsequent policy specification activities.

- **Element Additions.** At the moment when a user A creates a policy element B, it obtains a reference to the newly created B, which it can use in conjunction with other existing policy elements to build up a specification. The ability to create certain policy elements may be reserved exclusively for particular users or administrators.

- **Relationship Changes.** When user A successfully creates a policy element B, it may then assign B to an existing compatible policy element for which it holds authority and thereby gain additional authority over B through the inheritance of properties. The authority that A holds over B and other policy elements may in turn allow A to define additional relationships among them or to delete exist relationships.

- **Element Deletions.** Any user A that holds sufficient administrative authority over a policy element B can delete the policy element. However, existing relationships involving B must be taken into account and addressed before deleting B.

- **Automation.** A user with sufficient administrative authority may define obligations that are used carry out a set of predefined activities on behalf of the user, based on the occurrence of specific types of events.

As discussed in later chapters, administrative actions are usually conducted through a graphical user interface that renders the authorization graph of a policy for an administrator to facilitate modifications.

5. Policy Specification

This chapter provides comprehensive details about formulating policy specifications, including levels of administrative authorities and policy, considerations for specifications involving multiple policy classes, and the use of primitive and consolidated operations. Examples are given to illustrate the key concepts outlined in the discussion.

5.1 Model Aspects and Use

From the material in previous chapters, it is evident that there are many facets to the PM model. They include the policy elements and assignments that make up a policy element diagram, the associations and prohibitions that apply to the policy element diagram to form the authorization graph, and obligations that are carried out when access-related events occur. Note that to compose a specific policy for the PM, each and every one of these items may not be required. For example, a policy may involve at a minimum only a simple policy element diagram with several associations. On the other hand, capturing a specific policy may require the use of all facets of the PM model.

A couple of more detailed examples are provided below to illustrate how aspects of the model can be brought together to define a specific access control policy. They involve a data service for electronic mail and an operating system. The examples are purposely limited in the range of functions provided to avoid extensive policy definitions. Nevertheless, the examples should provide a good foundation for the material in the remainder of this report.

A specific policy can be correctly expressed in numerous ways, depending on the preferences of the administrator specifying the policy and the conventions followed. The examples in this section should be interpreted as a general guideline to follow when developing policy specifications, and not as a mandatory approach to follow.

5.1.1 Electronic Mail

Electronic mail is a commonly used data service that needs little introduction. For the simple electronic mail system in question, the objects involved are messages, and the object attributes include an inbox, outbox, draft folder, and trash folder for each user. Each user of the system is able to read and delete messages in its inbox and to create, read, write, and delete messages in its outbox, draft folder, and trash folder. Each user can also write a copy of a message in its outbox to the inbox of any other user.

The policy administrator first must create a policy class for the mail system. It then can create the necessary containers and settings to organize the mail system and to manage users and establish their containers for messages. As an organizing step, the policy administrator creates the user attribute, Users, and the object attribute, Objects, and assigns them to the Mail System policy class. It also creates the object attributes Inboxes and Outboxes as system-wide containers to retain each user's inbox and outbox respectively, and assigns both Inboxes and Outboxes to Objects. Figure 7 illustrates the policy element diagram constructed so far.

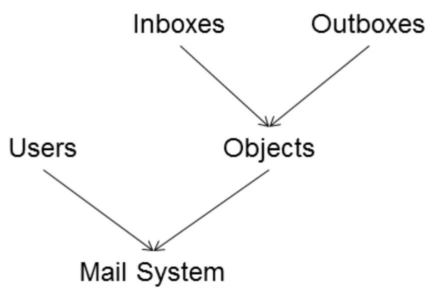

Figure 7: Partial Policy Element Diagram for the Mail System

For each new user, u_i, the administrator creates an associated user attribute, ID u_i, which is needed in forming associations that involve the user, and assigns it to Users. The following object attributes are also created for each new user: In u_i, Other u_i, Out u_i, Draft u_i, and Trash u_i. ID u_i is assigned to the Users container, In u_i, is assigned to the Inboxes container, and Out u_i and Other u_i are assigned to the Objects container. The remaining containers, Draft u_i and Trash u_i, are assigned to Other u_i.

Figure 8 below illustrates the policy element diagram constructed, along with the needed associations and prohibitions (i.e., the authorization graph) for a typical user, u_2, to conduct generic input/output operations on mail objects (i.e., messages) in its containers. No mail objects are shown in the figure. The user u_2 can read objects in its inbox, In u_2, and can read and write objects within its outbox, Out u_i, because of the associations between ID u_2 and those containers. The user can read and write objects within its own draft and trash folders (i.e., the Draft u_i and Trash u_i containers), which are contained in Other u_2, via the association between ID u_2 and Other u_2. The user can also write to the objects in the inbox of any user, which are by design assigned to Inboxes, but not to its own inbox, due to the write prohibition illustrated with a different style and orientation of connector (i.e., the dotted, upward-arcing connector). No mail objects can be created, however, without further authorizations, nor can any mail objects, if they existed, be deleted.

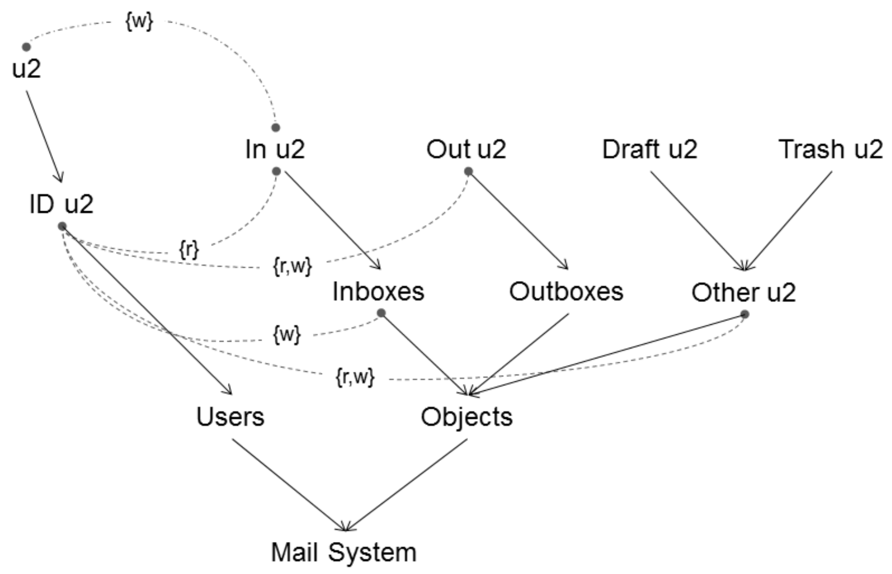

Figure 8: Authorization Graph for the Mail System

The administrative associations and prohibitions specified for the user complement those shown above and define the remaining actions for the user. The rationale behind the associations and prohibitions that are illustrated in the authorization graph in Figure 9 is as follows:

- The association between ID u_2 and Inboxes allows user u_2 to create messages in any user's inbox, including its own. Since the inbox of every mail system user gets assigned to Inboxes when the user is established, and the properties over Inboxes are inherited, the association essentially grants a system-wide authority.

- The prohibition between u_2 and In u_2 serves as a counter to the system-wide authority granted to every user through the previous association. Prohibitions are illustrated similarly to associations, but with a different style and orientation of connector. This prohibition denies the user from creating messages within its own inbox, slightly overriding the system-wide authority to create messages in any inbox.

- The association between ID u_2 and In u_2 allows the user delete messages from its own inbox.

- To create and delete messages within the containers Out u_i, Draft u_i, and Trash u_i, an association granting such authorization is needed between ID u_2 and each of those containers.

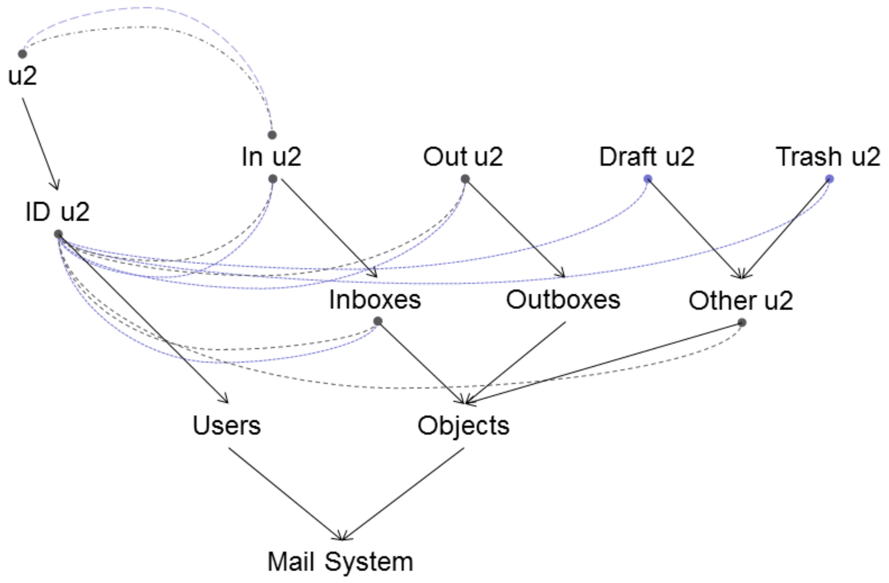

Figure 9: Authorization Graph with Administrative Associations and Prohibitions

A few improvements can be made to the current policy specification. For example, a user can update a copy of a sent message residing in its outbox, which can bring about an unwanted inconsistency from what was actually sent. To avoid this situation, the policy can be revised via an obligation that prevents alterations to messages in the outbox, once they are written to it. The obligation presumes that draft messages are composed in the sender's drafts folder, and then, when ready to be sent, copied over in their entirety to newly created message objects in the sender's outbox, before being deleted. The following obligation accomplishes the write-once restriction through the creation of a user prohibition:

> **When** EC.op = w $\wedge$ EC.o $\rightarrow^{+}$ Outboxes **do**
> CreateDisjunctiveU-Prohibition (EC.u, {w}, {EC.o}, ∅)[4,5]

Similarly, the current policy allows a user to update messages that it has posted to another user's inbox or messages that other users have posted. The policy can be extended slightly with an obligation to prevent any alterations to a message after it is initially written to an inbox. The obligation presumes that when a message composed in the drafts folder is sent, a new message is created in the receiver's inbox and the contents of the draft message copied over in its entirety to the new message. The following obligation created for each user u accomplishes the write-once restriction through the creation of a user attribute prohibition:

> **When** EC.op = w $\wedge$ EC.o $\rightarrow^{+}$ Inboxes **do**
> CreateDisjunctiveUA-Prohibition (Users, {w}, {EC.o}, ∅)[6]

The policy defined for the users of the mail system is discretionary. While the policy grants no authority for a user to create administrative associations for other users to access objects under its control, it grants a user authority to perform certain mail system activities that allow that information to be shared. For example, one user cannot allow another user to read messages residing in its inbox, but it can create a copy of the message and send it to another user.

5.1.2 Operating System

As mentioned previously, most present-day operating systems use DAC as their primary access control mechanism. For the simple DAC operating system in question, the objects involved are files and folders. The latter also serves as an object attribute or container for files and other folders. Each user of the system has a home container and is able to read, write, create, and delete folders and files contained within its home container. Each user can also grant other users the privileges to read and write any file contained within its home container.

The policy administrator first creates a policy class, DAC, for the DAC operating system. It then creates the user attribute, Users, and the object attribute, Objects, and assigns them to the DAC policy class. For each new user, u_i, the administrator creates an associated user attribute, ID u_i, and assigns it to Users. The user's home container, Home u_i, is also created and assigned to the

[4] Because the sets involved in the prohibition are a singleton and an empty set, a conjunctive deny involving these same sets would have same effect as the disjunctive deny used.

[5] The semantics of the administrative command used in this obligation is essentially the same as that for the command CreateDisjunctiveUserProhibition given in Appendix C, with one exception—the preconditions for this command asserts that the user who defined the obligation must hold sufficient authorization to execute the body of the command.

[6] As with the administrative command in the previous obligation, the CreateDisjunctive UserAttributeProhibition command given in Appendix C has essentially the same semantics, with the caveat of differences in preconditions.

Objects container. Figure 10 illustrates the policy element diagram constructed, along with the appropriate associations, for two typical users, u_2 and u_3.

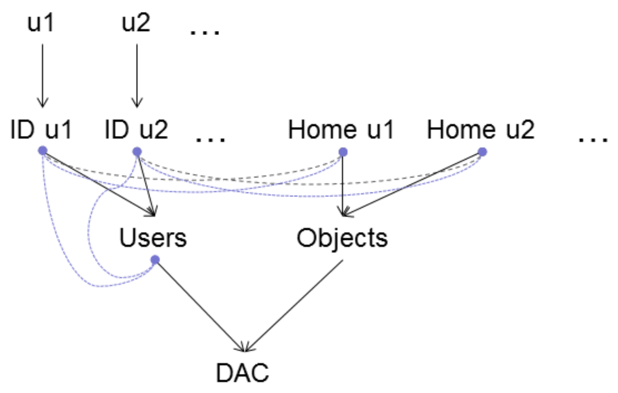

Figure 10: Authorization Graph for the DAC Operating System

No prohibitions are needed in this example. Two associations are needed for each user and are summarized in terms of u_1, as follows:

- The association between ID u_1 and Home u_1 allows the user to read, write, and execute files that are contained within its home container. The administrative association between those same attributes allows the user, u_1, to create, and delete files and other containers (i.e., folders) within its home container. It also allows the user to form or rescind an association with other user policy elements, involving the read, write, create, and delete privileges it holds over its home container, Home u_1, and through inheritance, to any objects that are contained by the home container.

- The administrative association between ID u_1 and Users allows the user to involve any user contained in Users (i.e., all users) in the formation of a new association. This privilege combined with the previous administrative association mentioned enables a user to grant the privileges it holds over objects in its home container selectively to any other user, or to rescind them. That is, u_1 has the discretion to grant the authority to read, write, execute, create, and delete files and folders within its home container to other users and subsequently, to take back that authority.

A slight expansion of this example can better illustrate the properties of the authorization graph that allows users to form new associations that affect the contents of their home container. User u_1 has created two files, o_{11} and o_{12}, in its home container, and would like u_2 to be able to read and update o_{11}. Using its discretionary authority, u_1 forms a new association between ID u_2 and o_{11}, illustrated in Figure 11, which allows u_2 the ability to read and write the contents of o_{11}. Although u_2 gains the ability to read and write o_{11}, it cannot pass that ability on to other users, since u_1 did not grant u_2 the ability to form or rescind an association with o_{11}.

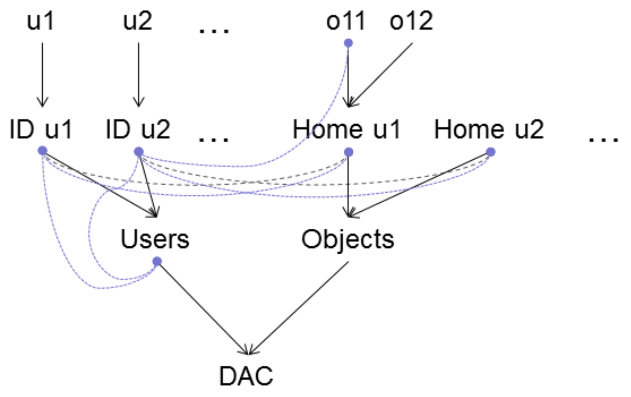

Figure 11: DAC Authorization Graph with Objects

As plainly evident, the policy defined for the users of the operating system is discretionary. The policy grants a user not only the authority to perform common operating system activities, such as creating or deleting files and folders, but also authority for a user to form and rescind associations that allow other users access to files and folders under its home container.

5.2 Levels of Policy and Administration

The PM model supports the definition of a single level or multiple levels of administrative authority. Three main types of authorities exist. They are as follows:

- The Principal Authority (PA), also known as the super user, is a compulsory, predefined entity of the PM. The PA is responsible for creating and controlling the policies of the PM in their entirety and inherently holds full access privileges to carry out those activities. The PA generally creates policy classes and first-level attributes that define an authorization administrator and a domain for the authorization administrator to manage, then allocates sufficient privileges to the authorization administrator to perform those duties. Multiple domains and administrators can be created by the PM, at its discretion. The PA can also forego the use of an authorization administrator and manage a domain itself.

- The Domain Administrator (DA) can create users, objects, and attributes within its domain and manage the entire domain itself. A DA can also define a sub-domain of its domain and allocate sufficient privileges for a subdomain administrator to manage it. The domain administered by a DA may be divided into more than one subdomain. However, the DA must possess sufficient privileges allocated by the PA to be able to define a subdomain and allocate the needed privileges to a subdomain administrator.

- The Subdomain Administrator (SA) can perform the activities of a domain administrator within the sub-domain under its control. The pattern of subdividing the sub-domain for Sub-SAs (S^2A), Sub-S^2As (S^3A), and so forth to manage can continue as needed.

Various usage patterns of authority levels can be used when specifying policy. The two main types of patterns discussed here are intra-policy class and inter-policy class patterns. It is important when specifying policy to keep in mind the underlying principle that once the PA creates the requisite domains and domain administrators and establishes the policy for a system, the system should move safely from state to state in accordance with that policy. That is, for given policy configuration and an initial starting state, it should not be possible to reach a state in which a particular access authorization is acquired by policy entity for which it is not intended.

5.2.1 Intra-Policy Class Patterns

An intra-policy class pattern denotes an arrangement of authority levels in which each authority and its according domain of control is contained within a single policy class. Figure 12 below illustrates one such pattern. Consistent with existing conventions, the solid-line arrows represent assignments between policy elements. The dotted-line connectors indicate administrative associations that allow an authorization administrator, represented by a user attribute, to preside over a domain, represented by user and object attributes.

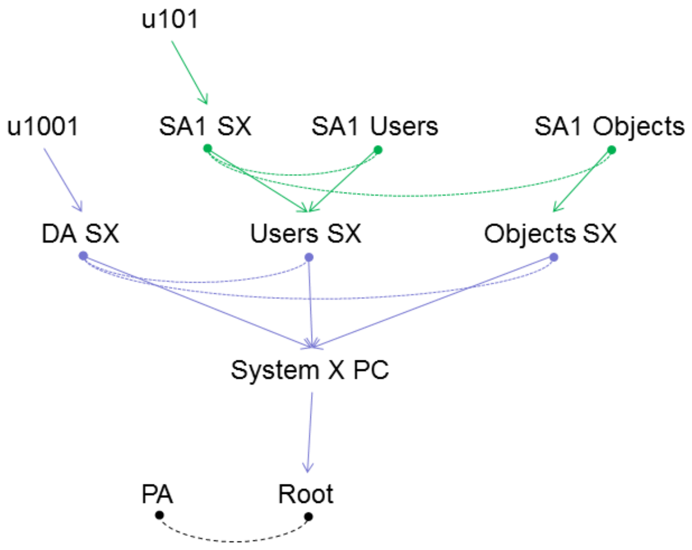

Figure 12: Pattern of Authority Levels

The assignment and association connectors are colored to convey which authority established the policy. PA's control over the PM root is depicted in black and conveys its de facto authority over all aspects of policy. The PA establishes a policy class for System X (SX), creates the user attribute for the DA (DA SX), and the Users and Objects attributes of SX, and assigns them to the policy class. The PA then creates the associations needed by a DA (i.e., a user assigned to DA), such that a DA has sufficient privileges to administer the users and objects of that domain. Finally, the PA creates a user, u_{1001}, and assigns it to DA to preside over that domain. The assignments and associations carried out by the PA are colored blue.

User u_{1001}, in its capacity as a DA, can create subsets of its domain for SAs to manage. Figure 12 illustrates the user attribute for the first SA (SA1) of SX, and the user and object attributes that comprise the subdomain, SA1 Users and SA1 Objects, along with the requisite assignments and associations made by the DA. The DA's assignments and association are colored green. The assigned SA, u_{101}, can then carry onward from this point populating the subdomain with any users, objects, and attributes that apply.

5.2.2 Inter-Policy Class Patterns

An intra-policy class pattern denotes an arrangement of authority levels in which each authority is contained within a policy class that is distinct from the policy class for its domain of control. An example inter-policy class pattern of authority levels is shown in Figure 13 below. Instead of maintaining DAs and SAs within the same policy class as the users and objects of their domain, a distinct Admin policy class is established by the PA for those authorities and is assigned to the PM root. This pattern requires the PA to create the user attribute, SX Admin, as a placeholder for the SX authorities and assign the attribute to Admin. A benefit of this pattern is that it allows the PA to create and manage authorities for other systems under the Admin policy class (e.g., by adding the SY Admin attribute for System Y) and to establish their domain of control over the applicable policy classes that represent those systems.

From this point the actions are similar to the previous pattern. The PA creates the user attribute for the DA, DA SX, and assigns it to SX Admin. It also establishes a policy class for System X (SX) and the users and objects attributes of SX (i.e., Users SX and Objects SX), and assigns them to the SX policy class. The PA then creates the associations needed by a DA, such that the DA has sufficient privileges to administer the users and objects of that domain, which entails privileges that span the two policy classes (i.e., Admin and System X PC), as well as privileges to manage itself via SX Admin. Finally, the PA creates a user, u_{1001}, and assigns it to DA SX to preside over that domain. As before, the assignments and associations carried out by the PA are colored blue, while those of the DA are colored green.

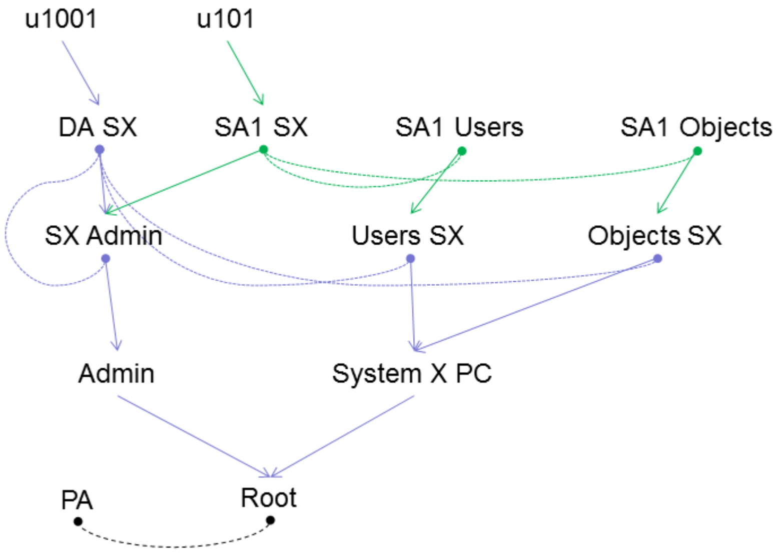

Figure 13: An Alternative Pattern of Authority Levels

User u_{1001}, in its capacity as a DA, can create subsets of its domain for SAs to manage. The main difference from the previous pattern is that the user attribute for the first SA (SA1) of SX is assigned to SX Admin. The DA creates the attributes for the users and objects that comprise the subdomain, SA1 Users and SA1 Objects, and makes the requisite assignments and associations for the SA1 administrator u_{101} to govern the subdomain. Note that if the DA wanted to grant SA1 the authority to create administrative subdomains of SA1 following this pattern, it would need to do the following: create an additional attribute (e.g., SA1 Admin), assign SA1 Admin to

SX Admin, assign SA1 SX to SA1 Admin instead of SX Admin, and allocate sufficient privilege for SA1 SX to govern SA1 Admin and to create administrative attributes for the next level of SAs.

One distinction between this pattern and the previous one is that as a side effect of this pattern, the DA has the authority to assign other users as DAs. This specific authority could be removed via a prohibition, if strict compliance with the policy of the previous pattern is needed. The opposite is also possible. While the previous pattern prescribes that the PA makes all DA user assignments, an association from DA SX to DA SX could be added to that specification to grant the DA this authority, if such a capability is needed.

A simple change to the above inter-policy class pattern can make it into an intra-policy class pattern. All that it takes is deleting the Admin policy class and assigning SX Admin directly to System X PC. Note too that it is possible to mix intra and inter-policy class patterns. That is, some authorities can be maintained in a distinct policy class external to the one being administered, while other authorities are maintained within the policy class being administered.

5.2.3 Personas and Patterns

In many situations, an administrator of a system is also potentially a user of that same system. Under the PM model, disregarding the principle of least privilege and assigning an individual the capabilities of both a system user and a system administrator through the same policy element can lead to security issues. One solution is to allow the individual to login under either of two distinct user policy elements (e.g., u_i and u_j), each representing a different persona. Having different personas to carry out different activities is a long-standing practice for attaining least privilege (e.g., [Sal75]). However, the fact that one individual can operate as two different users is retained outside of the policy specification and, because it is not expressed explicitly therein, easy to overlook or ignore, leading to problems. For example, when such an individual leaves the organization, only one of the two user elements may be deleted, allowing the individual continued system access through the remaining user element.

It is possible to express explicitly within the PM model an individual's ability to act in different capacities selectively at different times. Accommodating personas within the model is an advanced topic that builds upon the material covered in this chaper and Appendix C. Appendix D provides a detailed discussion of two alternative approaches. For simplicity, the examples and discussion in the main body of the report presume that individuals assigned as system administrators are not also assigned as users of the system.

5.3 Authority Level Examples

To illustrate authority levels and policy better, the examples of section 3.6 are reexamined in light of the above discussion.

5.3.1 Electronic Mail

To set up the initial authorization policy for a DA to administer the mail system described earlier, the PA creates a framework using the intra-policy class pattern of Figure 12. Figure 14 below illustrates the policy element diagram and associations the PA establishes for the DA. The

DA, Users, and Objects attributes shown constitute the key attributes of the Mail System policy class. The user, u_{1001}, is assigned as the DA. In this example, no SA is required. Instead, the DA is expected to manage the entire mail system. More than one user can be assigned as a DA, and a DA has sufficient authority to carve out subdomains, if eventually needed.

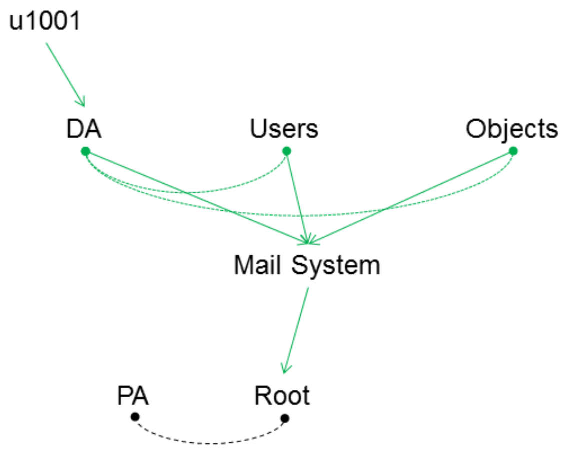

Figure 14: Policy Assignments and Associations for the Mail System DA

The DA serves as the policy authority responsible for creating the necessary containers to organize the mail system and for managing other users and establishing their containers for messages. Figure 15 illustrates the authorization graph for the mail system, highlighting a typical user, u_2. The administrative assignments, associations and prohibitions made by the DA are in blue to distinguish them from those made by the PA, which are in green.

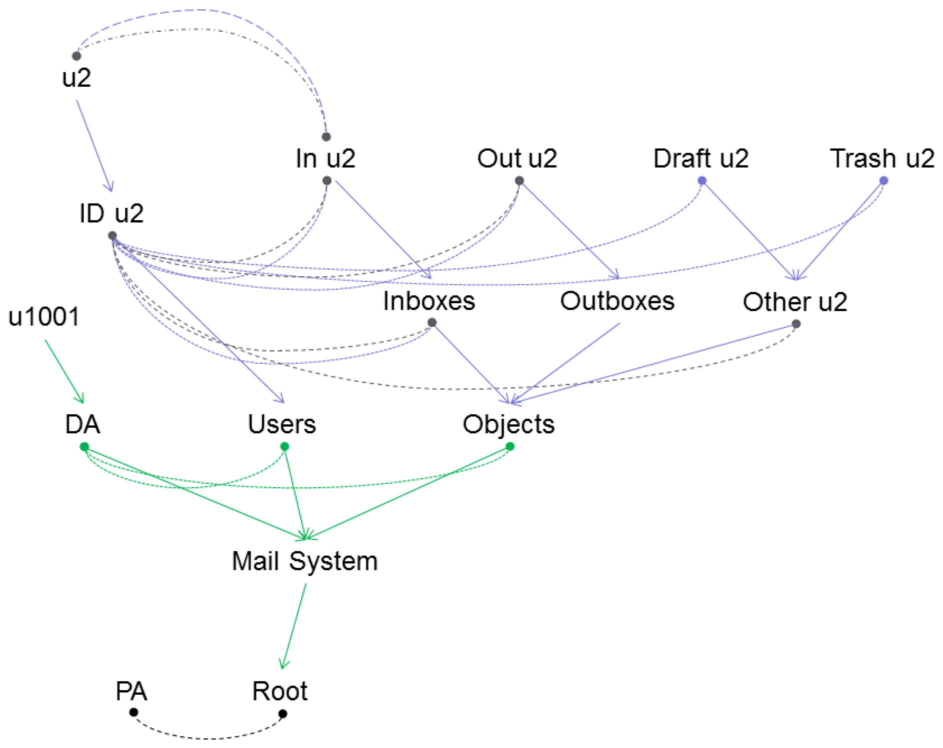

Figure 15: Authorization Graph for the Mail System

The PA has full discretionary authority over the PM, and grants discretionary authority to the DA over the policy domain it establishes for the mail system. This allows the DA to create the necessary attributes, associations, and other relationships for each user to use the functionality of the mail system. The DA also has the discretion to create subdomains and assign an SA to them. The policy for this system with regard to users is also discretionary, since each user has the freedom to use the established policy to share information with other users.

5.3.2 Operating System

The actions the PA takes to set up the initial authorization policy for a DA to administer the DAC operating system described earlier mirror those given above for the mail system. Figure 16 below illustrates the policy element diagram and administrative associations the PA establishes for the DA using the intra-policy class pattern. The DA, Users, and Objects attributes constitute the key attributes of the DAC policy class. The user, u_{1011}, is assigned as the DA. This example again requires no SA and instead relies on the DA to manage the entire system.

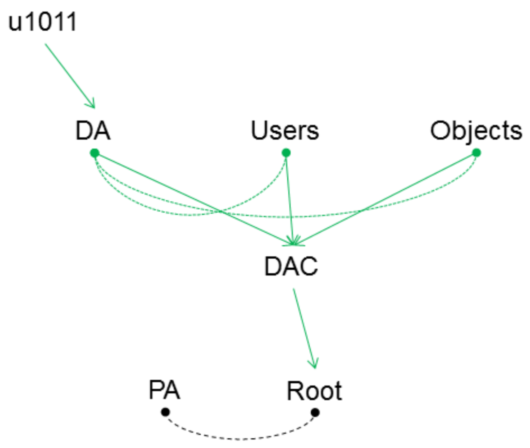

Figure 16: Policy Elements and Associations for the DAC DA

Figure 17 illustrates the authorization graph for the DAC operating system, showing two typical users, u_1 and u_2. The assignments, associations and prohibitions made by the DA are colored blue. The defined policy is discretionary at the PA, DA, and user levels. The PA has full discretionary authority over the PM. The PA in turn, grants discretionary authority to the DA over the policy domain it establishes for the operating system, which allows the DA to create the necessary attributes and associations between attributes for each user. The DA grants administrative authority for any user to create associations that allow other users to selectively access objects contained in its home container. The DA also has the discretion to create subdomains and assign an SA to them, if it chooses.

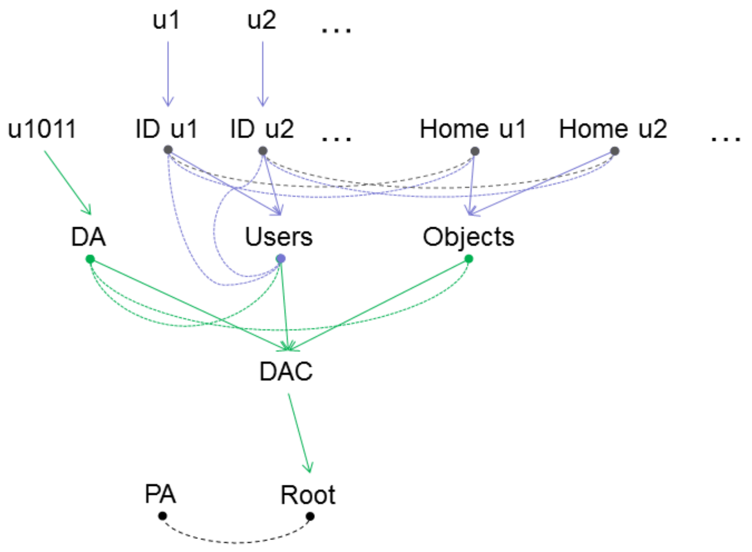

Figure 17: Authorization Graph for the DAC Operating System

5.4 Generic Operations

Generic operations refer to the authority needed to carry out a related mode of access or action. The examples given in this and the previous chapters allude to a variety of operations needed to structure policy for a system and establish levels of administration for governing the policy over its lifetime. These generic operations include authority to read and write objects, to create and destroy various policy elements, and to form and rescind various types of relationships between policy elements. This section discusses a core set of operations for the PM model in detail and explains how such authority can be utilized to specify access control policy.

Two general classes of generic operations exist. They are non-administrative operations that pertain to protected resources represented by objects, and administrative operations that pertain to a policy specification comprising the policy elements and relationships defined within and maintained by the PM. The first class of generic operations, as shown in Table 1, falls into one type of access mode: input and output of data to and from protected resources represented by objects, designated respectively as read and write operations. Protected resources may be logical (e.g., files and folders) or physical (e.g., printers and networking components). As mentioned previously, for non-administrative operations, abstract operations are synonymous with the authority needed to carry out those operations: to output data or write to an object requires write authority, and to input data or read from an object requires read authority.

Table 1: Generic Non-administrative Access Operations

Type	Non-administrative Operation	Applies to	Affects
Input/Output Resources	read	Object attribute	Protected resource represented by the object attribute or an object contained by the object attribute
	write	Object attribute	Protected resource represented by the object attribute or an object contained by the object attribute

The second class of generic operations relate to one of four types of administrative actions: the creation and deletion of policy elements, the creation and deletion of assignments between policy elements that are contained within the same policy class, the creation and deletion of assignments between policy elements that are each contained within a different policy class, and the creation and deletion of associations, prohibitions, and obligations among policy elements. Administrative operations convey the authority to manipulate policy elements and relations maintained by the PM, and thereby institute or update the policy specification for a system. However, unlike non-administrative operations, the authority associated with an administrative operation is not necessarily synonymous with an administrative action. Instead, the authority stemming from one or more administrative operations may be required for a single action to be authorized.

Table 2 below lists the generic administrative operations for each type of access modality, following the naming conventions for policy elements established in the previous chapters. In both Tables 1 and 2, the "Applies to" column identifies the type of policy element to which the mode of access relates, while the "Affects" column identifies the policy element or relation affected.

Table 2: Generic Administrative Access Operations by Type

Type	Administrative Operation	Applies to	Affects
Create/Delete Policy Elements	create-u, delete-u	User attribute	User policy element
	create-ua, delete-ua	User attribute, Policy class	User attribute policy element
	create-o, delete-o	Object attribute	Object policy element
	create-oa, delete-oa	Object attribute, Policy class	Object attribute policy element
	create-pc, delete-pc	The conceptual root policy node	Policy class policy element
Create/Delete Assignments (Intra-Policy Class)	create-uua, delete-uua	User attribute	Assignment from a user to the user attribute
	create-uaua, delete-uaua	User attribute	Assignment from a user attribute to the user attribute
	create-uapc, delete-uapc	Policy class	Assignment from a user attribute to the policy class
	create-ooa, delete-ooa	Object attribute	Assignment from an object to the object attribute
	create-oaoa, delete-oaoa	Object attribute	Assignment from an object attribute to the object attribute
	create-oapc, delete-oapc	Policy class	Assignment from an object attribute to the policy class

Type	Administrative Operation	Applies to	Affects
Create/Delete Assignments (Inter-Policy Class)	create-uua-from, delete-uua-from	User attribute	Assignment from a user element in the referent's subgraph to a user attribute
	create-uua-to, delete-uua-to	User attribute	Assignment from a user to a user attribute element in the referent's subgraph
	create-uaua-from, delete-uaua-from	User attribute	Assignment from a user attribute element in the referent's subgraph to a user attribute
	create-uaua-to, delete-uaua-to	User attribute	Assignment from a user attribute to a user attribute element in the referent's subgraph
	create-uapc-from, delete-uapc-from	User attribute	Assignment from a user attribute element in the referent's subgraph to the policy class
	create-uapc-to, delete-uapc-to	Policy class	Assignment from a user attribute to the policy class
	create-ooa-from, delete-ooa-from	Object attribute	Assignment from an object element in the referent's subgraph to an object attribute
	create-ooa-to, delete-ooa-to	Object attribute	Assignment from an object attribute to an object attribute element in the referent's subgraph
	create-oaoa-from, delete-oaoa-from	Object attribute	Assignment from an object attribute in the referent's subgraph to an object attribute element
	create-oaoa-to, delete-oaoa-to	Object attribute	Assignment from an object attribute element to an object attribute in the referent's subgraph
	create-oapc-from, delete-oapc-from	Object attribute	Assignment from an object attribute element in the referent's subgraph to a policy class
	create-oapc-to, delete-oapc-to	Policy class	Assignment from an object attribute to the policy class

Type	Administrative Operation	Applies to	Affects
Create/Delete Multi-way Relationships	create-assoc-from, delete-assoc-from	User attribute	Association involving a user attribute element in the user attribute's subgraph and an object attribute
	create-assoc-to, delete-assoc-to	Object attribute	Association involving a user attribute and an object attribute element in the object attribute's subgraph
	create-admin-assoc-from, delete-admin-assoc-from	User attribute	Administrative association involving a user attribute element in the user attribute's subgraph and a policy element
	create-admin-assoc-to, delete-admin-assoc-to	Policy element	Administrative association involving a user attribute and a policy element in the policy element's subgraph
	create-deny-from, delete-deny-from	User, User attribute	Prohibition involving the user, a process operating for the user, or a user element in the referent user attribute's subgraph, and an object
	create-deny-to, delete-deny-to	Set of Object attributes	Prohibition involving a user or process, and an object element in or outside the referent's subgraph
	create-admin-deny-from, delete-admin-deny-from	User, User attribute	Administrative prohibition involving the user, a process operating for the user, or a user element in the referent user attribute's subgraph, and a policy element
	create-admin-deny-to, delete-admin-deny-to	Set of Policy elements	Administrative prohibition involving a user or process, and a policy element in or outside the referent's subgraph
	create-oblig, delete-oblig	Policy element	Obligation on an access request involving the user or process holding authorization
	create-admin-oblig, delete-admin-oblig	Object attribute	Obligation on an administrative access request involving the user or processing holding authorization

The prefix "create-" denotes the reification of a policy element or a relationship between policy elements, as designated by its stem. Unlike read and write operations, two or more administrative operations are typically needed to carry out some overall meaningful action on the policy representation. In addition administrative operations are typically allocated in conjunction with other. For example, the authority to create a user (create-u) is not useful, if the authority to assign it to a user attribute (create-uua) is not also held. Similarly, the authority to create a policy element (create-u) is not useful, if the authority to delete a policy element (delete-u) is not also held.

Some administrative operations are explicitly divided into two parts, as denoted by the "from" and "to" suffixes. Both parts of the authority must be held to carry out the implied administrative action. A case in point is the ability to form associations between policy elements from different policy classes. A user must hold create-assoc-from authority over a user attribute in one and create-assoc-to authority over an object attribute in the other to form an association between them or other attributes that are in the subgraph of each. The correspondence between administrative operations and administrative activities is evidenced in the preconditions given for each of the administrative commands listed in Appendix C.

While the list of generic operations in Tables 1 and 2 is complete, it is not intended to be absolute. The control objectives of a policy may differ from one system to another and need to be realized using a different set of operations to capture and designate the appropriate authority. For example, a requirement for higher level of assurance may dictate more granular operations, or a requirement for compliance with some prevailing law, regulation, or policy may necessitate additional operations. The modes of access allowed also depend on the types of resources represented and on the functionality of the system

The computational environment may also influence the set of operations defined. Take, for instance, the write operation. If compatible with the computational environment, this operation could be refined or augmented with a write-append variant that allows a user to add additional data to an object, but does not allow a user to change the previous contents of or view an object [NCSC87]. Allowing data to be added only at the beginning or the end of an object would provide more control for maintaining audit information. Similarly, the read operation, which includes the ability to execute an object, could be revised and an explicit execute operation defined for this purpose to allow more granular control, if compatible with the computational environment.

One other consideration that can influence the set of operations is usability. While granular operations allow a fine degree of control, the sheer number can create difficulties when assigning authority within a significantly sized policy specification. One solution is to consolidate multiple operations that are usually assigned together (e.g., create- and delete- operations) into distinct sets and use those sets in lieu of individual operations to assign a broad range of authority collectively when defining policy.

In summary, operations are abstractions of the levels of authorization possible within a computational environment to support a given policy and, as abstractions, may be adjusted to fit a unique situation. In practice, however, it is often the case that only some subset of the authorizations listed is needed to specify the policy for a particular system. For example, the policy specified for the DAC operating system did not require the use of prohibitions or obligations, thus related authorizations, such as create-deny-to/from or create-oblig, were not required.

6. Multiple Policy Class Considerations

Some policy specifications, such as the inter-policy class pattern for expressing levels of policy and administration discussed in the previous chapter, can involve more than one policy class. Multiple policy class situations may arise when two or more policies, each represented by a single policy class, are merged together and overlap to the extent that objects fall under each policy. They can also occur when an administrator chooses to express a single policy using multiple policy classes, even though the policy could be easily expressed with a single policy class.

The basic PM framework discussed so far largely ignores policy specifications that involve multiple policy classes. In order to handle these situations correctly, some slight adjustments to the PM framework are needed. These adjustments specifically involve refining the way privileges are derived for objects that are contained in two or more policy classes and the way prohibitions are applied when two or more policy classes contain objects involved in the prohibition. This chapter looks at the necessary refinements to the PM framework and provides examples of policy specifications that involve multiple policy classes.

6.1 Association Refinements

Non-administrative associations are defined as a relation of the form ASSOC $\subseteq$ UA×Ops×OA. ASSOC is a set of ordered triples. Deriving privileges from a triple (ua, ops, oa) $\in$ ASSOC involves identifying all users that are contained by the first element of the triple, ua, all members of the set ops, the second element, and all objects that are contained by the last element, oa. Each combination of the three resultant sets forms a valid privilege of the form (u, op, o).

A major difference when deriving privileges from associations in specifications that involve multiple policy classes is that the policy classes containing the object attribute play a major role in privilege derivation. In multiple policy class situations, the triple (u, op, o) is a PM privilege, iff for each policy class pc_l that contains o, there exists an association (ua_i, ops_j, oa_k), such that user $u \rightarrow^+ ua_i$, op $\in ops_j$, $o \rightarrow^* oa_k$, and $oa_k \rightarrow^+ pc_l$. That is, a privilege involving an object is valid, iff it can be derived with respect to each of the policy classes that contain the object. This method of derivation works equally well when only a single policy class prevails, since all objects are contained by the sole policy class.

Privileges can also be derived from the user's or object's perspective, by involving inherent capabilities and inherent access entries respectively. That is, a triple (u, op, o) is a privilege, iff, for each policy class pc_l that contains o, there exists a user attribute ua_i with an assigned or inherited inherent capability (ops_j, oa_k), such that $o \rightarrow^* oa_k$, $oa_k \rightarrow^+ pc_l$, $u \rightarrow ua_i$, and op $\in ops_j$. Similarly, the triple (u, op, o) is a privilege, iff for each policy class pc_l that contains o, there exists an object attribute oa_k with an assigned or inherited inherent access entry (ua_i, ops_j), such that $o \rightarrow^* oa_k$, $oa_k \rightarrow^+ pc_l$, $u \rightarrow^+ ua_i$, and op $\in ops_j$.

Administrative privileges are derived similarly. Administrative associations are defined as a relation of the form Admin_ASSOC $\subseteq$ UA×AOps×PE. If multiple policy classes are involved, the triple (u, aop, pe) is an administrative privilege, iff for each policy class pc_l that contains pe, there exists an administrative association (ua_i, $aops_j$, pe_k), such that user $u \rightarrow^{+} ua_i$, $aop \in aops_j$, $pe \rightarrow^{*} pe_k$, and $pe_k \rightarrow^{+} pc_l$.

6.2 Prohibition Refinements

As discussed earlier, non-administrative prohibitions are used to override access to objects based on whether the objects are contained within or not contained within a set of object attributes. Prohibitions in multiple policy class situations, like associations, also follow the precept that an object and the policy classes containing the object have relevance when determining the scope of a prohibition. The scope of a non-administrative prohibition can be defined as the set of object attributes affected by the relation. Wherever the scope of a prohibition overlaps with that of a policy class, the prohibition affects some, but not necessarily all of the same objects within the policy class, which can potentially lead to difficulties in the specification and interpretation of policies involving multiple policy classes.

Recall that the set of objects affected by a disjunctive user deny, (u, ops, oas, oacs) $\in$ U_deny_disjunctive, where oas $\cup$ oacs $\neq \emptyset$, is the union of $oa_i^{\blacklozenge}$ and $oac_j^{\lozenge}$, for all oa_i in oas and all oac_j in oacs (i.e., the set $(oa_1^{\blacklozenge} \cup oa_2^{\blacklozenge} \ldots \cup oa_n^{\blacklozenge}) \cup (oac_1^{\lozenge} \cup oac_2^{\lozenge} \ldots \cup oac_m^{\lozenge})$). Similarly, for a conjunctive user deny, (u, ops, oas, oacs) $\in$ U_deny_conjunctive, the set of objects affected is the intersection of $oa_i^{\blacklozenge}$ and $oac_j^{\lozenge}$, for all oa_i in oas and all oac_j in oacs (i.e., the set $(oa_1^{\blacklozenge} \cap oa_2^{\blacklozenge} \ldots \cap oa_n^{\blacklozenge}) \cap (oac_1^{\lozenge} \cap oac_2^{\lozenge} \ldots \cap oac_m^{\lozenge})$). The reasoning about the scope of user-based prohibitions applies as well to disjunctive and conjunctive process-based prohibitions.

In both the conjunctive and disjunctive classes of user-based prohibitions, any member of the set oas that is contained by a policy class does not present a problem when multiple policy classes apply, since for each object attribute in oas, the scope of the prohibition is always a subset of the scope of any policy class that contains the object attribute and affects the same set of objects. Therefore, the existing definitions for prohibitions apply, without issue, in multiple policy class situations where oacs is equal to the empty set and oas is not. However, when the reverse is true, and oas is equal to the empty set and oacs is not, issues arise in the context of multiple policy classes. The reason is that for each object attribute in oacs, although the scope of the prohibition is a subset of the scope of any single policy class that contains the object attribute, the objects affected are a vastly different set of objects that fall outside the policy class into one or more other policy classes.

It is possible to redefine non-administrative prohibitions to restrict their scope solely to the scope of the policy classes in which they appear. However, that same effect can be realized through other means, such as constraining the prohibition to a specific policy class through the use of an attribute contained by the policy class in the definition of the prohibition. Moreover, in some cases, the broader unconstrained scope of a prohibition may match the target policy more closely and produce the desired effect. For these reasons, no redefinition of non-administrative prohibitions is considered necessary at this time. However, caution is advised when defining prohibitions involving exclusory object attributes.

Because of the similarity in their structure, the same considerations apply to administrative prohibitions as those discussed for non-administrative prohibitions. That is, the current definition of administrative prohibitions is deemed sufficient at this time to express access control policies accurately and no redefinition is necessary.

6.3 Obligation Refinements

The case of obligations is somewhat different from either that for associations or prohibitions. Obligations are unaffected by multiple policy class considerations. The main reason policy classes do not need special consideration is that the scope of control of an obligation is stipulated by its event pattern, which is capable of defining explicitly whether one or more policy classes pertain to the obligation. The PM reference mediation function plays no role in the processing of an obligation's event pattern. Therefore, the existing definition and treatment of obligations remains valid and requires no adjustment.

Revised Notation for Multiple Policy Classes (MPC). Privilege derivation for multiple policy class situations can be defined more formally as shown below.

- **Privileges (revised for MPC):** The ternary relation PRIV from U to Op to O.
  - $PRIV \subseteq U \times Op \times O$
  - $\forall u \in U, \forall op \in Op, \forall o \in O$: $((u, op, o) \in PRIV$, iff $\forall pc \in PC: (o \rightarrow^{+} pc \Rightarrow$ $\exists ops \in Ops, \exists ua \in UA, \exists oa \in OA: ((ua, ops, oa) \in ASSOC \land$ $u \rightarrow^{+} ua \land op \in ops \land o \rightarrow^{*} oa \land oa \rightarrow^{+} pc)))$

- **Administrative Privileges (revised for MPC):** The ternary relation Admin_PRIV from U to AOp to PE.
  - $Admin\_PRIV \subseteq U \times AOp \times PE$
  - $\forall u \in U, \forall aop \in AOp, \forall pe \in PE$: $((u, op, pe) \in Admin\_PRIV$, iff $\forall pc \in PC: (pe \rightarrow^{*} pc \Rightarrow$ $\exists aops \in AOps, \exists ua \in UA, \exists pe_i \in PE: ((ua, aops, pe_i) \in Admin\_ASSOC \land$ $u \rightarrow^{+} ua \land aop \in aops \land pe \rightarrow^{*} pe_i \land pe_i \rightarrow^{*} pc)))$

6.4 Amalgamated Policy Examples

Combining the access control policies of two or more systems can be done in a number of ways. The resulting policy should make sense from a security standpoint and maintain the intended security objectives asserted originally by each system individually. Ideally, the resulting policy should also gain efficiency in operation of the administrative levels. For example, rather than having redundant user policy elements to represent a user separately under each system policy, only one set of policy elements could be maintained and applied to both.

The amalgamation of two or more systems together under a unified policy requires making some assumptions about and adjustments to policy coverage and also to administrative responsibilities. For instance, the degree of interdependence among policy authorities is an important factor. While some duties may be shared between the authorities of each system, other may be allocated exclusively to certain authorities to effect the required policy. The examples given below illustrate the types of considerations involved in the amalgamation of system policies and the types of trade-off decisions that can occur. Other, less involved examples are also available elsewhere [Fer05, Fer11].

6.4.1 DAC and Email

As an example of an amalgamated policy specification that involves multiple policy class, consider the operating system and mail system examples described earlier in this report. Each system policy is expressed using a single policy class that involves an intra-policy class pattern for administration. The individual policies are somewhat independent, insofar as the scope of objects covered by each policy is distinct and non-overlapping. However, the set of users is potentially the same for each system and overlaps considerably.

The main adjustment needed in this example is to determine how administrative duties over users should be treated. The approach taken is to treat the operating system as foundational and a prerequisite for use of the mail system. Accordingly, the PA assigns the entire responsibility for creating and deleting users to the DA of the operating system (DA-OS). The DA for the mail system (DA-MS) no longer creates or deletes a user, and instead, relies on the DA-OS to perform this function. Once a user has been established by the DA-OS, the DA-MS can assign or unassign mail containers to and from the user, thereby enabling and disabling the user's capability to use the mail system.

Figure 18 gives an example of a partial authorization graph illustrating the DAC segment of the integrated DAC-Mail System policy. As before, user u_{1101} is the DA-OS for the DAC policy class, and u_1 and u_2 are typical users of the system. The domain authority is created by the PA (not shown) and establishes the users, objects, and relationships for the system. Those relationships are shown in blue and gray, whereby the blue denotes administrative relations and the gray denotes non-administrative relations.

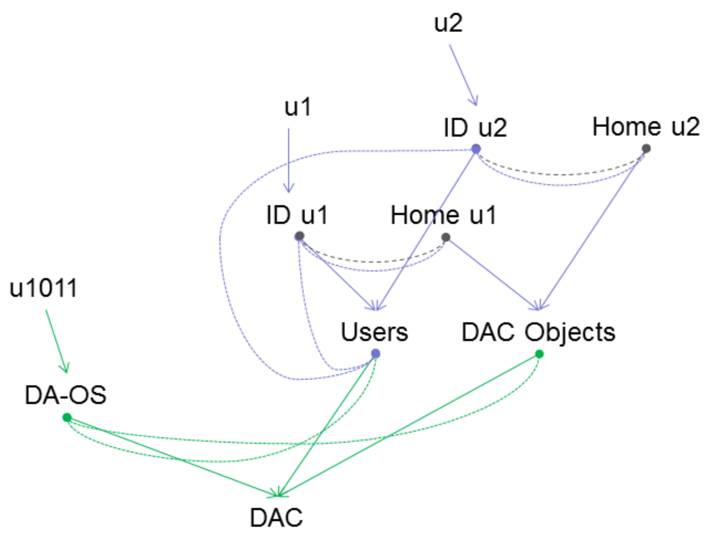

Figure 18: DAC Segment of the Integrated System Policy

Figure 19 gives an example of a partial authorization graph illustrating the Mail System segment of the integrated DAC-Mail System policy. User u_{1001} is the domain authority for the policy class, and u_1 and u_2 are DAC users over which the DA-MS has been assigned authority from the PA to establish objects and relationships that pertain exclusively to the mail system. The relationships established for u_2 are shown in red and gray, whereby the red denotes administrative relations and the gray denotes non-administrative relations.

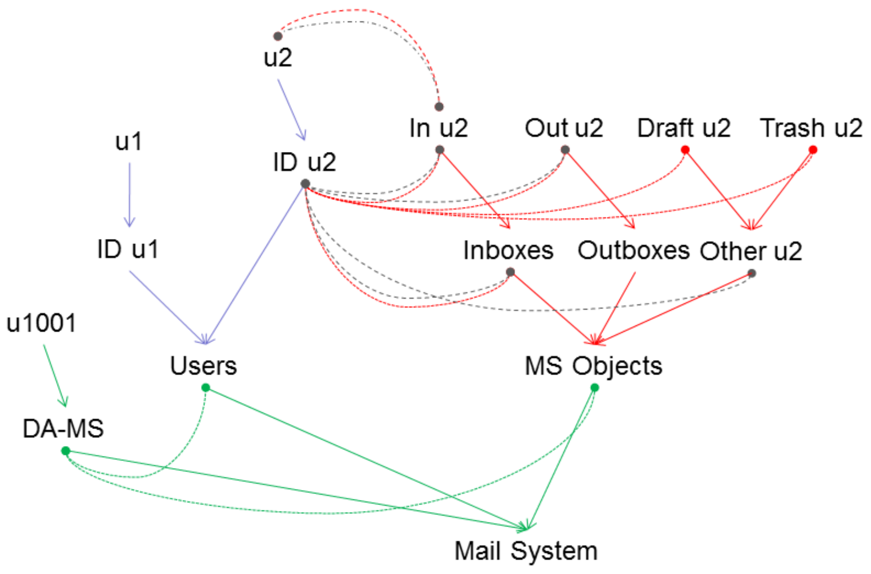

Figure 19: Mail System Segment of the Integrated System Policy

An example of an entire authorization graph for the integrated system policy is shown in Figure 20. The DAC operating system objects and relations for user u_2, which were shown in a Figure 18, are omitted to avoid an overly busy illustration.

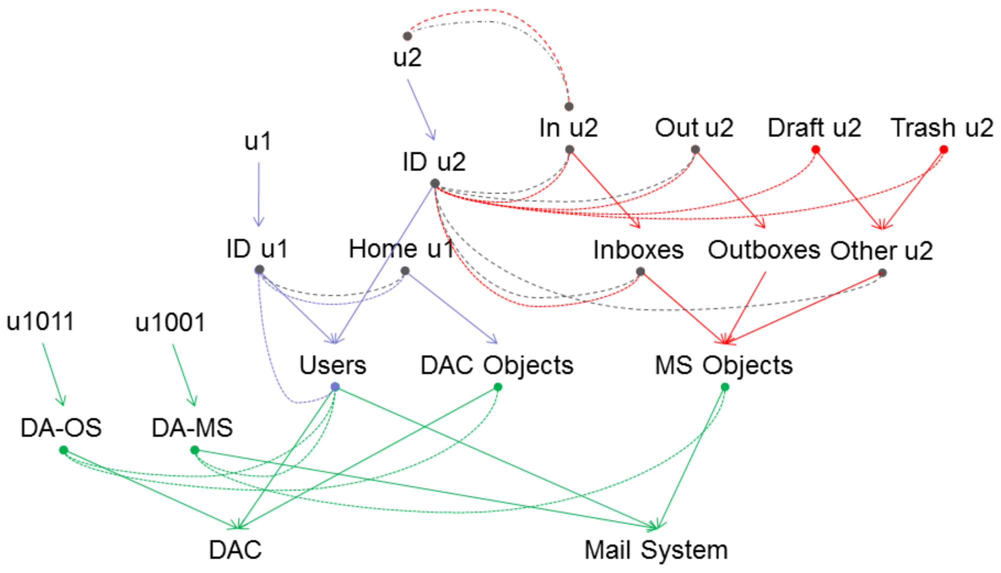

Figure 20: Authorization Graph of the Integrated System Policy

Several inferences can be drawn from this example. The first is that policies grow very quickly and become unwieldy to view in their entirety. The second is that administrative relationships, particularly associations, exceed non-administrative ones with respect to the overall authorization pattern. The third and final inference is that when amalgamating policies together, establishing an approach that fits the needs of all policy stakeholders is an important prerequisite to making any adjustments to existing policies.

6.4.2 DAC and MAC

The following MAC policy is defined as an extension to the DAC policy discussed previously to form an integrated DAC-MAC policy. A MAC policy is by its very nature comprehensive; therefore, all existing users and objects need to be placed under it for compliance. Three security levels pertain to this multi-level security policy extension: high, medium, and low. Security levels are assigned to users and objects, and are also applicatory to processes working on behalf of users. Users are assigned levels that represent their trustworthiness, while objects are assigned levels that represent their sensitivity. A security level x is said to dominate a security level y, if x is greater than or equal to y. In this example, the security level high dominates medium and low, while medium dominates low.

The PA establishes the clearance and classification levels for the multi-level policy illustrated in Figure 21. Since this part of the policy specification is mandatory and remains constant, there is no need to have a DA manage the policy once it is specified. All users are assigned to one of three user attributes that represent a user clearance. Users cleared to the high, medium, and low levels of trust are assigned to the H_T, M_T, and L_T user attributes respectively. Similarly, all objects are assigned to one of three object attributes that represent a classification. Objects classified at the high, medium, and low sensitivity levels are assigned to the H_S, M_S, and L_S object attributes. In this example, read means that information flows from the object to the user (or one of its processes), which implies execute, while write means that information flows from the user to the object.

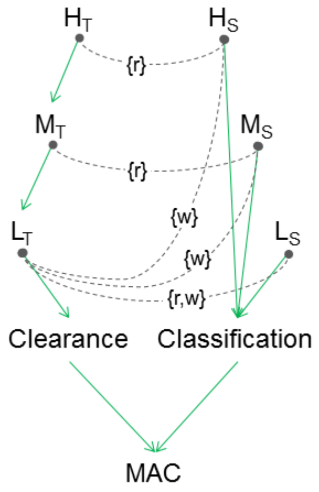

Figure 21: Authorization Graph for the MAC Policy Segment

These policy assignments allow users and associated processes that are cleared at the high security level to perform read operations on objects classified at the high, medium, and low security levels. Users (and their processes) that are cleared at the medium level are allowed to perform read operations only on objects classified at the medium and low levels. Finally, users (and their processes) that are cleared low are allowed to perform read operations only on objects classified at the medium and low levels. That is, the simple security property is reflected in the policy.

The approach typically used with the PM to prevent leakage of sensitive data to unauthorized principals is to recognize when an authorized process reads sensitive information and then constrain that process or its associated user from writing to objects accessible to any unauthorized principals. This approach, which entails the use of obligations, is general enough to support a large variety of policies that depend on the absence of leakage. Separation of duty and other history-based policies can also be supported in a similar manner—recognizing when a critical event occurs and taking action to constrain the process involved or its associated user from taking an unwanted action or set of actions.

For this MAC example, to prevent a user's process from writing to an object that is at a lower security level than any object it has read, additional restrictions are needed. The obligations specified for this policy to fulfill this objective are as follows:

When EC.op = r $\wedge$ EC.o $\rightarrow^*$ H_S **do**
CreateConjunctiveP-Prohibition (EC.p, {w}, ∅, {H_S})[7]

When EC.op = r $\wedge$ EC.o $\rightarrow^*$ M_S **do**
CreateConjunctiveP-Prohibition (EC.p, {w}, ∅, {H_S, M_S})

The first obligation specifies that once a process successfully reads an object in the H_S container, a process prohibition is created to prevent the process from writing to objects that are outside the H_S container. Similarly, the second obligation specifies that once a process successfully reads an object in the M_S container, a process prohibition is created to prevent the process from writing to objects outside M_S or H_S containers. The two obligations can also be written using the disjunctive form of a process deny prohibition, as follows:

When EC.op = r $\wedge$ EC.o $\rightarrow^*$ H_S **do**
CreateDisjunctiveP-Prohibition (EC.p, {w}, {L_S, M_S}, ∅)

When EC.op = r $\wedge$ EC.o $\rightarrow^*$ M_S **do**
CreateDisjunctiveP-Prohibition (EC.p, {w}, {L_S}, ∅)

The first obligation establishes that once a process successfully reads an object in the H_S container, the process cannot write to objects that are in the L_S and M_S containers. The second establishes that once a process successfully reads an object in the M_S container, the process cannot write to objects that are in the L_S container. These complimentary ways to state a process deny prohibition for this policy are possible, since the L_S, M_S, and H_S containers are mutually exclusive and collectively exhaustive with respect to the objects of the DAC-MAC system. Regardless of the form of the obligation pair used, the first successful read of an object by a

[7] The semantics of the administrative command used in this obligation is essentially the same as that for the command CreateDisjunctiveProcessProhibition given in Appendix C, with the exception thatthe preconditions for this command asserts that the user who defined the obligation must hold sufficient authorization to execute the body of the command.

process constrains the process to write only at or above the sensitivity level of the object, consistent with the ★-property.

Adding the DAC policy specification discussed in earlier examples to the MAC policy specification, results in the authorization graph illustrated in Figure 22. Note that the policy specified up to this point is done by the PA. Going forward, the DA established by the PA has responsibility to govern the users and objects of the system. One subtle extension made to the DAC policy is that the DA must have the authority to assign users a security clearance via the H_T, M_T, and L_T attributes. This authority is represented in the authorization graph by the administrative association from the DA user attribute in the DAC policy class to the Clearance user attribute in the MAC policy class.

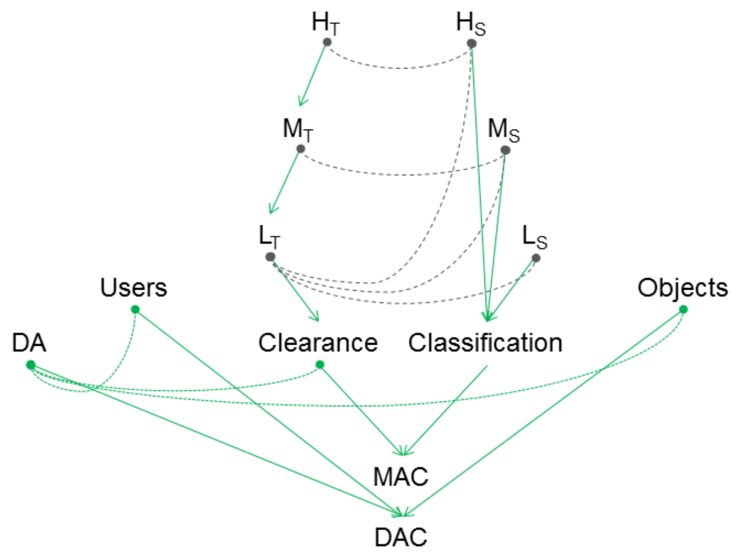

Figure 22: Authorization Graph for MAC-DAC system

One additional consideration concerning object creation is required, however. When an object is created under the DAC policy, it is assigned to the home container of the user. A newly created object also needs to be assigned an appropriate classification level under the MAC policy. Different policies regarding object creation can be supported by the PM model. For the integrated DAC-MAC policy, the policy is that the assigned classification level of the object defaults to that of the user's clearance. Figure 23 illustrates the policy applied to the object o_1 in the home container of user u_1. Such assignments can be accomplished in one of two ways. The first is to have the create object routine make the assignment directly. The second way is more indirect and carried out through an administrative obligation that makes the assignment when triggered by a create-object-in-object-attribute (create-OinOA) event.

For the first approach, the existing create object in object attribute command, as part of the trusted computing base would be retained, and a slightly modified version that allows a user to make classification assignments for newly created objects would be created and added to the trusted computing base. Users would also need to be granted sufficient authority to execute the new command via an administrative association. The enhanced object creation command in effect consitutes an extension to the PM model, which can support not only this MAC policy, but also similar types of lattice-based policies.

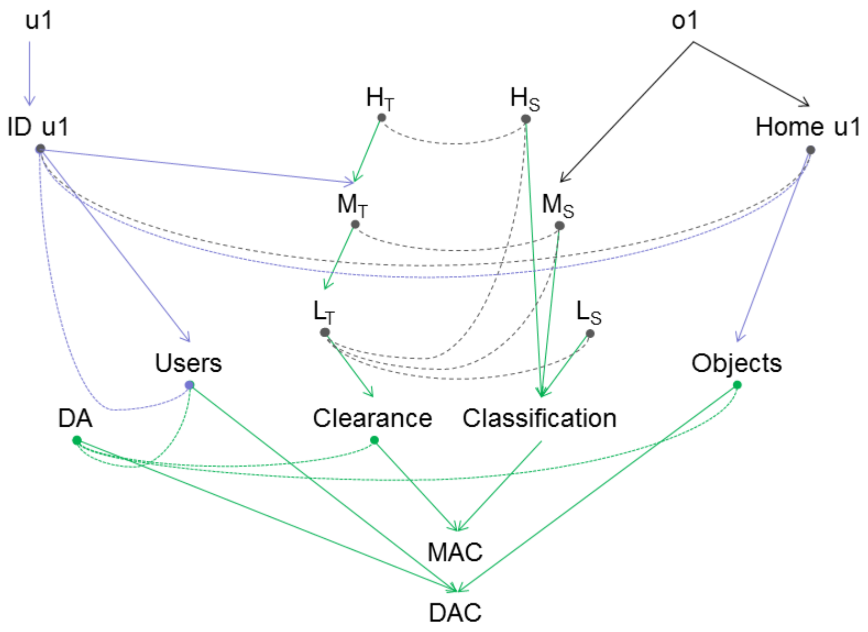

Figure 23: DAC-MAC Authorization Graph with Populated User

For the second approach, additional authority needs to be granted to the DA via an administrative association from DA to Classification. That authority, coupled with the authority the DA already holds over Objects, allows a DA to create obligations that exercise the DA's authority to assign a newly created object in the DAC policy class to an appropriate classification level in the MAC policy class. The following set of administrative obligations is needed for this approach:

When AEC.aact = create-OinOA $\wedge$ AEC.u $\rightarrow^+ L_T$ $\wedge$ $\neg$(AEC.u $\rightarrow^+ M_T$) **do**
CreateOinOAAssignment (AEC.argseq.1, L_S)[8]

When AEC.aact = create-OinOA $\wedge$ AEC.u $\rightarrow^+ M_T$ $\wedge$ $\neg$(AEC.u $\rightarrow^+ H_T$) **do**
CreateOinOAAssignment (AEC.argseq.1, M_S)

When AEC.aact = create-OinOA $\wedge$ AEC.u $\rightarrow^+ H_T$ **do**
CreateOinOAAssignment (AEC.argseq.1, H_S)

The first obligation applies to users with L_T clearance. For those users, it assigns objects newly created within their home container to the L_S classification container. The second and third obligations carry out similar assignments of newly created objects to M_S and H_S containers for

[8] The AEC.argseq of an administrative event context for an administrative access request involving the create-OinOA administrative action is [o, oa], where AEC.argseq.1 contains o, the identifier of the object that was created, and AEC.argseq.2 contains oa, the identifier of the object attribute to which the object was assigned.

users with M_T and H_T clearances respectively. The main drawback with this approach is that the DA must be trusted to a greater degree than in the first approach, since the DA involvement is needed to specify part of the MAC policy. However, given that the DA is already trusted to assign newly created users to a clearance level, the additional responsibility is not unreasonable.

As with the original DAC system, other users have the discretion to grant user u_1 the authority to read or write objects they control, which are classified at the H_S, M_S, or L_S sensitivity level. However, because that authority must be held under both the DAC and MAC policy classes, the multi-level restrictions defined in MAC prevail over any conflicting authority granted in DAC, as would be expected. The reverse is also true. Even if a user has sufficient clearance to access certain information under MAC, the user may not be given access to the information unless the user has a specific need to know. That is, access to the information must be necessary to carry out official duties and must be expressed via an explicit grant of authority.

One further improvement to the policy is possible. Note that under either approach, a user currently can create an object only at the classification level equivalent to its clearance. If a user is granted discretionary access by another use, it can write to an object at a lower level of classification, provided that it has not read an object at a higher classification level. However, it cannot create an object at a lower level of classification than its clearance equivalent and write to it. With the first approach discussed above, this feature can be instituted easily by modifying the enhanced create object in object attribute routine to create objects based on a classification level argument supplied by the user. Existing prohibitions prevent writing to the object, if the user has read an object at a higher classification level, as would be warranted. With the second approach, however, this policy adjustment is not possible, because there is not a way for the user to convey the intended security level to the prohibition making the assignment.

7. Architecture

The PM functional architecture is intended to accommodate a number of different situations using a variety of approaches. The architectural components of the PM are amenable to implementation in both centralized and distributed systems. For the former, interactions between the architectural components take place entirely within a computer system and the interfaces between components are defined in terms of programing interfaces. For the latter, interactions take place across a network and the same information is conveyed through a network protocol.

Many types of hybrid designs in which some components reside within a single system, while others are located in other systems, are also possible. A separate decision to use either a programming interface or network protocol for each interface may be made as appropriate, because the two variants are functionally equivalent.

7.1 Architectural Components

The PM functional architecture involves several components that work together to bring about controlled access to protected resources. The components include a Policy Enforcement Point (PEP), a Policy Decision Point (PDP), an Event Processing Point (EPP), a Policy Administration Point (PAP), a Policy Information Point (PIP), and a Resource Access Point (RAP). Figure 24 illustrates these components and their interfaces.

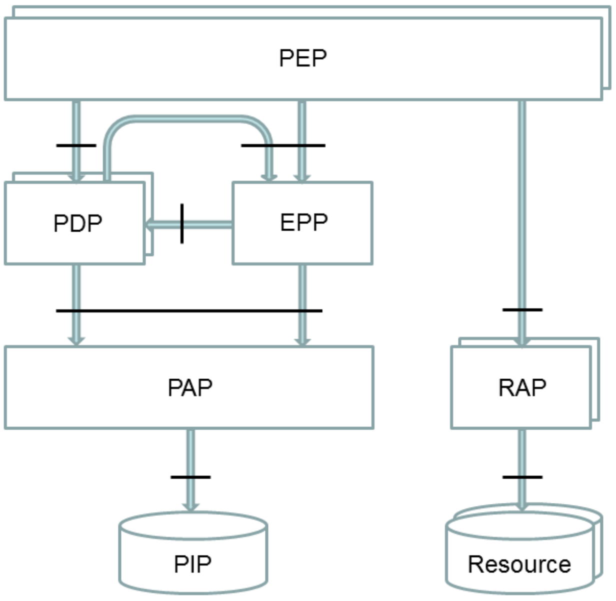

Figure 24: Architectural Components of the PM

Further details for each of the architectural components are as follows:

- Policy Enforcement Point. PM-aware applications must rely on a PEP to gain access to protected resources and to policy information via a programming interface that it provides. More than one PEP may exist to service applications. The PEP ensures that access requests are validated as meeting the specified policy before access to the resources in question occurs. To accomplish this objective, the PEP works in tandem with a PDP and must not be bypassable.

The PEP conveys access requests issued by the application to a PDP for a reference mediation decision. An application's access request includes the identity of the requestor, the requested action, and the arguments of the action, including the targeted resource(s) and optional data. Both non-administrative (i.e., input/output operations) and administrative access requests are handled by the PEP.

For non-administrative input/output requests that are denied, the PEP notifies the application of an authorization failure. For requests that are granted, the PEP receives the Uniform Resource Identifier (URI) for the physical resource in question. This enables the PEP to carry out the requested access using the URI to identify the RAP, issue the appropriate command(s) against it, and return the results to the application. The PEP also generates an event after each access request it successfully executes, which conveys the event context to the EPP.

Decisions on access requests involving administrative actions are handled somewhat differently. For administrative requests that are denied, the PEP notifies the application of an authorization failure. For requests that are granted, the PEP receives the results of the administrative action taken against the abstract resources in the PIP, which the PDP carries out itself via the PAP. The PEP does not generate events for access requests carried out by the PDP.

- Policy Decision Point. A PDP determines whether an access request made by a PEP complies with policy and renders a grant, deny, or error decision accordingly. The PDP performs the reference mediation function defined in the PM model. It also carries out all access requests that involve administrative actions for which a grant decision has been rendered. Multiple PDPs may exist in the PM environment.

  The PDP obtains the information it needs to validate the access request from the PIP via the PAP. If an access that involves input/output operations on a physical resource is granted, the PDP supplies the necessary details to the requesting PEP for locating and accessing the resource. If denied, only the decision is conveyed back to the PEP. If an access that involves administrative actions on an abstract resource is granted, the PDP performs the access and supplies the results to the requesting PEP along with the decision. The PDP also generates an event describing the access, for eventual processing by the EPP.

  The PDP also performs reference mediation for the EPP on obligations that the EPP has matched to an event it received. In this situation, accesses that involve multiple administrative actions must be mediated collectively, and if granted, carried out by the PDP. The PDP also generates events describing the accesses, for eventual processing by the EPP.

- Policy Information Point. The PIP contains the data structures that define the policy elements and the relationships between policy elements that collectively constitute the access control policy enforced by the PM. All changes to the policy occur at the PIP, but originate from the PAP. The PIP must ensure that transactions issued by the PAP are processed reliably.

- Resource Access Point. A RAP allows one or more PEPs to gain access to protected resources. The only method of accessing protected resources is via a RAP. Multiple RAPs can exist, but each protected resource is accessible only through a single RAP. The PEP issues a command containing its identifier, the location of the physical resource, the operation, and any required data to the RAP. The RAP returns data and status information to the PEP. The RAP does not allow access to resources to any entity other than a PEP.

- Policy Administration Point. A single PAP manages all access to the contents of the PIP, similar to the way a RAP serves as a managed access point to protected resources. A PAP provides read, modify, and write access to the data contained within the PIP (i.e., the policy configuration), and ensures that access is serialized. A PAP limits the EPP to read access only, but allows a PDP both read and write access.

- Event Processing Point. A single EPP is responsible for comparing events against event patterns that have been defined in obligations residing at the PIP. For each event that is matched, the EPP uses a PDP to perform reference mediation on the associated event response (i.e., the sequence of administrative actions defined for each obligation), to carry out the response, if access is granted, and also to generate events describing each access. The EPP can be viewed as a transaction processing monitor, whose performance is crucial to the overall effectiveness of the architecture. To avoid contention for accessing a PDP, one of them can be designated for exclusive use by the EPP and collocated with it.

The PM model separates the policy expression, represented by the data elements and relationships maintained in the PIP, from the mechanisms that enforce the policy, contained mainly in the PEP and PDP, and supported by the PAP and RAP. The EPP can be regarded as an automation facility for defining administrative actions that must be taken immediately after the occurrence of certain, predefined, successfully executed access requests. While the EPP is not needed to express all security policies, for some, such as those that involve separation of duty constraints, it is essential.

The architecture of the PM lends itself to a range of implementation choices, as mentioned earlier. One interesting aspect is that the more distributed a system implementation becomes, the greater the propensity is for race conditions to arise. The main source of contention is that access request decisions taken by one set of components are carried out by others, all of which are acting concurrently against shared resources. To complicate matters further, event-driven administrative actions taken automatically may occur, which also affect the state of shared resources. Undesired, inconsistent results can ensue unless methods are in place to allow critical sections of an execution stream to be executed atomically.

7.2 Client Applications

A user signs onto the PM from a client system typically through a Graphical User Interface (GUI). A successful signon opens a user session with the PM environment.

A user can have only a single PM session open at any time. Within a session, a logical view can be rendered for the user, which displays all of the user's accessible resources, such as files, e-mail messages, and work items. As an alternative, the user can be presented with a view of available resource categories and prompted to select a specific set of accessible resources. Within either approach, the user launches applications via resource selection and initiates processes that request access to resources protected by the PM. Changes in policy can affect the user's view of accessible resources and must be reflected immediately.

PM-aware applications require the use of a PEP to access protected resources. The PEP provides an Application Programming Interface (API) for developing PM-compliant applications. As shown in Figure 25, the PEP API is the only means available for an application to interact with the PM environment and gain access to protected resources. Alternatively, existing applications developed without the PM in mind can be adapted for the PM by intercepting access requests at key points in the code and converting them to calls on the PEP interface, for eventual mediation by the PDP. The physical location of each object is unknown to the application, but is known to the PM and is included with each access request that is granted by the PDP. The PEP enforces the PDP's decision, granting or rejecting the access to the object from the application's processes.

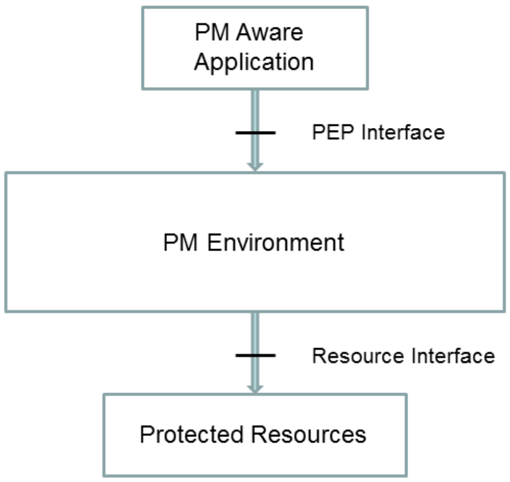

Figure 25: Application's Perspective of the PM Environment

Applications that conduct administrative actions on policy structures work a bit differently. Take, for example, a policy manager application developed to allow an administrator to render part of the PM's current policy configuration within its domain (e.g., in the form of an authorization graph), to navigate the configuration, and to create and delete policy elements and relations between policy elements (i.e., assignments, associations, prohibitions, and obligations). While such an application would use a PEP as other PM-aware applications do, the requested administrative actions do not involve protected physical resources and instead, pertain exclusively to abstract resources—the policy structures maintained within the PM environment at the PIP.

7.3 Security Considerations

The effectiveness of the PM architecture to control access depends on adequately protecting the data elements and relationships that represent the security policy, and also the PM components that contain the mechanisms for policy enforcement. It is also critical that the PM components enforcing policy cannot be bypassed. Potential adversaries may involve more than one legitimate user working in collaboration to defeat access control to PM protected resources, as well as non-legitimate external parties. Adversaries may have access to the data paths between components and be able to eavesdrop on exchanges.

PM entities are trusted parties that must work together closely to ensure reference mediation is carried out correctly. Authentication protocols enable one entity to prove its claimed identity to another entity, typically through some cryptographic means. In a distributed system where several entities of the same type exist (e.g., multiple PDPs), it may be necessary to find an available entity to use from amongst them. Since the potential for an attacker to masquerade as a trusted entity exists (e.g., via a man-in-the-middle attack), authentication between PM components is an important safeguard for verifying that an entity is what it claims to be. In addition, authentication can prevent PM components from being bypassed by an attacker. However, authentication protocols are complex, and because of the complexity involved, implementations can be done incorrectly and result in vulnerabilities such as incorrect interpretation of credentials.

Distributed entities rely on networks communications to interoperate. Without sufficient safeguards in place, messages transmitted between PM entities are potentially susceptible to attack by malicious third parties. Security protocols are complex, which often leads to implementations containing errors that allow exploitation. Protocols may also contain design flaws that lend themselves to exploitation. Protocol attacks may involve message replay, content analysis, deletion, and modification attacks and result in unauthorized disclosure, policy circumvention, state corruption, violation of privacy, or denial of service. Single-occurrence PM components such as the PAP can be particularly attractive targets for denial of service attacks, since they represent choke points in the access control mechanism.

The PEP, PDP, PAP, and other PM components may themselves contain vulnerabilities that could be exploited to compromise the access control policy and its enforcement by the PM. For example, race conditions between components, discussed earlier, may result in time-of-check/time-of-use vulnerabilities. Other components of a distributed system on which the PM components depend, such as a virtual machine monitor, operating system, or domain-name system (DNS) resolver, may also be exploited and lead to a policy compromise. Similarly, systems supporting client applications and also the client applications themselves may contain vulnerabilities susceptible for exploitation. Even if the PM implementation functions perfectly, transactions stemming from the application may be forged, or intercepted and modified on the client system before the PM components are involved.

8. References

[Bre89] David F.C. Brewer and Michael J. Nash, The Chinese Wall Security Policy, IEEE Symposium on Research in Security and Privacy, Oakland, California, pp. 206-214, May 1-3, 1989.

[Fer05] David F. Ferraiolo, Serban I. Gavrila, Vincent C. Hu, and D. Richard Kuhn, Composing and Combining Policies Under the Policy Machine, ACM Symposium on Access Control Models and Technologies (SACMAT) 2005, pp. 11-20, <URL: https://csrc.nist.gov/staff/Kuhn/sacmat05.pdf>.

[Fer11] David F. Ferraiolo, Vijayalakshmi Atluria, and Serban I. Gavrila, The Policy Machine: A novel architecture and framework for access control policy specification and enforcement, Journal of Systems Architecture, Volume 57, Issue 4, April 2011, pp. 412-424.

[Gra72] G. Scott Graham and Peter J. Denning, Protection: Principles and Practice, AFIPS Spring Joint Computer Conference, pp. 417-429, May 1972.

[Har76] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman, Protection in Operating Systems, Communications of the ACM, Volume 19, Number 8, August 1976.

[Hu06] Vincent C. Hu, David F. Ferraiolo, and D. Rick Kuhn, Assessment of Access Control Systems, Interagency Report (IR) 7316, NIST, September 2006, <URL: http://csrc.nist.gov/publications/nistir/7316/NISTIR-7316.pdf>.

[Kuh98] D. Rick Kuhn, Role Based Access Control on MLS Systems Without Kernel Changes, ACM Workshop on Role Based Access Control, October 22, 1998, pp. 25-32, <URL: http://csrc.nist.gov/groups/SNS/rbac/documents/design_implementation/kuhn-98.pdf>.

[NCSC87] A Guide to Understanding Discretionary Access Control in Trusted Systems, National Computer Security Center, NCSC-TG-003, Version-1, September 30, 1987.

[Osb00] Sylvia Osborn, Ravi Sandhu, and Qamar Munawer, Configuring Role-Based Access Control to Enforce Mandatory and Discretionary Access Control Policies, ACM Transactions on Information and System Security (TISSEC), Volume 3, Number 2, May 2000, pp. 85-106, <URL: http://profsandhu.com/journals/tissec/dac00(org).pdf>.

[San92] Ravi S. Sandhu, Lattice-Based Enforcement of Chinese Walls, Computers and Security, Volume 11, Number 8, December 1992, pages 753-763, <URL: http://profsandhu.com/journals/csec/csec92-cwall-org.pdf>.

[San94] Ravi Sandhu and Pierangela Samarti, Access Control: Principles and Practice, IEEE Communications Magazine, Volume 32, Number 9, pages 40-48, September 1994, <URL: http://profsandhu.com/journals/commun/i94ac(org).pdf>.

[Sal75] Jerome H. Saltzer and Michael D. Schroeder, The Protection of Information in Computer Systems, pp. 1278-1308, Proceedings of the IEEE, Vol. 63, No. 9, September 1975.

Appendix A—Acronyms

API	Application Programming Interface
DA	Domain Administrator
DAC	Discretionary Access Control
DNS	Domain Name System
EPP	Event Processing Point
MAC	Mandatory Access Control
MS	Mail System
PA	Principal Authority
PAP	Policy Administration Point
PDP	Policy Decision Point
PEP	Policy Enforcement Point
PIP	Policy Information Point
PM	Policy Machine
RAP	Resource Access Point
SA	Sub-domain Administrator
S^2A	Sub-SA
S^3A	Sub-S^2A
URI	Uniform Resource Identifier
XACML	eXtensible Access Control Markup Language
XML	eXtensible Markup Language

Appendix B—Notation

The **empty set** is denoted by $\emptyset$.

The **powerset** of a set S is the set of all subsets of S, including the empty set, and is denoted by 2^S.

A **finite sequence** in a set S is a function from {1, 2, ..., n} to S for some n > 0. The k^th element is denoted as s_k, and the entire sequence as $s_1, s_2, ..., s_n$.

A **list** is a linearly ordered, finite collection of items from one or more sets. The k^th element is denoted as l_k, and the entire sequence as $[l_1, l_2, ..., l_n]$.

The **cardinality** of a set S is defined as the number of elements of the set and denoted by |S|.

The **union** between two sets, S1 and S2, is defined as {x | (x∈S1 or x∈S2) or (x∈S1 and S2)}, and is denoted by S1 ∪ S2.

The **intersection** between two sets, S1 and S2, is defined as {x | x∈S1 and x∈S2}, and is denoted by S1 ∩ S2.

The **Cartesian product** or **cross product** of two sets, S1 and S2, is defined as {(x, y) | x∈S1 and y∈S2}, and is denoted by S1 × S2.

The **relative complement** of the set S2 in the set S1, also known as the **set theoretic difference** between S1 and S2, is defined as {x | x∈S1 and x∉S2}, and is denoted by S1 – S2.

The **absolute complement** of a set S, denoted by $\overline{S}$, is the set of elements not in S, but in the universal set of all elements, U (i.e., $\overline{S}$ = U – S). For the notation used in the PM model, U is equal to PE, the set of all policy elements.

R is a **binary relation on** a set S, iff R ⊆ S × S.

R is a **binary relation from** the set S1 to the set S2, iff R ⊆ S1 × S2.

An **ordered pair** from the relation R is denoted by either (x, y) ∈ R or x R y.

List of Common Symbols:

=	equality
≠	inequality
$\overset{\text{def}}{=}$	definition

$\Rightarrow$	material implication (i.e., implies)
$\Leftrightarrow$	material equivalence (i.e., iff)
$\neg$	logical negation
$\exists$	existential quantification
$\exists!$	uniqueness quantification
$\nexists$	existential quantification negation (i.e., $\neg\exists$)
$\forall$	universal quantification
$\wedge$	logical conjunction
$\vee$	logical disjunction

Precedence among logical operators given in descending order is as follows: $\neg$, $\forall$ and $\exists$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$. All operators are right associative.

$\emptyset$	empty set (i.e., { })
$\in$	set membership
$\notin$	set membership negation
$\subseteq$	subset
$\subset$	proper subset
$\supseteq$	superset
$\supset$	proper superset
$\overline{\quad}$	absolute complement of a set
$-$	set-theoretic difference
$\cup$	set-theoretic union
$\cap$	set-theoretic intersection
x	Cartesian product
2^S	power set of set S
$\lvert S \rvert$	cardinality of set S

$\rightarrow^{+}$	the transitive closure of the assignment relation $\rightarrow$
$\rightarrow^{*}$	the reflexive and transitive closure of the assignment relation $\rightarrow$
$x^{\blacklozenge}$	object range of a policy element x
$x^{\lozenge}$	complementary object range of a policy element x
$x^{\bullet}$	element range of a policy element x
x°	complementary element range of a policy element x

Appendix C—Semantics of Administrative Routines and Commands

All administrative actions are conducted through the use of administrative commands and routines Administrative routines constitute the range of primitive actions that can be taken against the policy elements and relationships of the PM and are the means for specifying the authorization state of a system. Administrative commands are a composition of administrative routines, which are used to define more complex administrative actions. Administrative routines are executed on behalf of a user via administrative commands and may also be executed in response to a recognized event occurrence. This appendix contains a complete list of the core administrative routines and commands of the PM and their semantic definition.

The semantic description of an administrative routine or command differs from a syntactic description or programming language representation. The semantic descriptions define the correct behavior expected of routines and commands, necessary to maintain the security properties of the PM as it operates and transitions between states. The specifications should not be interpreted as programming statements, and instead be interpreted as changes to model structures that occur when a command or routine is correctly invoked. Behavioral aspects other than security are outside the scope of these descriptions.

Pre- and postconditions are defined for each administrative routine and command. Preconditions denote requirements. They are expressed as a logical expression that must be satisfied when the routine or command is invoked. Postconditions denote expectations. They specify properties of the model that change with the invocation of the routine or command and should be met when the execution is completed. Postconditions are also expressed as a logical expression.

The preconditions for administrative routines ensure validation that the arguments supplied for the formal parameters of the routine are of the correct type, and that the basic properties of the model are observed. One small exception applies regarding the PC-reachability property (i.e., for all x in PE, there exists a policy class pc, such that $x \rightarrow^* pc$). The preconditions for administrative commands are similar, but in addition, ensure validation that the process involved in the access has sufficient authorization to carry out the command, and that the PC-reachability property is maintained consistently.

The following conventions are observed in the semantic descriptions given below:

- Administrative routines and commands are atomic; their effects are indivisible and uninterruptable.
- The main body of an administrative routine or command specifies state changes for those model elements and relations that are affected by its execution—the state of any unspecified element or relation is unaffected and remains unchanged.
- Model elements and relations, whose state changes with the execution of an administrative routine or command, are indicated with the prime symbol.

- All specified preconditions must be satisfied for the change of state described in the body of the routine or command to occur. More simply stated, no state changes occur unless the preconditions are satisfied.

- Comments may appear in the semantic descriptions; single line comments begin with a double backslash, and multiline comments begin with a backslash followed by an asterisk and end with an asterisk followed by a backslash.

- The formal parameters of an administrative routine and command serve individually as either an input or output to the routine or command, but never as both an input and an output.

To simplify the specification of preconditions, liberties were taken with some of the notation used; namely, the tuples of a relation are treated as members of a set when the predicate calculus qualifiers $\exists$, $\nexists$, and $\forall$ are applied. For example, a triple of the relation ASSOC has three elements: a user attribute, a set of operations, and an object attribute. To specify that a triple (a, b, c) of ASSOC with the property a=x does not exist would normally be done as follows:

$\forall a \in UA, \forall b \in Ops, \forall c \in OA: \neg((a, b, c) \in ASSOC \wedge a = x)$

Instead, this formula is expressed in the preconditions as follows:

$\nexists (a, b, c) \in ASSOC: a = x$

The qualifier in the latter shorthand expression more succinctly denotes both the set membership of each element of the tuple and the tuple membership (or rather, the lack thereof) with the relation. Full predicate calculus notational equivalencies of shorthand expressions involving an existential or universal quantifier (i.e., $\nexists$ in the above formula replaced by $\exists$ or $\forall$ respectively) also exist.

C.1 Element Creation Routines

The routines below specify the semantics for the routines used to create the various policy elements of the model. No preconditions apply, since variables supplied as arguments pertain only to output parameters. The postconditions specified for these routines ensure that any variable returned as a formal parameter of a routine is valid.

Instantiation(set), returns id

```
{
/* a semantic function that denotes the allocation of an instance of an entity
comparable to members of a specified set and returns a unique identifier for
the entity */ [9]
}
```

[9] Unique system-generated identifiers are essential for determining whether two references pertain to the same entity. This is particularly in situations where the name of an entity can change or an entity can be referenced in multiple ways. To avoid covert channels, the pattern of successive identifiers generated should not be predictable.

```
CreateU(x)
    preconditions: none
        {
        $x' = \text{Instantiation}(U)$   // x′ is the unique identifier of the new policy element
        $U' = U \cup \{x'\}$
        }
    postconditions: $x' \in U'$

CreateUA(x)
    preconditions: none
        {
        $x' = \text{Instantiation}(UA)$
        $UA' = UA \cup \{x'\}$
        }
    postconditions: $x' \in UA'$

CreateO(x)
    preconditions: none
        {
        $x' = \text{Instantiation}(O)$
        $O' = O \cup \{x'\}$
        $OA' = OA \cup \{x'\}$
        }
    postconditions: $x' \in O' \land x' \in OA'$

CreateOA(x)
    preconditions: none
        {
        $x' = \text{Instantiation}(OA)$
        $OA' = OA \cup \{x'\}$
        }
    postconditions: $x' \in OA'$

CreatePC(x)
    preconditions: none
        {
        $x' = \text{Instantiation}(PC)$
        $PC' = PC \cup \{x'\}$
        }
    postconditions: $x' \in PC'$

CreateP(x)
    preconditions: none
        {
        $x' = \text{Instantiation}(P)$
        $P' = P \cup \{x'\}$
        }
    postconditions: $x' \in P'$
```

C.2 Element Deletion Routines

The routines below specify the semantics for the routines used to delete the various policy elements of the model. Besides ensuring that the arguments supplied for the formal parameters of a routine are valid, the preconditions specified for these routines also ensure that certain model properties are preserved. The policy element in question must not be involved in any defined relation. For example, if a user attribute is involved in an assignment, association, or prohibition relation, the attribute cannot be deleted until it is no longer involved in the relation.

Disinstantiation(x), returns nihil

```
        {
        /* a semantic function that denotes the deallocation of an instance of an entity
        with the identifier x */
        }
```

DeleteU(x)

```
    preconditions: $x \in U \ \wedge \ \nexists y \in UA: x \rightarrow^{+} y \ \wedge$
    // ensure no processes that operate on behalf of x exist
    $\nexists p \in P: x = process\_user(p) \ \wedge$
    // ensure no assignments stemming from the user exist
    $\nexists (a, b) \in \rightarrow: x = a$
    // ensure no prohibitions exist that involve the user
    $\nexists (a, b, c, d) \in U\_deny\_disjunctive: a = x \ \wedge$
    $\nexists (a, b, c, d) \in U\_deny\_conjunctive: a = x \ \wedge$
    $\nexists (a, b, c, d) \in U\_Admin\_deny\_disjunctive: a = x \ \wedge$
    $\nexists (a, b, c, d) \in U\_Admin\_deny\_conjunctive: a = x$
        {
        $U' = U - \{x\}$
        $x' = Disinstantiation(x)$
        }
    postconditions: $x \notin U'$
```

DeleteUA(x)

```
    preconditions: $x \in UA \ \wedge$
    // ensure no assignments involving the user attribute exist
    $\nexists y \in UA: x \rightarrow^{+} y \ \wedge \ \nexists y \in PC: x \rightarrow^{+} y \ \wedge \ \nexists y \in UA: y \rightarrow^{+} x \ \wedge$
    // an alternative expression for the above: $\nexists (a, b) \in \rightarrow: (x = a \vee x = b) \ \wedge$
    // no associations or prohibitions must exist in which the ua involved
    $\nexists (a, b, c) \in ASSOC: x = a \ \wedge$
    $\nexists (a, b, c) \in Admin\_ASSOC: x = a \ \wedge$
    $\nexists (a, b, c, d) \in UA\_deny\_disjunctive: x = a \ \wedge$
    $\nexists (a, b, c, d) \in UA\_deny\_conjunctive: x = a \ \wedge$
    $\nexists (a, b, c, d) \in UA\_Admin\_deny\_disjunctive: x = a \ \wedge$
    $\nexists (a, b, c, d) \in UA\_Admin\_deny\_conjunctive: x = a$
```

```
        {
        $UA' = UA - \{x\}$
        $x' =$ Disinstantiation(x)
        }
    postconditions: $x \notin UA'$
```

DeleteO(x)

```
    preconditions: $x \in O \land x \in OA \land$
    $\nexists y \in OA: x \rightarrow^{+} y \land \nexists y \in PC: x \rightarrow^{+} y \land \nexists y \in OA: y \rightarrow^{+} x$
    $\nexists(a, b, c) \in$ ASSOC: $x = c \land$
    $\nexists(a, b, c) \in$ Admin_ASSOC: $x = c \land$
    // ensure no prohibitions exist that involve the object attribute
    $\nexists(a, b, c, d) \in$ U_deny_disjunctive: $(x \in c \lor x \in d) \land$
    $\nexists(a, b, c, d) \in$ P_deny_disjunctive: $(x \in c \lor x \in d) \land$
    $\nexists(a, b, c, d) \in$ U_deny_conjunctive: $(x \in c \land x \in d) \land$
    $\nexists(a, b, c, d) \in$ P_deny_conjunctive: $(x \in c \land x \in d) \land$
    $\nexists(a, b, c, d) \in$ U_Admin_deny_disjunctive: $(x \in c \lor x \in d) \land$
    $\nexists(a, b, c, d) \in$ P_Admin_deny_disjunctive: $(x \in c \lor x \in d) \land$
    $\nexists(a, b, c, d) \in$ U_Admin_deny_conjunctive: $(x \in c \land x \in d) \land$
    $\nexists(a, b, c, d) \in$ P_Admin_deny_conjunctive: $(x \in c \land x \in d) \land$
    $\nexists(a, b, c, d) \in$ UA_deny_disjunctive: $(x \in c \lor x \in d) \land$
    $\nexists(a, b, c, d) \in$ UA_deny_conjunctive: $(x \in c \land x \in d) \land$
    $\nexists(a, b, c, d) \in$ UA_Admin_deny_disjunctive: $(x \in c \lor x \in d) \land$
    $\nexists(a, b, c, d) \in$ UA_Admin_deny_conjunctive: $(x \in c \land x \in d)$
        {
        $O' = O - \{x\}$
        $OA' = OA - \{x\}$
        $x' =$ Disinstantiation(x)
        }
    postconditions: $x \notin O' \land x \notin OA'$
```

DeleteOA(x)

```
    preconditions: $x \in OA \land x \notin O \land$
    $\nexists y \in OA: x \rightarrow^{+} y \land \nexists y \in PC: x \rightarrow^{+} y \land \nexists y \in OA: y \rightarrow^{+} x \land$
    $\nexists(a, b, c) \in$ ASSOC: $x = c \land$
    $\nexists(a, b, c) \in$ Admin_ASSOC: $x = c \land$
    // ensure no prohibitions exist that involve the object attribute
    $\nexists(a, b, c, d) \in$ U_deny_disjunctive: $(x \in c \lor x \in d) \land$
    $\nexists(a, b, c, d) \in$ P_deny_disjunctive: $(x \in c \lor x \in d) \land$
    $\nexists(a, b, c, d) \in$ U_deny_conjunctive: $(x \in c \land x \in d) \land$
    $\nexists(a, b, c, d) \in$ P_deny_conjunctive: $(x \in c \land x \in d) \land$
    $\nexists(a, b, c, d) \in$ U_Admin_deny_disjunctive: $(x \in c \lor x \in d) \land$
    $\nexists(a, b, c, d) \in$ P_Admin_deny_disjunctive: $(x \in c \lor x \in d) \land$
    $\nexists(a, b, c, d) \in$ U_Admin_deny_conjunctive: $(x \in c \land x \in d) \land$
    $\nexists(a, b, c, d) \in$ P_Admin_deny_conjunctive: $(x \in c \land x \in d) \land$
    $\nexists(a, b, c, d) \in$ UA_deny_disjunctive: $(x \in c \lor x \in d) \land$
    $\nexists$(a, b, c, d) $\in$ UA_deny_conjunctive: (x $\in$ c $\land$ x $\in$ d) $\land$
    $\nexists$(a, b, c, d) $\in$ UA_Admin_deny_disjunctive: (x $\in$ c $\lor$ x $\in$ d) $\land$
    $\nexists$(a, b, c, d) $\in$ UA_Admin_deny_conjunctive: (x $\in$ c $\land$ x $\in$ d)
        {
        OA′ = OA – {x}
        x′ = Disinstantiation(x)
        }
    postconditions: x $\notin$ OA′
```

DeletePC(x)

```
    preconditions: x $\in$ PC $\land$ $\nexists$y $\in$ PE: y $\rightarrow^{+}$ x
        {
        PC′ = PC – {x}
        x′ = Disinstantiation(x)
        }
postconditions: x $\notin$ PC′
```

DeleteP(x)

```
    preconditions: x $\in$ P $\land$ $\nexists$u $\in$ U: u = process_user(x) $\land$
    // ensure no prohibitions exist that involve the process $\land$
    $\nexists$(a, b, c, d) $\in$ P_deny_disjunctive: a = x $\land$
    $\nexists$(a, b, c, d) $\in$ P_deny_conjunctive: a = x $\land$
    $\nexists$(a, b, c, d) $\in$ P_Admin_deny_disjunctive: a = x $\land$
    $\nexists$(a, b, c, d) $\in$ P_Admin_deny_conjunctive: a = x
        {
        P′ = P – {x}
        x′ = Disinstantiation(x)
        }
    postconditions: x $\notin$ P′
```

C.3 Relation Formation Routines

Besides ensuring that the arguments supplied for the formal parameters of a routine are valid, the preconditions specified below must also maintain certain model properties. An attempt to add tuple that already exists to a relation presents no problem, due to the set operation involved.

CreateAssign(x, y)

```
    preconditions: ((x $\in$ U $\land$ y $\in$ UA) $\lor$ (x $\in$ UA $\land$ y $\in$ UA) $\lor$ (x $\in$ UA $\land$ y $\in$ PC) $\lor$
    (x $\in$ O $\land$ y $\in$ OA) $\lor$ (x $\in$ OA $\land$ y $\in$ OA) $\lor$ (x $\in$ OA $\land$ y $\in$ PC)) $\land$ x $\neq$ y $\land$
        $\nexists$a sequence $s_1,s_2,...,s_n$ in PE: (n > 1 $\land$ $s_n \rightarrow s_1$ $\land$ ($s_i \rightarrow s_{i+1}$ for i = 1,2,...,n-1))
        {
        $\rightarrow' = \rightarrow \cup$ {(x, y)}    // union op precludes use of duplicate assignment precondition
        }
    postconditions: (x, y) $\in \rightarrow'$
```

```
CreatePUmapping(x, y)
    preconditions: $x \in P \land y \in U$
        {
        process_user′ = process_user ∪ {(x, y)}
        }
    postconditions: $(x, y) \in$ process_user′

CreateAssoc(x, y, z)
    preconditions: $x \in UA \land y \in Ops \land z \in OA$
        {
        ASSOC′ = ASSOC ∪ {(x, y, z)}
        }
    postconditions: $(x, y, z) \in ASSOC'$

CreateAdminAssoc(x, y, z)
    preconditions: $x \in UA \land y \in AOps \land z \in PE$
        {
        Admin_ASSOC′ = Admin_ASSOC ∪ {(x, y, z)}
        }
    postconditions: $(x, y, z) \in$ Admin_ASSOC′

CreateU_deny_disjunctive(w, x, y, z)
    preconditions: $w \in U \land x \in Ops \land y \in OAs \land z \in OACs$
        {
        U_deny_disjunctive′ = U_deny_disjunctive ∪ {(w, x, y, z)}
        }
    postconditions: $(w, x, y, z) \in$ U_deny_disjunctive′

CreateP_deny_disjunctive(w, x, y, z)
    preconditions: $w \in P \land x \in Ops \land y \in OAs \land z \in OACs$
        {
        P_deny_disjunctive′ = P_deny_disjunctive ∪ {(w, x, y, z)}
        }
    postconditions: $(w, x, y, z) \in$ P_deny_disjunctive′

CreateUA_deny_disjunctive(w, x, y, z)
    preconditions: $w \in UA \land x \in Ops \land y \in OAs \land z \in OACs$
        {
        UA_deny_disjunctive′ = UA_deny_disjunctive ∪ {(w, x, y, z)}
        }
    postconditions: $(w, x, y, z) \in$ UA_deny_disjunctive′

CreateU_Admin_deny_disjunctive(w, x, y, z)
    preconditions: $w \in U \land x \in Ops \land y \in OAs \land z \in OACs$
        {
        U_Admin_deny_disjunctive′ = U_Admin_deny_disjunctive ∪ {(w, x, y, z)}
        }
    postconditions: $(w, x, y, z) \in U\_Admin\_deny\_disjunctive'$
```

CreateP_Admin_deny_disjunctive(w, x, y, z)

```
    preconditions: $w \in P \land x \in Ops \land y \in OAs \land z \in OACs$
        {
        $P\_Admin\_deny\_disjunctive' = P\_Admin\_deny\_disjunctive \cup \{(w, x, y, z)\}$
        }
    postconditions: $(w, x, y, z) \in P\_Admin\_deny\_disjunctive'$
```

CreateUA_Admin_deny_disjunctive(w, x, y, z)

```
    preconditions: $w \in UA \land x \in Ops \land y \in OAs \land z \in OACs$
        {
        $UA\_Admin\_deny\_disjunctive' = UA\_Admin\_deny\_disjunctive \cup \{(w, x, y, z)\}$
        }
    postconditions: $(w, x, y, z) \in UA\_Admin\_deny\_disjunctive'$
```

The conjunctive forms of user, user attribute, and process-based prohibition formation are defined similarly to their disjunctive counterparts above.

CreateOblig(x, y, z)

```
    preconditions: $x \in U \land y \in Pattern \land z \in Response$
        {
        $Oblig' = Oblig \cup \{(x, y, z)\}$
        }
    postconditions: $(x, y, z) \in Oblig'$
```

CreateAdminOblig(x, y, z)

```
    preconditions: $x \in U \land y \in Pattern \land z \in Response$
        {
        $AdminOblig' = AdminOblig \cup \{(x, y, z)\}$
        }
    postconditions: $(x, y, z) \in AdminOblig'$
```

C.4 Relation Rescindment Routines

Besides ensuring that the arguments supplied for the formal parameters of a routine are valid, the preconditions must also maintain certain model properties. An attempt to delete a tuple that does not exist from a relation presents no problem, due to the set operation involved.

DeleteAssign(x, y)

```
    preconditions: $((x \in U \land y \in UA) \lor (x \in UA \land y \in UA) \lor (x \in UA \land y \in PC) \lor$
    $(x \in O \land y \in OA) \lor (x \in OA \land y \in OA) \lor (x \in OA \land y \in PC))$
        {
        $\rightarrow' = \rightarrow - \{(x, y)\}$
        }
    postconditions: $(x, y) \notin \rightarrow'$
```

```
DeletePUmapping(x, y)
    preconditions: $x \in P \land y \in U$
        {
        process_user′ = process_user – {(x, y)}
        }
    postconditions: $(x, y) \notin$ process_user ′

DeleteAssoc(x, y, z)
    preconditions: $x \in UA \land y \in Ops \land z \in OA$
        {
        ASSOC ′ = ASSOC – {(x, y, z)}
        }
    postconditions: $(x, y, z) \notin$ ASSOC ′

DeleteAdminAssoc(x, y, z)
    preconditions: $x \in UA \land y \in AOps \land z \in PE$
        {
        Admin_ASSOC′ = Admin_ASSOC – {(x, y, z)}
        }
    postconditions: $(x, y, z) \notin$ Admin_ASSOC′

DeleteU_deny_disjunctive(w, x, y, z)
    preconditions: $w \in U \land x \in Ops \land y \in OAs \land z \in OACs$
        {
        U_deny_disjunctive′ = U_deny_disjunctive – {(w, x, y, z)}
        }
    postconditions: $(w, x, y, z) \notin$ U_deny_disjunctive′

DeleteP_deny_disjunctive(w, x, y, z)
    preconditions: $w \in P \land x \in Ops \land y \in OAs \land z \in OACs$
        {
        P_deny_disjunctive′ = P_deny_disjunctive – {(w, x, y, z)}
        }
    postconditions: $(w, x, y, z) \notin$ P_deny_disjunctive′

DeleteUA_deny_disjunctive(w, x, y, z)
    preconditions: $w \in UA \land x \in Ops \land y \in OAs \land z \in OACs$
        {
        UA_deny_disjunctive′ = UA_deny_disjunctive – {(w, x, y, z)}
        }
    postconditions: $(w, x, y, z) \notin$ UA_deny_disjunctive′

DeleteU_Admin_deny_disjunctive(w, x, y, z)
    preconditions: $w \in U \land x \in Ops \land y \in OAs \land z \in OACs$
        {
```

```
        U_Admin_deny_disjunctive′ = U_Admin_deny_disjunctive – {(w, x, y, z)}
        }
    postconditions: (w, x, y, z) ∉ U_Admin_deny_disjunctive′
```

DeleteP_Admin_deny_disjunctive(w, x, y, z)

```
    preconditions: w ∈ P  ⋀  x ∈ Ops  ⋀  y ∈ OAs  ⋀  z ∈ OACs
        {
        P_Admin_deny_disjunctive′ = P_Admin_deny_disjunctive – {(w, x, y, z)}
        }
    postconditions: (w, x, y, z) ∉ P_Admin_deny_disjunctive′
```

DeleteUA_Admin_deny_disjunctive(w, x, y, z)

```
    preconditions: w ∈ UA  ⋀  x ∈ Ops  ⋀  y ∈ OAs  ⋀  z ∈ OACs
        {
        UA_Admin_deny_disjunctive′ = UA_Admin_deny_disjunctive – {(w, x, y, z)}
        }
    postconditions: (w, x, y, z) ∉ UA_Admin_deny_disjunctive′
```

The conjunctive forms of user, user attribute, and process-based prohibition rescindment are defined similarly to their disjunctive counterparts above.

DeleteOblig(x, y, z)

```
    preconditions: x ∈ U  ⋀  y ∈ Conditions  ⋀  z ∈ Response
        {
        OBLIG′ = OBLIG – {(x, y, z)}
        }
    postconditions: (x, y, z) ∉ OBLIG′
```

DeleteAdminOblig(x, y, z)

```
    preconditions: x ∈ U  ⋀  y ∈ Pattern  ⋀  z ∈ Response
        {
        Admin_OBLIG′ = Admin_OBLIG – {(x, y, z)}
        }
    postconditions: (x, y, z) ∉ Admin_OBLIG′
```

C.5 Relation Formation Commands

Besides ensuring that the arguments supplied for the formal parameters of a command are valid, the preconditions ensure that sufficient authority is held by the process attempting the access, including the absence of any prohibitions.[10] That is, reference mediation must be successfully carried out as a prerequisite to executing the body of an administrative command. Note that

[10] In modeling administrative commands, the process attempting access is represented as the first parameter of the command. It could have been modeled instead by eliminating the parameter and using in its place a semantic function (e.g., getProcessID()) that denotes the retrieval of the identifier of the process in question.

individual access rights beginning with the prefixes create- and delete- are represented by constants in the precondition formulas. That is, access rights are specified as text strings associated with the preconditions of administrative commands.

```
CreateUserAttributeInPolicyClass(p, x, pc)
    preconditions: $p \in P \wedge pc \in PC \wedge$
    $(create\text{-}ua, pc) \in APCap(p) \wedge (create\text{-}uapc, pc) \in APCap(p) \wedge$ // holds basic authorization
    $(p, create\text{-}ua, pc) \in NoDeny \wedge (p, create\text{-}uapc, pc) \in NoDeny)$   // no prohibitions apply
        {
        CreateUA(x)            // routine returns x; $UA' = UA \cup \{x\}$
        CreateAssign(x, pc)   // $\rightarrow' = \rightarrow \cup \{(x, pc)\}$
        }
     postconditions: $x \in UA' \wedge (x, pc) \in \rightarrow'$
```

```
AssignUserAttributeToPolicyClass(p, ua, pc)
    preconditions: $p \in P \wedge pc \in PC \wedge ua \in UA \wedge (ua, pc) \notin \rightarrow \wedge$
    $(((create\text{-}uapc, pc) \in APCap(p) \wedge ua \rightarrow^{+} pc \wedge (p, create\text{-}uapc, pc) \in NoDeny) \vee$
    $((create\text{-}uapc\text{-}from, ua), (create\text{-}uapc\text{-}to, pc) \in APCap(p) \wedge$
    $(p, create\text{-}uapc\text{-}from, ua), (p, create\text{-}uapc\text{-}to, pc) \in NoDeny))$
        {
        CreateAssign(ua, pc)
        }
     postconditions: $(ua, pc) \in \rightarrow'$
```

```
CreateUserAttributeInUserAttribute(p, x, ua)
    preconditions: $p \in P \wedge ua \in UA \wedge$
    $(create\text{-}ua, ua), (create\text{-}uaua, ua) \in APCap(p) \wedge$
    $(p, create\text{-}ua, ua), (p, create\text{-}uaua, ua) \in NoDeny$
        {
        CreateUA(x)
        CreateAssign(x, ua)
        }
     postconditions: $x \in UA' \wedge (x, ua) \in \rightarrow'$
```

```
AssignUserAttributeToUserAttribute(p, uafrom, uato)
    preconditions: $p \in P \wedge uafrom, uato \in UA \wedge (uafrom, uato) \notin \rightarrow \wedge$
    $(((create\text{-}uaua, uato) \in APCap(p) \wedge (p, create\text{-}uaua, uato) \in NoDeny \wedge$
    $\exists x \in PC: (uafrom \rightarrow^{+} x \wedge uato \rightarrow^{+} x)) \vee$
    $((create\text{-}uaua\text{-}from, uafrom), (create\text{-}uaua\text{-}to, uato) \in APCap(p) \wedge$
    $(p, create\text{-}uaua\text{-}from, uafrom), (p, create\text{-}uaua\text{-}to, uato) \in NoDeny))$
        {
        CreateAssign(uafrom, uato)
        }
     postconditions: $(uafrom, uato) \in \rightarrow'$
```

```
CreateUserInUserAttribute(p, x, ua)
    preconditions: $p \in P \wedge ua \in UA \wedge$
    $(create\text{-}u, ua), (create\text{-}uua, ua) \in APCap(p) \wedge$
    $(p, create\text{-}u, ua), (p, create\text{-}uua, ua) \in NoDeny$
        {
        CreateU(x)
        CreateAssign(x, ua)
        }
    postconditions: $x \in U' \wedge (x, ua) \in \rightarrow'$

AssignUserToUserAttribute(p, u, ua)
    preconditions: $p \in P \wedge u \in U \wedge ua \in UA \wedge (u, ua) \notin \rightarrow \wedge$
    $(((create\text{-}uua, ua) \in APCap(p) \wedge \exists x \in PC: (u \rightarrow^{+} x \wedge ua \rightarrow^{+} x)) \vee$
    $(create\text{-}uua\text{-}from, u), (create\text{-}uua\text{-}to, ua) \in APCap(p)) \wedge$
    $(p, create\text{-}uua\text{-}from, x), (p, create\text{-}uua\text{-}to, z) \in NoDeny$
        {
        CreateAssign(u, ua)
        }
    postconditions: $(u, ua) \in \rightarrow'$
```

Relation formation commands for object and object attribute assignments are defined similarly to those given above for user and user attributes.

```
CreateAssociation(p, x, y, z)
    preconditions: $p \in P \wedge x \in UA \wedge y \in Ops \wedge z \in OA \wedge (x, y, z) \notin ASSOC \wedge$
    $(create\text{-}assoc\text{-}from, x), (create\text{-}assoc\text{-}to, z) \in APCap(p) \wedge$   // holds basic authorization
    $(p, create\text{-}assoc\text{-}from, x), (p, create\text{-}assoc\text{-}to, z) \in NoDeny$   // no prohibitions apply
        {
        CreateAssoc(x, y, z)
        }
    postconditions: $(x, y, z) \in ASSOC'$

CreateAdministrativeAssociation(p, x, y, z)
    preconditions: $p \in P \wedge x \in UA \wedge y \in AOps \wedge z \in OA \wedge (x, y, z) \notin Admin\_ASSOC \wedge$
    $(create\text{-}admin\text{-}assoc\text{-}from, x), (create\text{-}admin\text{-}assoc\text{-}to, z) \in APCap(p) \wedge$
    $(p, create\text{-}admin\text{-}assoc\text{-}from, x), (p, create\text{-}admin\text{-}assoc\text{-}to, z) \in NoDeny$
        {
        CreateAdminAssoc(x, y, z)
        }
    postconditions: $(x, y, z) \in Admin\_ASSOC'$

CreateDisjunctiveUserProhibition(p, w, x, y, z)
    preconditions: $p \in P \wedge w \in U \wedge x \in Ops \wedge y \in OAs \wedge z \in OACs \wedge$
    $(w, x, y, z) \notin U\_deny\_disjunctive \wedge (create\text{-}deny\text{-}from, w) \in APCap(p) \wedge$
    $(p, create\text{-}deny\text{-}from, w) \in NoDeny \wedge$
    $\forall oa \in OAs:((create\text{-}deny\text{-}to, oa) \in APCap(p) \wedge (p, create\text{-}deny\text{-}to, oa) \in NoDeny) \wedge$
    ∀oac ∈ OACs:((create-deny-to, oac) ∈ APCap(p) ⋀ (p, create-deny-to, oac) ∈ NoDeny)
        {
        CreateU_deny_disjunctive(w, x, y, z)
        }
    postconditions: (w, x, y, z) ∈ U_deny_disjunctive′
```

CreateDisjunctiveProcessProhibition(p, w, x, y, z)

```
    preconditions: p, w ∈ P ⋀ x ∈ Ops ⋀ y ∈ OAs ⋀ z ∈ OACs ⋀
    (w, x, y, z) ∉ P_deny_disjunctive ⋀ (create-deny-from, w) ∈ APCap(p) ⋀
    (p, create-deny-from, w) ∈ NoDeny ⋀
    ∀oa ∈ OAs:((create-deny-to, oa) ∈ APCap(p) ⋀ (p, create-deny-to, oa) ∈ NoDeny) ⋀
    ∀oac ∈ OACs:((create-deny-to, oac) ∈ APCap(p) ⋀ (p, create-deny-to, oac) ∈ NoDeny)
        {
        CreateP_deny_disjunctive(w, x, y, z)
        }
    postconditions: (w, x, y, z) ∈ P_deny_disjunctive′
```

CreateDisjunctiveUserAttributeProhibition(p, w, x, y, z)

```
    preconditions: p ∈ P ⋀ w ∈ UA ⋀ x ∈ Ops ⋀ y ∈ OAs ⋀ z ∈ OACs ⋀
    (w, x, y, z) ∉ UA_deny_disjunctive ⋀ (create-deny-from, w) ∈ APCap(p) ⋀
    (p, create-deny-from, w) ∈ NoDeny ⋀
    ∀oa ∈ OAs:((create-deny-to, oa) ∈ APCap(p) ⋀ (p, create-deny-to, oa) ∈ NoDeny) ⋀
    ∀oac ∈ OACs:((create-deny-to, oac) ∈ APCap(p) ⋀ (p, create-deny-to, oac) ∈ NoDeny)
        {
        CreateUA_deny_disjunctive(w, x, y, z)
        }
    postconditions: (w, x, y, z) ∈ UA_deny_disjunctive′
```

CreateAdministrativeDisjunctiveUserProhibition(p, w, x, y, z)

```
    preconditions: p ∈ P ⋀ w ∈ U ⋀ x ∈ AOps ⋀ y ∈ OAs ⋀ z ∈ OACs ⋀
    (w, x, y, z) ∉ U_Admin_deny_disjunctive ⋀ (create-admin-deny-from, w) ∈ APCap(p) ⋀
    (p, create-admin-deny-from, w) ∈ NoDeny ⋀
    ∀oa ∈ OAs:((create-admin-deny-to, oa) ∈ APCap(p) ⋀
    (p, create-admin-deny-to, oa) ∈ NoDeny) ⋀
    ∀oac ∈ OACs:((create-admin-deny-to, oac) ∈ APCap(p) ⋀
    (p, create-admin-deny-to, oac) ∈ NoDeny)
        {
        CreateU_Admin_deny_disjunctive(w, x, y, z)
        }
    postconditions: (w, x, y, z) ∈ U_Admin_deny_disjunctive′
```

CreateAdministrativeDisjunctiveProcessProhibition(p, w, x, y, z)

```
    preconditions: p, w ∈ P ⋀ x ∈ Ops ⋀ y ∈ OAs ⋀ z ∈ OACs ⋀
    (w, x, y, z) ∉ P_Admin_deny_disjunctive ⋀ (create-admin-deny-from, w) ∈ APCap(p) ⋀
    (p, create-admin-deny-from, w) ∈ NoDeny ⋀
    ∀oa ∈ OAs:((create-admin-deny-to, oa) ∈ APCap(p) ⋀
    (p, create-admin-deny-to, oa) ∈ NoDeny)  ∧
    ∀oac ∈ OACs:((create-admin-deny-to, oac) ∈ APCap(p)  ∧
    (p, create-admin-deny-to, oac) ∈ NoDeny)
        {
        CreateP_Admin_deny_disjunctive(w, x, y, z)
        }
    postconditions: (w, x, y, z) ∈ P_Admin_deny_disjunctive′
```

CreateAdministrativeDisjunctiveUserAttributeProhibition(p, w, x, y, z)

```
    preconditions: p ∈ P  ∧  w ∈ UA  ∧  x ∈ AOps  ∧  y ∈ OAs  ∧  z ∈ OACs  ∧
    (w, x, y, z) ∉ UA_Admin_deny_disjunctive  ∧  (create-admin-deny-from, w) ∈ APCap(p)  ∧
    (p, create-admin-deny-from, w) ∈ NoDeny  ∧
    ∀oa ∈ OAs:((create-admin-deny-to, oa) ∈ APCap(p)  ∧
    (p, create-admin-deny-to, oa) ∈ NoDeny)  ∧
    ∀oac ∈ OACs:((create-admin-deny-to, oac) ∈ APCap(p)  ∧
    (p, create-admin-deny-to, oac) ∈ NoDeny)
        {
        CreateUA_Admin_deny_disjunctive(w, x, y, z)
        }
    postconditions: (w, x, y, z) ∈ UA_Admin_deny_disjunctive′
```

The conjunctive forms of user, user attribute, and process-based prohibition formation are defined similarly to their disjunctive counterparts above.

EvalPattern(eventpattern), returns Boolean

```
        {
        /* A semantic function that designates the result of an evaluation of the correctness
        of a logical expression that describes an event pattern involving the policy elements
        and relations of the PM (provided as an input string).  The syntax of the logical
        expression and the details of the evaluation algorithm are not prescribed by the PM,
        but ideally should be capable of expressing and checking first-order predicate calculus
        formulas. */
        }
```

EvalResponse(response), returns Boolean

```
        {
        /* A semantic function that designates the result of an evaluation of the correctness
        of the syntax of an obligation's response (provided as an input string).  The syntax
        of the response and its constituent administrative routine invocations and the details
        of the evaluation algorithm are not prescribed by the PM. */
        }
```

Note Concerning Obligations:

Obligations have unique characteristics that distinguish them from other relations. The logical expression of the event pattern cannot be fully evaluated at creation time, since variables used in the expression may refer to the value of items in the event context or to policy elements and relations that may or may not exist at match time. However, some syntax checks can be made to filter out incorrect expressions and verify that a string supplied as an event pattern is well-formed with respect to its respective grammar. A similar situation applies to the expression of the obligation response and to the arguments and invocation of the administrative routines that make up the response.

The creation of obligations is modeled with two administrative commands: CreateObligation and CreateAdministrativeObligation. The preconditions for each require that the event pattern supplied meets the formal grammar rules for the language used to specify logical expressions (i.e., the EvalPattern function returns True). Similarly, the preconditions for each also require that the response meets the formal grammar rules for the language used to specify administrative routine invocations (i.e., the EvalResponse function returns True). The semantics for these routines describe the preservation of the partially checked event pattern and response statements, for later use in event context matching and response initiation. The user for which the obligation is created is also preserved to allow, at the time a match to the obligation occurs, verification that the user has sufficient authorization to execute the response.

Because obligation event patterns and responses fall largely outside the model specification, it is not possible to place restrictions on model elements that affect the authorization needed to define an obligation. The authorization to create obligations is essentially an all or none proposition, since the scope of user's authorization cannot be limited to specific elements of policy. An association that grants authority to form obligations with respect to a specific policy element, also grants authority to form obligations with any other policy element.

CreateObligation(x, y, z)

```
/* The pattern of an obligation created with this command will be matched only when an
event for a read or write operation occurs. */
preconditions: $x \in P \ \wedge\ y \in Pattern \ \wedge\ z \in Response \ \wedge$
$(process\_user(x), y, z) \notin OBLIG \ \wedge\ EvalPattern(y) \ \wedge\ EvalResponse(z) \ \wedge$
// ensure that the process has authorization to create the obligation
$\exists pe \in PEs: ((create\text{-}deny, pe) \in PCap(p) \ \wedge\ (p, create\text{-}deny, pe) \in NoDeny)$
    {
    CreateOblig(process_user(x), y, z)
    }
postconditions: $(process\_user(x), y, z) \in OBLIG'$
```

CreateAdministrativeObligation(x, y, z)

```
/* The pattern of an administrative obligation created with this command will be matched
only when an event for an administrative action occurs. */
preconditions: $x \in P \wedge\ y \in Pattern \ \wedge\ z \in Response \ \wedge$
$(process\_user(x), y, z) \notin Admin\_OBLIG \ \wedge\ EvalPattern(y) \ \wedge\ EvalResponse(z) \ \wedge$
// ensure that the process has authorization to create the obligation
$\exists pe \in PEs: ((create\text{-}admin\text{-}deny, pe) \in PCap(p) \ \wedge\ (p, create\text{-}admin\text{-}deny, pe) \in NoDeny)$
    {
    CreateAdminOblig(process_user(x), y, z)
    }
postconditions: $(process\_user(x), y, z) \in Admin\_OBLIG'$
```

C.6 Relation Rescindment Commands

Besides ensuring that the arguments supplied for the formal parameters of a command are valid, the preconditions ensure that sufficient authority is held by the process attempting the access, including the absence of prohibitions. As in the previous section, individual access rights beginning with the prefixes create- and delete- are represented by constants in the precondition formulas.

For commands that delete assignment between policy elements, the preconditions also ensure that the contained policy element is not left isolated. Note that for rescindment of relations to proceed correctly, the following rule must be observed: before attempting deletion of a policy element along with an associated assignment, any outstanding associations or prohibitions must first be deleted.

```
DeleteUserInUserAttribute(p, u, ua)
    /* check type of supplied variables, that an assignment exists, and that sufficient authority is
    held to delete both the assignment and the contained u */
    preconditions: $p \in P \land u \in U \land ua \in UA \land (u, ua) \in \rightarrow \land$        // verify input variables
    $(delete\text{-}u, ua) \in APCap(p) \land (p, delete\text{-}u, ua) \in NoDeny \land$        // verify authority exists
    $(((delete\text{-}uua, ua) \in APCap(p) \land (p, delete\text{-}uua, ua) \in NoDeny)) \lor$
    $((delete\text{-}uua\text{-}from, u), (delete\text{-}uua\text{-}to, ua) \in APCap(p) \land$
    $(p, delete\text{-}uua\text{-}from, u), (p, delete\text{-}uua\text{-}to, ua) \in NoDeny))$
        {
        DeleteAssign(u, ua)
        DeleteU(u)              // routine fails if any relations exist that involve u
        }
     postconditions: $u \notin UA' \land (u, ua) \notin \rightarrow'$

DisassignUserToUserAttribute(p, u, ua)
    /* check the type of supplied variables, that an assignment exists between the uas, that
    sufficient authority is held to delete the assignment, and that PC reachability is maintained */
    preconditions: $p \in P \land u \in U \land ua \in UA \land (u, ua) \in \rightarrow \land$
    $\exists x \in UA: (x \neq ua \land u \rightarrow^{+} x) \land$  // ensures that u is assigned to some other PE
    $(((delete\text{-}uua, ua) \in APCap(p) \land (p, delete\text{-}uua, ua) \in NoDeny) \lor$
    $((delete\text{-}uua\text{-}from, u), (delete\text{-}uua\text{-}to, ua) \in APCap(p) \land$
    $(p, delete\text{-}uua\text{-}from, u), (p, delete\text{-}uua\text{-}to, ua) \in NoDeny))$
        {
        DeleteAssign(u, ua)
        }
     postconditions: $(u, ua) \notin \rightarrow' \land \exists x \in PC: u \rightarrow^{+} x$

DeleteUserAttributeInUserAttribute(p, uafrom, uato)
    /* check type of supplied variables, check that an assignment exists between the uas, and
    check that sufficient authority is held to delete both the assignment and the contained ua */
    preconditions: $p \in P \land uafrom, uato \in UA \land (uafrom, uato) \in \rightarrow \land$
    $(delete\text{-}ua, uato) \in APCap(p) \land (p, delete\text{-}ua, uato) \in NoDeny \land$
    $(((delete\text{-}uaua, ua) \in APCap(p) \land (p, delete\text{-}uaua, uato) \in NoDeny) \lor$
    ((delete-uaua-from, uafrom), (delete-uaua-to, uato) $\in$ APCap(p) $\wedge$
    (p, delete-uaua-from, uafrom), (p, delete-uaua-to, uato) $\in$ NoDeny))
        {
        DeleteAssign(uafrom, uato)
        DeleteUA(uafrom)          // routine fails if any relations exist that involve uafrom
        }
     postconditions: uafrom $\notin$ UA′ $\wedge$ (uafrom, uato) $\notin$ $\rightarrow'$
```

DisassignUserAttributeToUserAttribute(p, uafrom, uato)

```
/* check the type of supplied variables, that an assignment exists between the uas, that
sufficient authority is held to delete the assignment, and that PC reachability is maintained */
preconditions: p $\in$ P $\wedge$ uafrom, uato $\in$ UA $\wedge$ (uafrom, uato) $\in \rightarrow$ $\wedge$
$\exists$x $\in$ PE: (x $\neq$ uato $\wedge$ uafrom $\rightarrow$ x) $\wedge$ // ensures that uafrom is assigned to some other PE
(((delete-uaua, uato) $\in$ APCap(p) $\wedge$ (p, delete-uaua, uato) $\in$ NoDeny) $\vee$
((delete-uaua-from, uafrom), (delete-uaua-to, uato) $\in$ APCap(p) $\wedge$
(p, delete-uaua-from, uafrom), (p, delete-uaua-to, uato) $\in$ NoDeny))
    {
    DeleteAssign(uafrom, uato)
    }
 postconditions: (uafrom, uato) $\notin$ $\rightarrow'$ $\wedge$ $\exists$x $\in$ PC: uafrom $\rightarrow^{+}$ x
```

DeleteUserAttributeInPolicyClass(p, ua, pc)

```
/* check type of supplied variables, check that an assignment exists between the ua and pc,
and that sufficient authority is held to delete both the assignment and the contained ua */
preconditions: p $\in$ P $\wedge$ ua $\in$ UA $\wedge$ pc $\in$ PC $\wedge$ (ua, pc) $\in \rightarrow$ $\wedge$
(delete-ua, pc) $\in$ APCap(p) $\wedge$ (p, delete-ua, pc) $\in$ NoDeny $\wedge$
(((delete-uapc, pc) $\in$ APCap(p) $\wedge$ (p, delete-uapc, pc) $\in$ NoDeny) $\vee$
((delete-uapc-from, ua), (delete-uapc-to, pc) $\in$ APCap(p) $\wedge$
(p, delete-uapc-from, ua), (p, delete-uapc-to, pc) $\in$ NoDeny))
    {
    DeleteAssign(ua, pc)
    DeleteUA(ua)          // routine fails if any relations exist that involve ua
    }
 postconditions: ua $\notin$ UA′ $\wedge$ (ua, pc) $\notin$ $\rightarrow'$
```

DisassignUserAttributeToPolicyClass(p, ua, pc)

```
/* check the type of supplied variables, that an assignment exists, that sufficient authority is
held to delete the assignment, and that PC-reachability is maintained*/
preconditions: p $\in$ P $\wedge$ ua $\in$ UA $\wedge$ pc $\in$ PC $\wedge$ (ua, pc) $\in \rightarrow$ $\wedge$
$\exists$x $\in$ PC: (x $\neq$ pc $\wedge$ ua $\rightarrow^{+}$ x) $\wedge$ // ensures that the ua can reach some other PC
(((delete-uapc, pc) $\in$ APCap(p) $\wedge$ (p, delete-uapc, pc) $\in$ NoDeny) $\vee$
((delete-uapc-from, ua), (delete-uapc-to, pc) $\in$ APCap(p) $\wedge$
(p, delete-uapc-from, ua), (p, delete-uapc-to, pc) $\in$ NoDeny))
    {
    DeleteAssign(ua, pc)
```

```
        }
    postconditions: (ua, pc) ∉ →′ ∧ ∃x ∈ PC: ua →⁺ x "
```

Relation recindment commands for object and object attribute assignments are defined similarly to those given above for user and user attributes.

DeleteAssociation(p, x, y, z)

```
  preconditions: p ∈ P ∧ x ∈ UA ∧ y ∈ Ops ∧ z ∈ OA ∧ (x, y, z) ∈ ASSOC ∧
  (delete-assoc-from, x), (delete-assoc-to, z) ∈ APCap(p) ∧
  (p, delete-assoc-from, x), (p, delete-assoc-to, z) ∈ NoDeny
      {
      DeleteAssoc(x, y, z)
      }
  postconditions: (x, y, z) ∉ ASSOC′
```

DeleteAdministrativeAssociation(p, x, y, z)

```
  preconditions: p ∈ P ∧ x ∈ UA ∧ y ∈ AOps ∧ z ∈ OA ∧ (x, y, z) ∈ Admin_ASSOC ∧
  (delete-admin-assoc-from, x), (delete-admin-assoc-to, z) ∈ APCap(p) ∧
  (p, delete-admin-assoc-from, x), (p, delete-admin-assoc-to, z) ∈ NoDeny
      {
      DeleteAdminAssoc(x, y, z)
      }
  postconditions: (x, y, z) ∉ Admin_ASSOC′
```

DeleteDisjunctiveUserProhibition(p, w, x, y, z)

```
  preconditions: p ∈ P ∧ w ∈ U ∧ x ∈ Ops ∧ y ∈ OAs ∧ z ∈ OACs ∧
  (w, x, y, z) ∈ U_deny_disjunctive ∧ (delete-deny-from, w) ∈ APCap(p) ∧
  (p, delete-deny-from, w) ∈ NoDeny ∧
  ∀oa ∈ OAs:((delete-deny-to, oa) ∈ APCap(p) ∧ (p, delete-deny-to, oa) ∈ NoDeny) ∧
  ∀oac ∈ OACs:((delete-deny-to, oac) ∈ APCap(p) ∧ (p, delete-deny-to, oac) ∈ NoDeny)
      {
      DeleteU_deny_disjunctive(w, x, y, z)
      }
  postconditions: (w, x, y, z) ∉ U_deny_disjunctive′
```

DeleteDisjunctiveProcessProhibition(p, w, x, y, z)

```
  preconditions: p, w ∈ P ∧ x ∈ Ops ∧ y ∈ OAs ∧ z ∈ OACs ∧
  (w, x, y, z) ∈ P_deny_disjunctive ∧ (delete-deny-from, w) ∈ APCap(p) ∧
  (p, delete-deny-from, w) ∈ NoDeny ∧
  ∀oa ∈ OAs:((delete-deny-to, oa) ∈ APCap(p) ∧ (p, delete-deny-to, oa) ∈ NoDeny) ∧
  ∀oac ∈ OACs:((delete-deny-to, oac) ∈ APCap(p) ∧ (p, delete-deny-to, oac) ∈ NoDeny)
      {
      DeleteP_deny_disjunctive(w, x, y, z)
      }
  postconditions: (w, x, y, z) ∉ P_deny_disjunctive′
```

DeleteDisjunctiveUserAttributeProhibition(p, w, x, y, z)

```
    preconditions: p ∈ P ∧ w ∈ UA ∧ x ∈ Ops ∧ y ∈ OAs ∧ z ∈ OACs ∧
    (w, x, y, z) ∈ UA_deny_disjunctive ∧ (delete-deny-from, w) ∈ APCap(p) ∧
    (p, delete-deny-from, w) ∈ NoDeny ∧
    ∀oa ∈ OAs:((delete-deny-to, oa) ∈ APCap(p) ∧ (p, delete-deny-to, oa) ∈ NoDeny) ∧
    ∀oac ∈ OACs:((delete-deny-to, oac) ∈ APCap(p) ∧ (p, delete-deny-to, oac) ∈ NoDeny)
        {
        DeleteUA_deny_disjunctive(w, x, y, z)
        }
    postconditions: (w, x, y, z) ∉ UA_deny_disjunctive′
```

DeleteAdministrativeDisjunctiveUserProhibition(p, w, x, y, z)

```
    preconditions: p ∈ P ∧ w ∈ U ∧ x ∈ AOps ∧ y ∈ OAs ∧ z ∈ OACs ∧
    (w, x, y, z) ∈ U_Admin_deny_disjunctive ∧ (delete-admin-deny-from, w) ∈ APCap(p) ∧
    (p, delete-admin-deny-from, w) ∈ NoDeny ∧
    ∀oa ∈ OAs:((delete-admin-deny-to, oa) ∈ APCap(p) ∧
    (p, delete-admin-deny-to, oa) ∈ NoDeny) ∧
    ∀oac ∈ OACs:((delete-admin-deny-to, oac) ∈ APCap(p) ∧
    (p, delete-deny-to, oac) ∈ NoDeny)
        {
        DeleteU_Admin_deny_disjunctive(w, x, y, z)
        }
    postconditions: (w, x, y, z) ∉ U_Admin_deny_disjunctive′
```

DeleteAdministrativeDisjunctiveProcessProhibition(p, w, x, y, z)

```
    preconditions: p, w ∈ P ∧ x ∈ Ops ∧ y ∈ OAs ∧ z ∈ OACs ∧
    (w, x, y, z) ∈ P_Admin_deny_disjunctive ∧ (delete-admin-deny-from, w) ∈ APCap(p) ∧
    (p, delete-admin-deny-from, w) ∈ NoDeny ∧
    ∀oa ∈ OAs:((delete-admin-deny-to, oa) ∈ APCap(p) ∧
    (p, delete-admin-deny-to, oa) ∈ NoDeny) ∧
    ∀oac ∈ OACs:((delete-admin-deny-to, oac) ∈ APCap(p) ∧
    (p, delete-deny-to, oac) ∈ NoDeny)
        {
        DeleteP_Admin_deny_disjunctive(w, x, y, z)
        }
    postconditions: (w, x, y, z) ∉ P_Admin_deny_disjunctive′
```

DeleteAdministrativeDisjunctiveUserAttributeProhibition(p, w, x, y, z)

```
    preconditions: p ∈ P ∧ w ∈ UA ∧ x ∈ AOps ∧ y ∈ OAs ∧ z ∈ OACs ∧
    (w, x, y, z) ∈ UA_Admin_deny_disjunctive ∧ (delete-admin-deny-from, w) ∈ APCap(p) ∧
    (p, delete-admin-deny-from, w) ∈ NoDeny ∧
    ∀oa ∈ OAs:((delete-admin-deny-to, oa) ∈ APCap(p) ∧
    (p, delete-admin-deny-to, oa) ∈ NoDeny) ∧
    ∀oac ∈ OACs:((delete-admin-deny-to, oac) ∈ APCap(p) ∧
    (p, delete-deny-to, oac) ∈ NoDeny)
        {
```

```
        DeleteUA_Admin_deny_disjunctive(w, x, y, z)
        }
    postconditions: (w, x, y, z) ∉ UA_Admin_deny_disjunctive′
```

The conjunctive forms of user, user attribute, and process-based prohibition rescindment are defined similarly to their disjunctive counterparts above.

DeleteObligation(x, y, z)

```
    preconditions: x ∈ P  ⋀  y ∈ Pattern  ⋀  z ∈ Response  ⋀
    (process_user(x), y, z) ∈ OBLIG  ⋀
    // ensure that the process has authorization to delete the obligation
    ∃pe ∈ PEs:((delete-deny, pe) ∈ PCap(p)  ⋀  (p, delete-deny, pe) ∈ NoDeny)
        {
        DeleteOblig(process_user(x), y, z)
        }
    postconditions: (process_user(x), y, z) ∉ OBLIG′
```

DeleteAdministrativeObligation(x, y, z)

```
    preconditions: x ∈ P ⋀  y ∈ Pattern  ⋀  z ∈ Response  ⋀
    (process_user(x), y, z) ∈ Admin_OBLIG  ⋀
    // ensure that the process has authorization to delete the obligation
    ∃pe ∈ PEs:((delete-admin-deny, pe) ∈ PCap(p)  ⋀  (p, delete-admin-deny, pe) ∈ NoDeny)
        {
        CreateAdminOblig(process_user(x), y, z)
        }
    postconditions: (process_user(x), y, z) ∉ Admin_OBLIG′
```

Appendix D—Defining Personas

The idea behind personas is that in many circumstances, it is desirable to have certain individuals act in two different, mutually exclusive modes of operation: that of an administrator and that of a user. However, assigning an individual two distinct user identities, one for each mode of operation, takes an important aspect of policy management outside of the policy specification, which eventually could lead to problems as policy evolves and personnel changes occur. It would be preferable to accommodate this type of functionality explicitly within the policy specification. Two general approaches are possible, which are discussed below.

The first approach is to incorporate the functionality of personas into the PM model. This can be done by defining an extension to the model, which would allow a user with sufficient authorization to change its assignment to a user attribute representing one mode of operation, to a different user attribute that represents the other mode of operation. The extension described here provides a straightforward example of this approach. It entails defining a new access right, reassign-user, for the administrative action, together with an administrative command that carries out the indicated action. To grant the requisite authority, the system administrator has only to establish administrative associations that allow the user in question to switch between each user attribute that serves as one of its personas. With that authority in place, the user can initiate the administrative command via an administrative access request to cause its assignment to change.

The administrative command below, SwitchAssignmentBetweenUAs, specifies the creation of an assignment from the user u to the new user attribute uanew and the deletion of the assignment from the user u to the current user attribute uacurrent. The syntax and notation for the command follows that described in Appendix C.

```
SwitchAssignmentBetweenUAs (p, u, uacurrent, uanew)
    Preconditions: $p \in P \land u \in U \land$ uacurrent, uanew $\in UA \land$
    // u is assigned only to uacurrent and the process is requesting access for u
    (u, uacurrent) $\in \rightarrow \land$ (u, uanew) $\notin \rightarrow \land$ u = process_user(p) $\land$
    // u must hold reassign-user authorization over uacurrent and uanew
    (reassign-user, uacurrent), (reassign-user, uanew) $\in$ APCap(p)
        {
        CreateAssign( u, uanew)
        DeleteAssign (u, uacurrent)
        }
    Postconditions: (u, uacurrent) $\notin \rightarrow \land$ (u, uanew) $\in \rightarrow$
```

The solution is general purpose. Persona attributes are not restricted to switching a user between non-administrative and administrative modes of operation, although that is a common use. They can also apply to switching a user solely between either administrative modes or non-administrative modes of operation. Moreover, the approach works with not only two user persona attributes, each representing an alternative mode of operation for the user, but also any number of such attributes.

A simple example based on Figure 5 illustrates usage of the extension more concretely. For the policy specified, u_2 and u_4 are presumed to represent two personas for a single individual. Applying the above solution requires the following steps:

- Create the association (OUadmin, reassign-user, Group2) ∈ Admin_ASSOC, which grants users assigned to OUadmin the authorization to change that assignment to Group2.
- Create the association (OUadmin, reassign-user, OUadmin) ∈ Admin_ASSOC, which grants users assigned to OUadmin the authorization to change that assignment away from OUadmin.
- Create the association (Group2, reassign-user, OUadmin) ∈ Admin_ASSOC, which grants users assigned to Group2 the authority to change that assignment to OUadmin.
- Create the association (Group2, reassign-user, Group2) ∈ Admin_ASSOC, which grants users assigned to Group2 the authorization to change that assignment away from Group2.
- Delete the user policy element u_4 and its assignment to OUadmin, since they are no longer needed.

An individual logging in as u_2 for the first time defaults to the persona attribute for which u_2 is assigned (i.e., Group2). The user can switch to the other persona attribute by issuing the administrative access request <switch-assignment, [Group2, OrgUnitAdmin]>$_p$, which in turn results in the execution of the administrative command SwitchAssignmentBetweenUAs(p, u_2, Group2, OrgUnitAdmin) to carry out the action. The individual can switch back to the Group2 persona attribute by issuing a similar access request with the order of the arguments reversed.

Note that if multiple users are expected to be assigned to a user attribute designated as a persona, but not all of them require the ability to switch among personas, a slight adjustment can be made to the authorization graph to accommodate the situation. Adding a container, such as persona-ua, and assigning it to the user attribute ua allows the container to be substituted in lieu of ua as the basis for reassign-user associations and persona reassignment requests for the user in question and any other users that operate via the same set of personas. In the Figure 5 example, for instance, if users that do not perform administrative functions are expected to be assigned to Group2, a new user attribute persona-Group2 can be created and assigned to the Group2 user attribute, and the two administrative associations can be redefined with persona-Group2 used in place of Group2.

The second way to accommodate personas is through obligations. This approach would involve defining a command very similar to SwitchAssignmentBetweenUAs, but with different preconditions appropriate for use in an obligation. It would also require assigning the appropriate authorization to a user to enable the triggering of the obligation and execution of the command. In this case, however, no new authorization like reassign-user would apply; instead, an existing core access right would apply, such as reading from or writing to some file created for this purpose. Exercising the assigned authority to perform input or output to a file would correspond to a specific switch in user assignments.

Using the Figure 5 policy as an example again, files called switch-to-OUadmin and switch-to-Group2 could be defined and assigned to the Projects container. An obligation could be defined such that a user in Group2 reading the switch-to-OUadmin file would trigger an obligation that causes the user to be resigned to OUadmin. Likewise, another obligation could be defined such that a user in OUadmin reading the switch-to-Group2 file would trigger an obligation that causes the user to be reassigned to Group2.

With this approach, issuing an access request to read one of the designated files has a similar effect to issuing an administrative access requested to switch assignments in the other—they both cause the administrative command to be executed to carry out the change in assignments. As with the earlier approach, if it is intended to assign multiple users to a user attribute designated as a persona, but only some of them require the ability to switch among personas, the same adjustment to the authorization graph can be applied to accommodate the situation when obligations are used.

While personas can be instituted employing obligations, the approach is less direct and more cumbersome than incorporating personas into the model. For example, two or more persona attributes can be supported for a user or class or users, but each persona attribute would require the definition of an obligation and a special-purpose file to trigger its respective obligation. Nevertheless, for policies where only one or two classes of administrators are needed, obligations can provide a useful means to support personas.