NISTIR 7987
Revision 1

Policy Machine: Features, Architecture, and Specification

David Ferraiolo
Serban Gavrila
Wayne Jansen

This publication is available free of charge from:
http://dx.doi.org/10.6028/NIST.IR.7987r1

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

NISTIR 7987
Revision 1

Policy Machine: Features, Architecture, and Specification

David Ferraiolo
Serban Gavrila
Computer Security Division
Information Technology Laboratory

Wayne Jansen
Bayview Behavioral Consulting
Point Roberts, WA

This publication is available free of charge from:
http://dx.doi.org/10.6028/NIST.IR.7987r1

October 2015

U.S. Department of Commerce
Penny Pritzker, Secretary

National Institute of Standards and Technology
Willie E. May, Under Secretary of Commerce for Standards and Technology and Director

National Institute of Standards and Technology Internal Report 7987 Revision 1
119 pages (October 2015)

This publication is available free of charge from:
http://dx.doi.org/10.6028/NIST.IR.7987r1

Comments on this publication may be submitted to:

National Institute of Standards and Technology
Attn: Computer Security Division, Information Technology Laboratory
100 Bureau Drive (Mail Stop 8930) Gaithersburg, MD 20899-8930

Reports on Computer Systems Technology

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; Attribute Based Access Control (ABAC); access control policies; policy enforcement; privilege management; computer security

Acknowledgements

The authors, David Ferraiolo and Serban Gavrila of the National Institute of Standards and Technology, and Wayne Jansen of Bayview Behavioral Consulting, wish to thank colleagues at NIST and elsewhere, particularly Roger Cummings, 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.

Note to Readers

This version (Revision 1) of NISTIR 7987 revises the original publication (dated May 2014). Changes were made to reorganize and improve the content of the report, incorporate additional material, and bring the report into close alignment with the terminology and notation of the emerging NGAC-GOADS standard. This report, while aligned with NGAC-GOADS, provides additional details and background material that are intended to aid readers in understanding the function and operation of the access control model and provide insight into its implementation.

Table of Contents

List of Figures

List of Tables

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 work 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 [Tri04]. 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 framework, 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 the PM framework was reified 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 control features offered by the PM and using them to meet the access control needs for objects within the layer of abstraction the application provides. That is, the PM framework affords applications a single generic facility that can not only enforce access control policies comprehensively across distributed and centralized operating environments, but also subsume aspects involving the characterization, distribution, and control of implemented capabilities, resulting in a dramatic alleviation of many of the administrative, policy enforcement, data interoperability, and usability challenges faced by enterprises today.

1.1 Purpose and Scope

The purpose of this Internal Report (IR) 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 Standards Alignment

NIST and other members of an Ad Hoc working group within the International Committee for Information Technology Standards (INCITS) body are developing a multi-part access control standard, under the title of "Next Generation Access Control" (NGAC). This work is being conducted under three sub-projects:

- Project 2193–D: Next Generation Access Control – Implementation Requirements, Protocols and API Definitions.
- Project 2194–D: Next Generation Access Control – Functional Architecture.
- Project 2195–D: Next Generation Access Control – Generic Operations and Abstract Data Structures.

The Policy Machine's architecture and specifications form the basis for the NGAC work within INCITS. An initial standard from Project 2194-D was published in 2013 and is now available from the American National Standards Institute (ANSI) estandards store as INCITS 499 – NGAC Functional Architecture (NGAC–FA) [ANSI13]. At the time of publication of this IR, a draft proposed standard for Project 2195–D: NGAC Generic Operations & Abstract Data Structures (NGAC-GOADS) had completed the Public Review stage of the approval process and entered into final balloting for approval as a ANSI standard [ANSI15]. Publication of the standard expected to occur in the fall of 2015.

Although this IR is in a self-consistent state and most aspects are consistent with ANSI INCITS 499 and NGAC-GOADS, some small differences do exist. The most significant of these is the grammar used in obligations, which is not a normative part of NGAC-GOADS.

1.3 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 more deeply 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 in such situations.

- 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 and summary 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 summarizes the mathematical notation used and the principal abstractions defined in the report. Appendix C provides a list of the core administrative commands for the PM model and their semantic description. Appendix D specifies the administrative routines that constitute the core services of the PM model. Appendix E outlines three different approaches to support 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 user is unable to access objects directly, and instead must perform accesses through a subject. A subject represents a user and any system process or entity that acts on behalf of a user. Subjects are 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 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 [Lam71, 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 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.

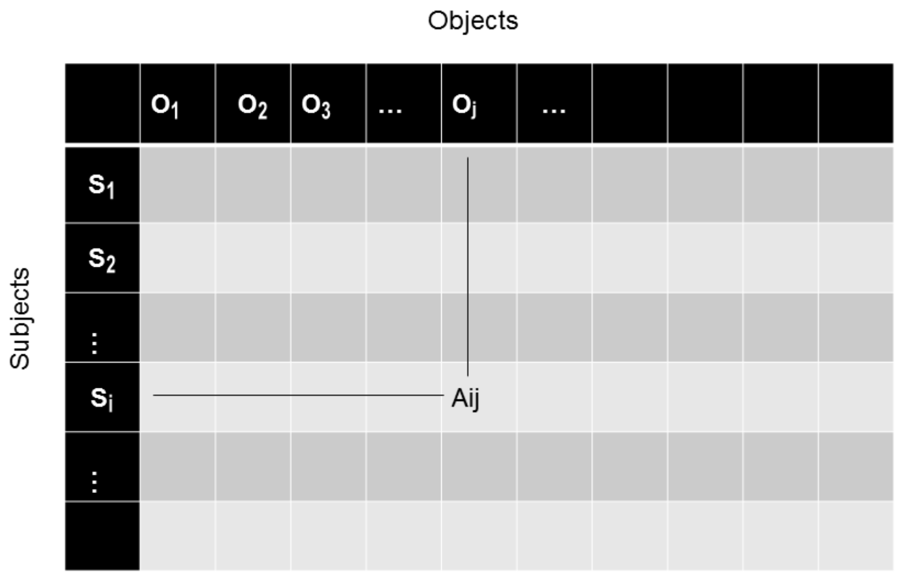

Figure 1: Access Matrix

A key difference between the capability list and access control list is the subject's ability to identify objects in the latter. 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

Discretionary models form a broad class of access control models. 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]. Non-discretionary models are the complement of discretionary models, insofar as they require that access control policy decisions are regulated by a central authority, not by the individual owner of an object. That is, authorizations to objects 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, whose key properties are generalized, abstracted, and described in some formal or semi-formal notation. Therefore, models typically differ from organizational policy in several ways. As mentioned, models deal with abstractions that involve a formal or semi-formal definition, from which the presence or lack of certain properties may be demonstrated. Organizational 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 Policy Machine 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 was originally envisioned as a discretionary access control (DAC) model [Lam71, Gra72]. Many other DAC models have been derived from the access matrix and share common characteristics. The access matrix was later formalized as the now well-known HRU model and used to analyze the complexity of computing the safety properties of the model, which was found to be undecidable [Har76, Tri06]. DAC policies can be expressed in the HRU model, but DAC should not be equated to it, since the HRU model is policy neutral and can also be used to express access control policies that are non-discretionary [Li05].

In addition to an administrator’s ability to manipulate a subject’s authorization to access objects, a DAC access matrix model 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 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 strong 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 model is based on several key entities: subjects, objects, and security labels. A security label designates the conflict-of-interest class and the company dataset of each object.

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 logically organized such that each company dataset belongs to exactly one conflict of interest class, and each object belongs to exactly one company dataset or the dataset of sanitized objects within a specially designated, non-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 company dataset different from 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]. The policy rules of the model are also more restrictive than necessary to meet the stated conflict-of-interest avoidance objective [Sha13]. For instance, as already mentioned, once a subject has read objects from two or more company datasets, it can no longer write to any data set. However, if the datasets reside in different conflict-of-interest classes, no violation of the policy would result were the subject allowed to write to those objects. That is, while the policy rules are sufficient to preclude a conflict of interest from occurring, they are not necessary from a formal logic perspective, since actions that do not incur a conflict of interest are also prohibited by the rules.

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 [Fer01, ANSI04]. It 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. Other roles authorized for the user, which have not been activated, remain dormant and the user does not acquire their associated capabilities. Through this role activation 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 constraints that limit a role's capacity to a fixed number of users, have been incorporated into SoD relations in standard RBAC.

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.

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) 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 policies and policy combinations through a single security model. An important characteristic of the PM framework 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.

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. The relationships incorporated by the PM model are independent of the data used to populate it. Policy specifications are attribute based and capable of expressing and enforcing non-discretionary and discretionary policies [Fer05, Fer11]. Each of the access control security models discussed in the previous chapter can be represented in terms of the PM model’s data elements and relations, such that an access decision rendered by the PM framework would be the same decision as that rendered by the access control model. The simultaneous enforcement of multiple policies, including reconciliation of policy conflicts, is an inherent part of the PM framework [Fer11].

Policy elements of the PM represent not only 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.

3.1 Core Policy Elements

The basic data elements of the PM include authorized users (U), processes (P), objects (O), user and object attributes (UA and OA), policy classes (PC), operations (Op), and access rights (AR). Users are individuals that have been authenticated by the system. A process is a system entity with memory, which operates on behalf of a user. Users submit access requests through processes. The PM treats users and processes as independent but related entities. Most other access control models use the term subject instead of process, while a few others use subject to mean both process and user.

Processes can issue access requests, have exclusive access to their own memory, but none to that of any other process. Processes communicate and exchange data with other processes through some logical 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.

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 based on the protection requirements of the system. By definition, every object is considered to be an object attribute within the PM model; i.e., O is a subset of OA. That is, the identifier of the object is treated not only as an object within PM relations, but may also be treated as an object attribute based on its context within a relation.

User and object attributes are policy elements that represent important characteristics, which are used to organize and distinguish respectively between classes of users and objects. They can also be thought of as containers for users and objects respectively. Policy classes are another important type of element that plays a somewhat similar role to attributes. 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. The way in which policy elements can be assembled and used to represent policy is covered in later chapters.

Operations denote actions that can be performed on the contents of objects that represent resources or on PM data elements and relations that represent policy. The entire set of generic operations, Op, are partitioned into two distinct, finite sets of operations: resource operations, ROp, and administrative operations, AOp. Common resource operations include read and write, for example.[1] Resource operations can also be defined specifically for the environment in which the PM is implemented. Administrative operations on the other hand pertain only to the creation and deletion of PM data elements and relations, and are a stable part of the PM framework, regardless of the implementation environment.

To be able to carry out an operation, the appropriate access rights are required. As with operations, the entire set of access rights, AR, are partitioned into two distinct, finite sets of access rights: resource access rights, RAR, and administrative access rights AAR. Normally a one-to-one mapping exists between ROP and RAR, but not necessarily between AOP and AAR. For instructive purposes, access to object resources are discussed separately from administrative access to policy expressions (i.e., data elements and relations comprising policy). Non-administrative resource operations and access rights are emphasized in this chapter, while the next chapter covers administrative operations and access rights in more detail.

[1] Besides read and write, other resource operations on objects may exist, which are dependent 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.

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 of authorized users; $U = \{u_1, \dots, u_n\}$, where u_i denotes a member of U.

- **P:** A finite set of system processes; $P = \{p_1, \dots, p_n\}$, where p_i denotes a member of P.

- **ROp:** A finite set of resource operations; $ROp = \{rop_1, \dots, rop_{n,}\}$, where rop_i denotes a member of ROp.

- **AOp:** A finite set of administrative operations; $AOp = \{aop_1, \dots, aop_n\}$, where aop_i denotes a member of AOp.

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

- **RAR:** A finite set of access rights; $RAR = \{rar_1, \dots, rar_{n,}\}$, where rar_i denotes a member of RAR.

- **AAR:** A finite set of administrative rights; $AAR = \{aar_1, \dots, aar_n\}$, where aar_i denotes a member of AAR.

- **AR:** A finite set of access rights; $AR = \{ar_1, \dots, ar_n\}$, where ar_i denotes a member of AR.
  $AR = RAR \cup AAR$

- **O:** A finite set of protected objects; $O = \{o_1, \dots, o_n\}$, where o_i denotes a member of O.
  $O \subseteq OA$

- **PC:** A finite set of policy classes; $PC = \{pc_1, \dots, pc_n\}$, where pc_i denotes a member of PC.

- **UA:** A finite set of user attributes; $UA = \{ua_1, \dots, ua_n\}$, where ua_i denotes a member of UA.

- **OA:** A finite set of object attributes; $OA = \{oa_1, \dots, oa_n\}$, where oa_i denotes a member of OA.
  $OA \supseteq O$

- **Process-to-User Mapping:** The function Process_User from domain P to codomain U, such that u = Process_User(p) $\Leftrightarrow$ $p \in P$ is a process operating on behalf of user $u \in U$.
  $\forall p \in P, \exists_1 u \in U$: u = Process_User(p)

3.2 Assignment Relation

Assignments are the means used to express relationships 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 relation is a binary relation on the set of policy elements, PE, where $PE = U \cup UA \cup OA \cup PC$ and $O \subseteq OA$. An individual assignment can be expressed as either $(x, y) \in \text{ASSIGN}$ or x ASSIGN y, on elements x, y of PE. The assignment relation is defined as follows:

$$\text{ASSIGN} \subseteq (\text{U}\times\text{UA}) \cup (\text{UA}\times\text{UA}) \cup (\text{OA}\times\text{OA}) \cup (\text{UA}\times\text{PC}) \cup (\text{OA}\times\text{PC})$$

The assignment relation must satisfy the following properties:

- It is irreflexive; i.e., for all x, y in PE, x ASSIGN y $\Rightarrow x \neq y$.
- It is acyclic; i.e., there does not exist a finite sequence of distinct elements $\langle x_1,x_2,...,x_n\rangle$ in PE, such that $n > 1 \wedge$ for all i in $\{1,2,...,n-1\}$, x_i ASSIGN $x_{i+1} \wedge x_n$ ASSIGN 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 $\text{U} \cup \text{UA} \cup \text{OA}$, there exists a sequence of distinct elements $\langle x_1,x_2,...,x_n\rangle$ in PE, such that $n > 1 \wedge x_1 = w \wedge x_n \in \text{PC} \wedge$ for all i in $\{1,2,...,n-1\}$, x_i ASSIGN x_{i+1}.
- An object attribute cannot be assigned to an object; i.e., for all $x \in \text{OA}$, all $y \in \text{PE}$, an assignment x ASSIGN y implies that $y \notin \text{O}$.

The assignment relation can be represented as a directed graph or digraph G = (PE, ASSIGN), where PE are the vertices of the graph, and each tuple (x, y) of ASSIGN represents a direct edge or arc that originates at x and terminates at y. Within this report, a digraph of policy elements and the assignments among them is referred to as a policy element diagram 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. The simplified policy element diagram of Figure 2 illustrates assignments between each type of policy element.

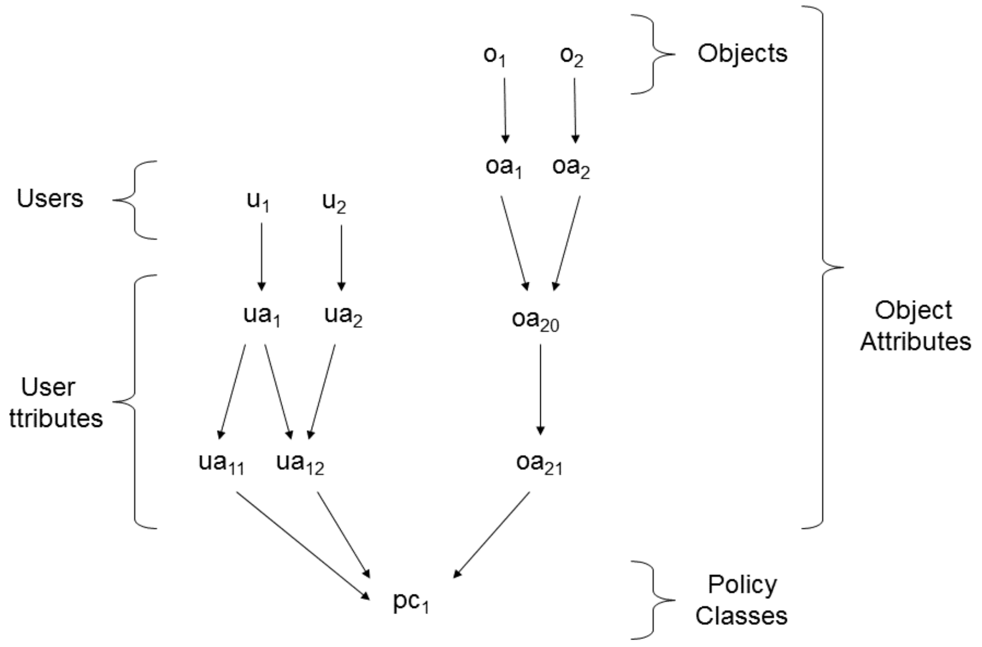

Figure 2: Simplified Policy Element Diagram

The transitive closure of the relation ASSIGN is written as ASSIGN^+. The expression x ASSIGN^+ y denotes that a series of assignments exists from x to y. An equivalent expression is $(x, y) \in \text{ASSIGN}^+$. For all x and y in PE, (x, y) is a member of ASSIGN^+, if and only if (iff) there exists a sequence of distinct elements $\langle x_1, x_2, \ldots ,x_n\rangle$ in PE, such that $n > 1 \ \wedge$ for all i in $\{1,2,\ldots,n-1\}$, x_i ASSIGN $x_{i+1} \wedge x=x_1 \wedge y=x_n$. For example, in Figure 2, a pair of assignments exists from u_1 to ua_1 and from ua_1 onto ua_{12}, which can be expressed as u_1 ASSIGN^+ ua_{12}. Transitive closure is synonymous with the concept of containment, which is an important concept in policy specification and management. For any x and y in PE, x is said to be contained by y, or y is said to contain x, iff x ASSIGN^+ y. For instance, in the aforementioned example, the user attribute ua_{12} can be said to contain the policy elements u_1, u_2, ua_1, ua_2, while user attribute ua_{11} can be said to contain the policy elements u_1, ua_1.

Occasionally, it is useful to express that one element in PE is contained by another through a series of zero or more assignments. The reflexive and transitive closure of the relation ASSIGN, denoted as ASSIGN^*, is a convenient way to represent this situation. That is, for any x and y in PE, the expressions x ASSIGN^* y or $(x, y) \in \text{ASSIGN}^*$ are equivalent ways of stating that y contains x or is itself the element x.

Several helpful functions that make use of the transitive closure and the reflexive and transitive closure relations can now be defined. They are the Users, Objects, and Elements functions. The Users function is a total function from UA to 2^U, which represents the mapping from a user attribute to the set of users that are contained by that user attribute. Intuitively, the function Users(ua) returns the set of users that are contained by or possess the characteristics of the user attribute ua. The function is defined as $\text{Users} \subseteq \text{UA} \times 2^U$, where the following property holds:

$$\forall ua \in UA, \forall x \in 2^U: ((ua, x) \in \text{Users} \Leftrightarrow (\forall u \in U: (u \in x \Leftrightarrow (u, ua) \in \text{ASSIGN}^+))).$$

Similarly, the Objects function represents the mapping from an object attribute to the set of objects that are contained by that object attribute. Intuitively, the function Objects(oa) returns the set of objects that are contained by or possess the characteristics of the object attribute oa. Since all objects are, by definition, members of the set of object attributes, the function must also include the domain of the function within its range, in such instances. The Objects function is a total function from OA to 2^O, which is defined as $\text{Objects} \subseteq \text{OA} \times 2^O$, where the following property holds:

$$\forall oa \in OA, \forall x \in 2^O: ((oa, x) \in \text{Objects} \Leftrightarrow (\forall o \in O: (o \in x \Leftrightarrow (o, oa) \in \text{ASSIGN}^*)))$$

The Elements function generalizes this concept for any policy element. The Elements function is a total function from PE to 2^{PE}, which represents the mapping from a given policy element to the set of policy elements that includes the policy element and all the policy elements contained by that policy element. The function is defined as $\text{Elements} \subseteq \text{PE} \times 2^{PE}$, where the following property holds:

$$\forall pe \in PE, \forall x \in 2^{PE}: ((pe, x) \in \text{Elements} \Leftrightarrow (\forall e \in PE: (e \in x \Leftrightarrow (e, pe) \in \text{ASSIGN*})))$$

3.2.1 User and Object Assignments

A user may be assigned to one or more user attributes. The assignment u ASSIGN ua means that the user u is assigned to or contained by the user attribute ua. It also denotes that user u takes on or acquires the properties held or represented by the attribute ua. The properties of a user attribute are defined as the capabilities for and constraints 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 ASSIGN oa means that that the object o is assigned to or contained by the object attribute oa and takes on or acquires the properties held by the attribute oa. The properties of an object attribute are defined as the capabilities and constraints allotted to users, which govern access to contained objects (i.e., the access modes allowed and denied to specific users).

3.2.2 Attribute Assignments

A user (object) attribute may be assigned to one or more other user (object) attributes. Assignments between user (object) attributes are, by definition, restricted to attributes of the same type (i.e., either all user attributes or object attributes). Containment, as it applies to assignments among attributes, denotes that a user (object) attribute takes on or acquires the properties held or represented by each user (object) attribute that contains it. Containment allows each attribute to augment the properties conferred directly to it with the properties held by every attribute that contains it. Recall that an attribute or other policy element x is said to be contained by another attribute or policy element y, iff x ASSIGN$^+$ y. For example, focusing exclusively on the object attributes in Figure 2, the following expressions are true: oa_1 ASSIGN$^+$ oa_{20}, oa_2 ASSIGN$^+$ oa_{20}, oa_1 ASSIGN$^+$ oa_{21}, oa_2 ASSIGN$^+$ oa_{21}, and oa_{20} ASSIGN$^+$ oa_{21}. With respect to these object attributes, both oa_1 and oa_2 are contained by oa_{20} and acquire the properties of oa_{20}, and oa_1, oa_2, and oa_{20} are contained by oa_{21} and likewise acquire its properties.

Assignments among attributes have an effect on the way users and objects contained by those attributes are treated within the PM model. A user x that is contained by user attribute y gains the properties that are both assigned to and acquired by attribute y. Similarly, an object x that is contained by object attribute y, gains the properties that are both assigned to and acquired by attribute y.

3.2.3 Policy Class Assignments

A user attribute or an object attribute may be assigned to one or more policy classes. 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 policy element is contained by at least one policy class. Unlike attributes, however, a policy class cannot be assigned to any other policy class. In addition, since policy classes do not have properties associated with them, attributes assigned to a policy class do not acquire any properties from it.

Policies can be constructed such that policy elements of one policy class are defined to be mutually exclusive from those of another policy class. That is, if a policy element x is contained by pc_1 in such a policy, it would be precluded from being contained by pc_2 or some other policy class. 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:** The finite set of configured policy elements; $PE \overset{\text{def}}{=} U \cup UA \cup OA \cup PC$ (i.e., {U, UA, OA, PC} is a partition on the set PE).

▪ **Assignment:** The binary relation ASSIGN in the set PE, such that the following hold:
- $ASSIGN \subseteq (U\times UA) \cup (UA\times UA) \cup (OA\times OA) \cup (UA\times PC) \cup (OA\times PC)$
- the relation is irreflexive; i.e., $\ldots x,y \in PE$: $(x\ ASSIGN\ y \Rightarrow x \neq y)$
- the relation is acyclic; i.e., $\nexists s \in iseq_1\ PE$: $(\#s > 1 \wedge$
  $\forall i \in \{1,...,(\#s - 1)\}$: $((s\ (i), s\ (i+1)) \in ASSIGN) \wedge (s\ (\#s), s\ (1)) \in ASSIGN)$
- a path exists from every element in U, UA, and OA to some element in PC; i.e.,
  $\forall w \in (PE \setminus PC), \exists s \in iseq_1\ PE$: $(s\ (1) = w \wedge s\ (\#s) \in PC \wedge$
  $\forall i \in \{1,...,(\#s - 1)\}$: $((s\ (i), s\ (i+1)) \in ASSIGN))$
- assignments to an object from an object attribute are precluded;
  i.e., $\ldots x \in OA, \ldots y \in PE$: $(x\ \ ASSIGN\ \ y \Rightarrow y \notin O)$

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

▪ **Containment:** The binary relation $ASSIGN^+$; i.e., $ASSIGN^+$ is the transitive closure of the assignment relation ASSIGN.
- $ASSIGN \subseteq ASSIGN^+$
- $\ldots x, y \in PE$: $((x, y) \in ASSIGN^+ \Leftrightarrow \exists s \in iseq_1\ PE$: $(\#s > 1 \wedge$
  $\forall i \in \{1,...,(\#s - 1)\}$: $((s\ (i), s\ (i+1)) \in ASSIGN) \wedge x=s(1) \wedge y=s(\#s))$
- x *is contained by* y $\overset{\text{def}}{=}$ $x,y \in PE \ \wedge\ x\ ASSIGN^+\ y$
- y *contains* x $\overset{\text{def}}{=}$ $x,y \in PE \ \wedge\ x\ ASSIGN^+\ y$

The notation x ASSIGN* y is a shorthand expression of the condition that $(x,y) \in ASSIGN^+ \vee\ (x = y)$; i.e., the reflexive and transitive closure of ASSIGN.

▪ **Contained Users Mapping:** The total function Users from domain UA to codomain 2^U.
- $Users \subseteq UA \times 2^U$
- $\forall ua \in UA, \forall x \in 2^U$: $((ua, x) \in Users \Leftrightarrow (\forall u \in U$: $(u \in x \Leftrightarrow (u, ua) \in ASSIGN^+)))$.

▪ **Reflexive and Contained Objects Mapping:** The total function Objects from domain OA to codomain 2^O.
- $Objects \subseteq OA \times 2^O$
- $\forall oa \in OA, \forall x \in 2^O$: $((oa, x) \in Objects \Leftrightarrow (\forall o \in O$: $(o \in x \Leftrightarrow (o, oa) \in ASSIGN^*)))$

▪ **Reflexive and Contained Elements Mapping:** The total function Elements from domain PE to codomain 2^{PE}.
- $Elements \subseteq PE \times 2^{PE}$
- $\forall pe \in PE, \forall x \in 2^{PE}$: $((pe, x) \in Elements \Leftrightarrow (\forall e \in PE$: $(e \in x \Leftrightarrow (e, pe) \in ASSIGN^*)))$

3.3 Associations and Privilege Relations

Associations define relationships that represent the authorization of access rights between policy elements. Privileges are derived from association and assignment relations and as discussed later in this section, are shaped in part by the attribute to attribute assignments defined for a policy.

3.3.1 Associations

The association relation is a policy setting that governs which users are authorized to access which objects and exercise which access rights. Associations are normally formed and rescinded through an according interface of the PM.

The ternary relation, ASSOCIATION $\subseteq$ UA $\times$ 2_1^{AR} $\times$ AT, where 2_1^{AR} represents the set of all subsets of AR, except for the empty set, and AT = OA $\cup$ UA (i.e., the set of all attributes), defines the set of possible associations within a policy specification. An individual triple of ASSOCIATION, (ua, ars, at), may also be denoted as ua—ars—at. For each policy class containing attribute at, the association ua—ars—at specifies that all users contained by ua possess the authority denoted by ars over at and all policy elements contained by at.

Note that attributes appearing within a relation, such as the association relation, often serve as referents. A referent attribute is treated as a designator or representative for the policy elements it contains and possibly the attribute itself, depending on the semantics of the relation. Each referent represents a policy graph containing all policy elements from which the referent is reachable through one or more assignments, and affects all relationships bound to those elements.

Within the context of an association, (ua, ars, at), the third term, attribute at, is treated as a referent for the policy elements within the section of the policy element diagram rooted at the attribute, as well as itself. Similarly, the first term, attribute ua, is treated as a referent for the users and user attributes within the section of the policy element diagram rooted at ua. Although a referent potentially represents many policy elements and relationships, an access right may be pertinent to only a subset of the policy elements that are represented by the referent. That is, an access right might be relevant for one or more of the policy elements a referent attribute contains, or the access right might have relevance only to the attribute itself—it depends entirely on the access right. For example, an association involving r and w access rights on an object attribute are relevant for the objects contained by the attribute, but not for the attribute or contained attributes.

Although not mandatory, as a rule of practice, specifying associations that involve resource access rights separately from associations that involve administrative access rights is advised. One reason for segregating associations this way is that a different perspective applies to each type. Resource associations do not authorize changes to policy and are directed mainly at the actions of ordinary users, while administrative associations are more complex, since they allow policy alterations to occur, and are directed mainly at policy authorities. Another difference relates to access requests: each resource operation is synonymous with and authorizable via a single resource access right (e.g., a "read" operation requires an "r" access right), authorization for an administrative operation, however, often requires possession of multiple administrative access rights.

Any association tuple, (ua, {rar_i, rar_j, … , rar_m, aar_k, aar_l, … , aar_n}, at), whose access right set is mixed with members that belong to RAR and AAR, can be represented as a pair of tuples, (ua, {rar_i, rar_j, … , rar_m}, at) and (ua, {aar_k, aar_l, … , aar_n}, at), without loss of meaning or function. When the access rights of associations pertain exclusively to either object resources or the data elements and relations that make up policy, the association relation can be expressed in two parts as follows: ASSOCIATION $\subseteq$ (UA $\times$ 2_1^{RAR} $\times$ AT) $\cup$ (UA $\times$ 2_1^{AAR} $\times$ AT).

3.3.2 Containment and Attribute Properties

Assignments among attributes affect the interpretation of associations. As mentioned earlier, the inherent properties of an attribute include not only those conferred directly to the attribute, but also the properties it acquires from every attribute in which it is contained. A simple example to demonstrate containment among attributes is given in Figure 3.

Figure 3 depicts an authorization graph containing the following policy elements: U = {u_1, u_2, u_3}, O = {o_1, o_2, o_3}, UA = {Group1, Group2, Division}, OA = {Project1, Project2, Projects} $\cup$ O, and PC = {OU}. An authorization graph is a policy element diagram annotated with associations and other relationships that exist between policy elements. Associations are illustrated using dotted, downward-arcing connectors between the elements involved in each association. The following three resource associations involving access rights that represent the authority to read or write (i.e., {r, w}) are shown in Figure 3: (Group1, {w}, Project1), (Group2, {w}, Project2), and (Division, {r}, Projects). The inherent properties of each user attribute in the diagram can be determined, by combining those properties granted directly to the attribute through a defined association with those acquired indirectly through containment by an attribute possessing the property, as follows:

- Group1 is granted the capability of ({w}, Project1) through an association and acquires the capability of ({r}, Projects) from Division through an assignment. The inherent properties (i.e., capabilities) of Group1, therefore, are ({w}, Project1) and ({r}, Projects).

- Group2 is granted the capability of ({w}, Project2) and acquires the capability of ({r}, Projects) from Division, resulting in the inherent properties ({w}, Project2) and ({r}, Projects).

- Division is granted the capability of ({r}, Projects), but acquires no capabilities, since it is not contained by any another user attribute, resulting in the single, inherent property ({r}, Projects).

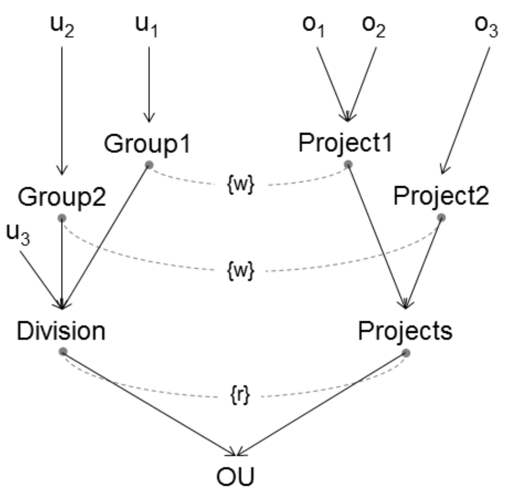

Figure 3: Simple Authorization Graph

For this same example, the properties of each object attribute in the hierarchy that are granted through a defined association or acquired through containment by another object attribute can also be determined. That is, rather than a list of inherent capabilities, a list of inherent access control entries can be determined for each object attribute, as follows:

- Project1 is granted the access control entry (Group1, {w}), and acquires the access control entry (Division, {r}) from Projects, which leads to the inherent properties (Group1, {w}) and (Division, {r}) for Project1.
- Project2 is granted the access control entry (Group2, {w}), and acquires the access control entry (Division, {r}) from Projects, which leads to the inherent properties (Group2, {w}) and (Division, {r}).
- Projects is granted the access control entry (Division, {r}), but acquires no access control entries from another object attribute, which leads to the inherent property (Division, {r}).

While it is relatively easy to determine the granted and acquired properties of attributes for the simple example given in Figure 3, it can be considerably more difficult to illustrate and analyze a more realistic example that involves a larger set of policy elements, assignments, and associations. The interactions between vertical assignment relations and horizontal association relations increase in complexity quickly as more elements and relations are added to an authorization graph.

3.3.3 Derived Privileges

A privilege specifies a relationship between a user, an access right, and an attribute. Privileges are derived from higher level relations, namely the associations between and the assignments among attributes. That is, every privilege originates from an association and the containment properties of the user attribute and the target attribute of that association, which are designated through assignments.

The ternary relation PRIVILEGE $\subseteq$ U $\times$ AR $\times$ (PE\PC) defines the set of possible privileges within a policy specification. An individual tuple of PRIVILEGE, (u, ar, e), denotes that user u holds the authority to exercise the access right ar on policy element e. The privilege relation may be partitioned into two parts, PRIVILEGE $\subseteq$ (U $\times$ RAR $\times$ (PE\PC)) $\cup$ (U $\times$ AAR $\times$ (PE\PC)), in which tuples of the first term, (u, rar, e), denote resource-oriented privileges, and tuples of the second term, (u, aar, e), denote policy-oriented privileges.

For policies comprising a single policy class, the derivation of privileges is straightforward. Specifically, the triple (u, ar, e) is a privilege, iff there exists an association, (ua, ars, at) $\in$ ASSOCIATION, such that u ASSIGN$^+$ ua, ar $\in$ ars, where ars $\in 2_1^{AR}$, and e ASSIGN* at. This can also be restated using the Users and Elements functions: the triple (u, ar, e) is a privilege, iff there exists a tuple (ua, ars, at) $\in$ ASSOCIATION, such that u $\in$ Users(ua), ar $\in$ ars, ars $\in 2_1^{AR}$, and e $\in$ Elements(at). 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 3, the entire set of privileges for the authorization graph can be enumerated for each of the associations, as follows:

- For the association triple (Group1, {w}, Project1), Users(Group1)={u_1}, rars={w}, and Elements(Project1)={Project1, o_1, o_2}. Therefore, the derived privileges for this association are (u_1, w, Project1), (u_1, w, o_1), and (u_1, w, o_2).

- For the association triple (Group2, {w}, Project2), Users(Group2)={u_2}, rars={w}, and Elements(Project2)={Project2, o_3}. Therefore, the derived privileges for this association are (u_2, w, Project2), (u_2, w, o_3).

- For the association triple (Division, {r}, Projects), Users(Division)={u_1, u_2, u_3}, rars={r}, and Elements(Projects)={Projects, Project1, Project2, o_1, o_2, o_3}. Therefore, the derived privileges for this association are (u_1, r, Projects), (u_1, r, Project1), (u_1, r, Project2), (u_1, r, o_1), (u_1, r, o_2), (u_1, r, o_3), (u_2, r, Projects), (u_2, r, Project1), (u_2, r, Project2), (u_2, r, o_1), (u_2, r, o_2), (u_2, r, o_3), (u_3, r, Projects), (u_3, r, Project1), (u_3, r, Project2), (u_3, r, o_1), (u_3, r, o_2), and (u_3, r, o_3).

Since r and w access rights apply only to objects representing resources, not all the privileges that are derived have relevance in forming an access decision. In this example, the privileges (u_1, w, Project1), (u_2, w, Project2), (u_1, r, Projects), (u_1, r, Project1), (u_1, r, Project2), (u_2, r, Projects), (u_2, r, Project1), (u_2, r, Project2), (u_3, r, Projects), (u_3, r, Project1), (u_3, r, Project2), which involve object attributes as the third element of the triple, are not the primary result. These privileges serve mainly as an intermediate factor in determining the more essential, remaining privileges that are used to reach decisions on access requests for object resources. An implementation could generate such intermediate privileges and, because of the mismatch between the type of policy element to which the access right applies and the type of policy element in the triple, ignore them once essential, primary privileges are derived. Alternatively, the essential privileges could be generated directly, provided that associations do not mix resource and administrative access rights.

Similar to the way inherent properties of associations can be view from either an access entry or capability orientation, these same perspectives can be applied to privileges. That is, a user u may access a policy element via the capability, (ar, e), iff the privilege (u, ar, e) exists, or a user may access an policy element e via the access control entry, (u, ar), iff a privilege (u, ar, e) exists.

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.

▪ **ARs:** A finite set of all subsets of access rights defined in AR, excluding the empty set; ars or ars_1, ars_2, … denote a member of ARs, unless otherwise specified.
  $ARs = 2_1^{AR}$

▪ **AT:** The finite set of user and object attributes; $AT \stackrel{def}{=} UA \cup OA$.

▪ **Associations:** The ternary relation ASSOCIATION from UA to 2_1^{AR} to AT.
- $ASSOCIATION \subseteq UA \times 2_1^{AR} \times AT$
- If segregated by RAR and AAR,
  $ASSOCIATION \subseteq (UA \times 2_1^{RAR} \times AT) \cup (UA \times 2_1^{AAR} \times AT)$

▪ **Resource Associations:** The ternary relation ASSOCIATION from UA to 2_1^{RAR} to AT.
  $ASSOCIATION \subseteq UA \times 2_1^{RAR} \times AT \subseteq UA \times 2_1^{AR} \times AT$

▪ **Administrative Associations:** The ternary relation ASSOCIATION from UA to 2_1^{AAR} to AT.
  $ASSOCIATION \subseteq UA \times 2_1^{AAR} \times AT \subseteq UA \times 2_1^{AR} \times AT$

▪ **Inherent Capabilities of a User Attribute:** The partial function ICap from UA to $2^{(ARs \times AT)}$, where $ARs = 2_1^{AR}$.
- $ICap \subseteq UA \times 2^{(ARs \times AT)}$
- …ua∈UA, …ars∈2_1^{AR}, …at∈AT: ((ars, at) ∈ ICap(ua) ⇔ (ua, ars, at) ∈ ASSOCIATION)

▪ **Inherent Access Entries of an Attribute:** The partial function IAE from AT to $2^{(UA \times ARs)}$, where $ARs = 2_1^{AR}$.
- $IAE \subseteq AT \times 2^{(UA \times ARs)}$
- …ua∈UA, …ars∈2_1^{AR}, …at∈AT: ((ua, ars) ∈ IAE(at) ⇔ (ua, ars, at) ∈ ASSOCIATION)

▪ **Privileges (for single policy class):** The ternary relation PRIVILEGE from U to AR to PE\PC.
- PRIVILEGE ⊆ U × AR × (PE\PC)
- If partitioned by RAR and AAR,
  PRIVILEGE ⊆ (U × RAR × (PE\PC)) ∪ (U × AAR × (PE\PC))
- …u∈U, …ar∈AR, …e∈(PE\PC): ((u, ar, e) ∈ PRIVILEGE ⇔ ∃ars∈2_1^{AR}, ∃ua∈UA, ∃at∈AT: ((ua, ars, at) ∈ ASSOCIATION ⋀ u ∈ Users(ua) ⋀ ar ∈ ars ⋀ e ∈ Elements(at)))

▪ **Access Control Entry of an Element:** The function ACE from PE\PC to $2^{(U \times AR)}$.
- $ACE \subseteq PE \backslash PC \times 2^{(U \times AR)}$
- …u∈U, …ar∈AR, …e∈(PE\PC): ((u, ar) ∈ ACE(e) ⇔ (u, ar, e) ∈ PRIVILEGE)

- **Capability of a User:** The function Cap from U to $2^{(AR\times(PE\backslash PC))}$.
  - Cap $\subseteq$ U × $2^{(AR\times(PE\backslash PC))}$
  - …u∈U, …ar∈AR, …e∈(PE\PC): ((ar, e) ∈ Cap(u) ⇔ (u, ar, e) ∈ PRIVILEGE)

3.4 Prohibitions and Restriction Relations

Prohibitions define relationships that govern the suppression of access rights between policy elements. Three distinct, but related types of prohibitions exist: user-based, process-based, and user attribute-based prohibitions. Prohibitions in a way can be thought of as the antithesis of associations, because restrictions, which are derived from prohibitions, essentially negate privileges that would otherwise allow access to occur. That is, each type of prohibition denotes an effective set of restrictions, which represent privileges that a specific user, process, or class of users is precluded from exercising, regardless of whether the user, process, or class of users in question hold any of the actual privileges.

To illustrate the use of prohibitions, Figure 4 reillustrates the example authorization graph discussed the previous section, augmented with a single user prohibition. The prohibition, denoted (u_2, {r}, {Project1}, ∅), is depicted using a dotted, upward-arcing connector. The prohibition denies user u_2 from exercising the read access right over Project1 and objects contained within Project1, which has the effect of negating the privileges (u_2, r, o_1), (u_2, r, o_2), and (u_2, r, Project1), derived from the association (Division, {r}, Projects1) (i.e, for user u_2, with regard to the objects contained in Project1, o_1 and o_2). Note that the prohibition does not negate the effect of the association with regard to objects contained in Project2 (viz., o_3), nor does it affect the access rights of any new user that might be assigned to Group2 in the future.

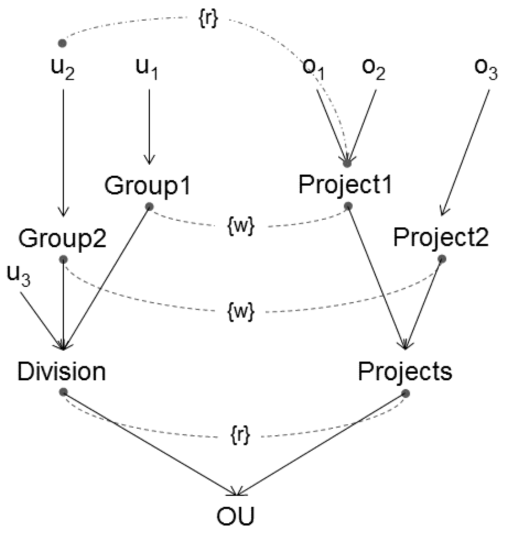

Figure 4: Authorization Graph with Prohibition

Prohibitions can be formed and rescinded through an according interface of the PM. User- and user attribute-based prohibitions are persistent and remain in existence until they are rescinded through an administrative action. Process-based prohibitions are less enduring and treated differently than user- or user attribute-based prohibitions. Process-based prohibitions are normally formed through predefined rules known as obligations, which are executed automatically based on event occurrence to establish the prohibition. Once the affected process terminates, the prohibition no longer has applicability and is rescinded automatically by the PM framework.

A 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 retained, the entire prohibition must be rescinded and replaced with new prohibition for the remaining subset that are still in effect. As with associations, it is advisable not to mix resource and administrative access rights when specify prohibitions, and instead define prohibitions with the same type of access rights.

The set of policy elements affected by a prohibition is designated via either conjunctive or disjunctive mappings over sets of referent attributes to the policy elements in question. A pair of functions facilitate the discussion of the affected policy elements. The disjunctive range function represents the mapping from two constraint sets of attributes—the first designating policy elements for inclusion, and the second designating policy elements for exclusion—to a set of policy elements formed by logical disjunction of the policy elements contained within or not contained respectively within the subgraphs of the referent attributes of each constraint set. More precisely, the set of policy elements returned by the disjunctive range function, DisjRange(atis, ates), where atis$\in 2^{AT}$ and ates$\in 2^{AT}$, is the union of Elements(ati), for all ati in the inclusion set atis, along with the union of ((PE\PC) \ Elements(ate)), for all ate in the exclusion set ates, which can be expressed more succinctly as follows:

$$\text{DisjRange(atis, ates)} = \bigcup_{\text{ati}\in\text{atis}} \text{Elements(ati)} \;\cup\; \bigcup_{\text{ate}\in\text{ates}} ((\text{PE} \setminus \text{PC}) \setminus \text{Elements(ate)})$$

Similarly, the conjunctive range function represents the mapping from two constraint sets of attributes—the first designating policy elements for inclusion, and the second designating policy elements for exclusion—to a set of policy elements formed by logical conjunction of the policy elements contained by or not contained by the attributes of each constraint set respectively. More precisely, the set of policy elements returned by the conjunctive range function, ConjRange(atis, ates), where atis$\in 2^{AT}$ and ates$\in 2^{AT}$, is the intersection of Elements(ati), for all ati in the inclusion set atis, along with the intersection of ((PE\PC) \ Elements(ate)), for all ate in the exclusion set ates, which can be expressed more succinctly as follows:

$$\text{ConjRange(atis,ates)} = \bigcap_{\text{ati}\in\text{atis}} \text{Elements(ati)} \;\cap\; \bigcap_{\text{ate}\in\text{ates}} ((\text{PE} \setminus \text{PC}) \setminus \text{Elements(ate)})$$

The disjunctive and conjunctive forms of prohibitions use these two functions to delineate the policy elements targeted by a prohibition. They allow complex expressions to be specified for a prohibition. In practice, the prohibitions for most policies typically require the use of only simple expressions. For example, the sets atis and ates may both be singletons, each containing only one member of the same attribute type (viz., either user or object), or one of the sets may be the empty set and the other a singleton. However, the capabilities are available to meet the demands of more complex policies that might arise.

3.4.1 User-based Prohibitions and Restrictions

The quaternary relation U_DENY_DISJ $\subseteq$ U $\times$ 2_1^{AR} $\times$ 2^{AT} $\times$ 2^{AT}, defines the set of user-based, disjunctive prohibitions for a policy specification. An individual tuple (u, ars, atis, ates) $\in$ U_DENY_DISJ, where u $\in$ U, ars $\in$ 2_1^{AR}, atis $\in$ 2^{AT}, ates $\in$ 2^{AT}, and atis $\cup$ ates $\neq$ $\emptyset$, denotes that all processes initiated by the user are withheld the authority to exercise any of the access rights within the access right set, ars, against any policy elements that lie within the disjunctive range of the inclusion and exclusion sets of attributes, DisjRange(atis,ates). That is, any process p, executing on behalf of user u (i.e., u = Process_User(p)), cannot exercise the access rights in ars on any policy element that is contained by at least one of the attributes in atis (i.e., the inclusionary attribute set) or, with the exception of policy classes, is not contained by at least one of the attributes in ates (i.e., the exclusionary attribute set).

A complementary relation to U_DENY_DISJ that defines user-based, conjunctive prohibitions, also exists within the PM model. An individual tuple (u, ars, atis, ates) of the quaternary relation U_DENY_CONJ $\subseteq$ U $\times$ 2_1^{AR} $\times$ 2^{AT} $\times$ 2^{AT}, where atis $\cup$ ates $\neq$ $\emptyset$, denotes that all processes initiated by the user are withheld the authority to exercise any of the access rights defined in the access right set, ars, against any policy elements that lie within the conjunctive range of the inclusion and exclusion sets of attributes, ConjRange(atis,ates). That is, any process p, executing on behalf of user u, cannot exercise the access rights in ars on any policy element that is contained by all of the attributes in atis and, with the exception of policy classes, is also not contained by any of the attributes in ates.

The user restriction relation, U_RESTRICT $\subseteq$ U $\times$ AR $\times$ PE\PC, is derived from one or more user prohibition relations. For a policy comprising a single policy class, the triple (u, ar, e) is a user restriction, iff there exists a user prohibition (u, ars, atis, ates), such that the following is true:

- The user, u, is designated by the user identifier of the prohibition.
- The access right, ar, is a member of the access right set of the prohibition.
- The policy element, e, lies within either the disjunctive or conjunctive range of the inclusion and exclusion attribute sets of the prohibition, respective of whether the prohibition is a disjunctive or conjunctive variant.

3.4.2 Process-based Prohibitions and Restrictions

Process-based prohibitions are defined similarly to user-based prohibitions. The relation P_DENY_DISJ $\subseteq$ P $\times$ 2_1^{AR} $\times$ 2^{AT} $\times$ 2^{AT} defines the set of process-based, disjunctive prohibitions. A tuple (p, ars, atis, ates) $\in$ P_DENY_DISJ denotes that the process, p, is prohibited from exercising any of the access rights defined in the access right set, ars, against any of the policy elements that lie within the disjunctive range of the inclusion and exclusion attribute sets, DisjRange(atis,ates). The inclusion and exclusion sets cannot both be the empty set.

As with user-based prohibitions, the conjunctive form of process-based prohibitions also exists. The relation P_DENY_CONJ $\subseteq$ P $\times$ 2_1^{AR} $\times$ 2^{AT} $\times$ 2^{AT} defines the set of process-based, conjunctive prohibitions. A tuple (p, ars, atis, ates) $\in$ P_DENY_CONJ denotes the process, p, is prohibited from exercising any of the access rights defined in the access right set, ars, against any of the policy elements that lie within the conjunctive range of the inclusion and exclusion attribute sets, ConjRange(atis,ates). The inclusion and exclusion sets cannot both be the empty set. 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 accesses through another of its processes, presuming that the appropriate associations are defined that would allow them. This situation can be easily remedied through the use of a user-based prohibition, whose scope is broader than a single process.

The process restriction relation, P_RESTRICT $\subseteq$ P $\times$ AR $\times$ PE\PC, is derived from one or more process prohibition relations. For a policy comprising a single policy class, the triple (p, ar, e) is a process restriction, iff there exists a process prohibition (p, ars, atis, ates), such that the following is true:

- The process, p, is designated by the process identifier of the prohibition.
- The access right, ar, is a member of the access right set of the prohibition.
- The policy element, e, lies within either the disjunctive or conjunctive range of the inclusion and exclusion attribute sets of the prohibition, respective of whether the prohibition is a disjunctive or conjunctive variant.

3.4.3 Attribute-based Prohibitions and Restrictions

The following pair of prohibitions are defined similarly to user-based and process-based prohibitions, but instead are based on user attribute. The quaternary relation UA_DENY_DISJ $\subseteq$ UA $\times$ 2_1^{AR} $\times$ 2^{AT} $\times$ 2^{AT} defines the set of user attribute-based, disjunctive prohibitions, and the quaternary relation UA_DENY_CONJ $\subseteq$ UA $\times$ 2_1^{AR} $\times$ 2^{AT} $\times$ 2^{AT} defines the set of user attribute-based conjunctive prohibitions. A tuple (ua, ars, atis, ates) $\in$ UA_DENY_DISJ denotes that all processes initiated by any user contained by the attribute, ua, are prohibited from exercising any of the access rights defined in the access right set, ars, against any of the policy elements that lie within the disjunctive range of the inclusion and exclusion attribute sets, DisjRange(atis, ates). Similarly, a tuple (ua, ars, atis, ates) $\in$ UA_DENY_CONJ denotes that all processes initiated by any user contained by the attribute, ua, are prohibited from exercising any of the access rights defined in the access right set, ars, against any of the policy elements that lie within the conjunctive range of the inclusion and exclusion attribute sets, ConjRange(atis,ates). The inclusion and exclusion attribute sets cannot both be the empty set.

The user attribute restriction relation, UA_RESTRICT $\subseteq$ UA $\times$ AR $\times$ PE, is derived from one or more user attribute prohibition relations. For a policy comprising a single policy class, the triple (ua, ar, pe) is a user attribute restriction, iff there exists a user attribute prohibition (ua, ars, atis, ates), such that the following is true:

- The attribute, ua, is designated by the user attribute identifier of the prohibition.
- The access right, ar, is a member of the access right set of the prohibition.

- The policy element, e, lies within either the disjunctive or conjunctive range of the inclusion and exclusion attribute sets of the prohibition, respective of whether the prohibition is a disjunctive or conjunctive variant.

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

- **User Deny Disjunctive Prohibition:** The quaternary relation U_DENY_DISJ from U to 2_1^{AR} to 2^{AT} to 2^{AT}.
  - $\text{U\_DENY\_DISJ} \subseteq U \times 2_1^{AR} \times 2^{AT} \times 2^{AT}$
  - $\forall u{\in}U, \forall ars{\in}2_1^{AR}, \forall atis{\in}2^{AT}, \forall ates{\in}2^{AT}: ((u, ars, atis, ates) \in \text{U\_DENY\_DISJ} \Rightarrow (atis \cup ates \neq \emptyset))$

- **User Deny Conjunctive Prohibition:** The quaternary relation U_DENY_CONJ from U to 2_1^{AR} to 2^{AT} to 2^{AT}.
  - $\text{U\_DENY\_CONJ} \subseteq U \times 2_1^{AR} \times 2^{AT} \times 2^{AT}$
  - $\forall u{\in}U, \forall ars{\in}2_1^{AR}, \forall atis{\in}2^{AT}, \forall ates{\in}2^{AT}: ((u, ars, atis, ates) \in \text{U\_DENY\_CONJ} \Rightarrow (atis \cup ates \neq \emptyset))$

- **User Restriction:** The quaternary relation U_RESTRICT from U to AR to (PE\PC).
  - $\text{U\_RESTRICT} \subseteq U \times AR \times (PE \backslash PC)$
  - $\forall u{\in}U, \forall ar{\in}AR, \forall e{\in}(PE \backslash PC): ((u, ar, e) \in \text{U\_RESTRICT} \Rightarrow$
    $\exists ars{\in}2^{AR}, \exists atis{\in}2^{AT}, \exists ates{\in}2^{AT}:$
    $(((u, ars, atis, ates) \in \text{U\_DENY\_DISJ} \wedge ar \in ars \wedge e \in \text{DisjRange}(atis, ates)) \vee$
    $((u, ars, atis, ates) \in \text{U\_DENY\_CONJ} \wedge ar \in ars \wedge e \in \text{ConjRange}(atis, ates)))$

- **Process Deny Disjunctive Prohibition:** The quaternary relation P_DENY_DISJ from P to 2_1^{AR} to 2^{AT} to 2^{AT}.
  - $\text{P\_DENY\_DISJ} \subseteq P \times 2_1^{AR} \times 2^{AT} \times 2^{AT}$
  - $\ldots p{\in}P, \forall ars{\in}2_1^{AR}, \forall atis{\in}2^{AT}, \forall ates{\in}2^{AT}: ((p, ars, atis, ates) \in \text{P\_DENY\_DISJ} \Rightarrow (atis \cup ates \neq \emptyset))$

- **Process Deny Conjunctive Prohibition:** The quaternary relation P_DENY_CONJ from P to 2_1^{AR} to 2^{AT} to 2^{AT}.
  - $\text{P\_DENY\_CONJ} \subseteq P \times 2_1^{AR} \times 2^{AT} \times 2^{AT}$
  - $\ldots p{\in}P, \forall ars{\in}2_1^{AR}, \forall atis{\in}2^{AT}, \forall ates{\in}2^{AT}: ((p, ars, atis, ates) \in \text{P\_DENY\_CONJ} \Rightarrow (atis \cup ates \neq \emptyset))$

- **Process Restriction:** The quaternary relation P_RESTRICT from P to AR to (PE\PC).
  - $\text{P\_RESTRICT} \subseteq P \times AR \times (PE \backslash PC)$
  - $\forall p{\in}P, \forall ar{\in}AR, \forall e{\in}(PE \backslash PC): ((p, ar, e) \in \text{U\_RESTRICT} \Rightarrow$
    $\exists ars{\in}2^{AR}, \exists atis{\in}2^{AT}, \exists ates{\in}2^{AT}:$
    $(((p, ars, atis, ates) \in \text{P\_DENY\_DISJ} \wedge ar \in ars \wedge e \in \text{DisjRange}(atis, ates)) \vee$
    $((p, ars, atis, ates) \in \text{P\_DENY\_CONJ} \wedge ar \in ars \wedge e \in \text{ConjRange}(atis, ates)))$

- **User Attribute Deny Disjunctive Prohibition:** The quaternary relation UA_DENY_DISJ from UA to 2_1^{AR} to 2^{AT} to 2^{AT}.
  - $\text{UA\_DENY\_DISJ} \subseteq UA \times 2_1^{AR} \times 2^{AT} \times 2^{AT}$

- $\ldots ua \in UA, \forall ars \in 2_1^{AR}, \forall atis \in 2^{AT}, \forall ates \in 2^{AT}$: $((ua, ars, atis, ates) \in UA\_DENY\_DISJ \Rightarrow (atis \cup ates \neq \emptyset))$

▪ **User Attribute Deny Conjunctive Prohibition:** The quaternary relation UA_DENY_CONJ from UA to 2_1^{AR} to 2^{AT} to 2^{AT}.

- $UA\_DENY\_CONJ \subseteq UA \times 2_1^{AR} \times 2^{AT} \times 2^{AT}$
- $\ldots ua \in UA, \forall ars \in 2_1^{AR}, \forall atis \in 2^{AT}, \forall ates \in 2^{AT}$: $((ua, ars, atis, ates) \in UA\_DENY\_CONJ \Rightarrow (atis \cup ates \neq \emptyset))$

▪ **User Attribute Restriction:** The quaternary relation UA_RESTRICT from UA to AR to (PE\PC).

- $UA\_RESTRICT \subseteq UA \times AR \times (PE \setminus PC)$
- $\forall ua \in UA, \forall ar \in AR, \forall e \in (PE \setminus PC)$: $((ua, ar, e) \in U\_RESTRICT \Rightarrow$
  $\exists ars \in 2^{AR}, \exists atis \in 2^{AT}, \exists ates \in 2^{AT}$:
  $(((ua, ars, atis, ates) \in UA\_DENY\_DISJ \wedge ar \in ars \wedge e \in DisjRange(atis, ates)) \vee$
  $((ua, ars, atis, ates) \in UA\_DENY\_CONJ \wedge ar \in ars \wedge e \in ConjRange(atis, ates)))$

3.5 Obligations

Obligations represent potential changes to the authorization state of the policy. They are used when one or more administrative actions need to be carried out under a specific, known set of circumstances. Events are the means by which obligations are triggered. An event occurs each time a requested access 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 sequence of arguments of the triggering event are mandatory and always returned as part of the event context. Other information conveyed via the event context varies based on the type of event that occurred and may include items such as the containing attributes of targeted policy elements (e.g., those pertaining to an object that was deleted), or the types of certain arguments used in an access request (e.g., for a resource access, the type of object (file, message, etc.) accessed.

The two main components needed to define an obligation are an event pattern and a response. The pattern and response elements each denote a sentence in a grammar that respectively expresses the triggering conditions of an event pattern and the administrative actions of the response. The event pattern and response each represents sentences that must conform to a formal language over their respective alphabet. That is, the syntax of the sequence of symbols for each item must be well formed according to its respective grammar.

The obligation relation, OBLIG, is defined as a ternary relation from U to PATTERN to RESPONSE. For each tuple (u, pattern, response) of the obligation relation, u represents the user that established the pattern and response, and under whose authorization the response is carried out. In this report, the pattern and response of obligation is sometimes expressed less formally, in a more readable form, as follows:

When pattern **do** response

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 description of a sequence of administrative actions to be taken (i.e., a computation over the policy specification) constitutes the response. Any unresolved terms used in the response are determined from the details of the triggering event, including items derived from evaluation of the event pattern. Administrative actions are capable of adjusting policy through changes to the prevailing policy element relationships and policy elements.

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* of an event context item. EC.c, EC.p, EC.u, EC.op, and EC.argseq refer respectively to the identifiers of the creator of the obligation, the triggering process, the user for the process, the triggering access operation, and the associated argument sequence (i.e., the operands of the operation), which are conveyed in the event context of every event.[2]

An obligation is typically created through an administrative routine. The user creating the obligation, normally an administrator, must have sufficient authorization to not only define the obligation, but also perform the body of the response. However, the latter typically cannot be verified when the obligation is created, since some terms used in the response cannot be resolved until the obligation is matched. When the event pattern of a defined obligation is matched, the associated response can be fully resolved and carried out automatically under the authorization of the user that created it, regardless how or by whom the event was caused. Therefore, the user that defined the obligation relation must possess adequate authority to carry out the associated response at the time the pattern is matched with the event; otherwise, the response is not attempted. An obligation's response can conceivably be involved in a race condition with administrative activities being conducted manually, as well as the responses of other obligations triggered concurrently.

Obligations provide a powerful means to define, as part of a policy specification, the specific circumstances associated with a triggering 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 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 $\subseteq$ U $\times$ PATTERN $\times$ RESPONSE. For a tuple of OBLIG, (u, pattern, response), u

[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.

represents the user responsible for establishing the obligation and under whose authorization the response is carried out. To interoperate, all functional entities supporting an application must use the same grammars for the pattern and the response syntaxes when forming and processing an obligation. For this report, the conditional expressions that appear in a pattern follow the syntax of first-order predicate calculus summarized in Appendix B; the response is expressed as a sequence of one or more administrative routines, whose syntax (i.e., the name of the routine, followed by the calling arguments) is discussed in detail in Section 4.5.2 of the next chapter.

Insofar as their ability to change the authorization state is concerned, obligations have a similar effect as administrative routines. An important difference exists between them. An administrative routine is typically carried out in response to an access request that has first been subjected to the access decision function for approval, which ensures that the requestor holds sufficient authorization. When the context of an event matches the circumstances of a defined obligation, the routines that make up the associated response are carried out automatically, under the authorization held by the user that defined the obligation. Therefore, to ensure comparable mediation as that done for access requests, before the response can be carried out, the requisite authorization for the administrative routines must be verified for the user that defined the obligation.

The benefits of obligations are twofold: complex policies that require more dynamic treatment than can be expressed in the policy representation can be accommodated, and repetitive administrative activities that involve a lengthly sequence of administrative actions can be automated. The main drawback is the potential to cause grave harm to the authorization state through error or intent. The former can be mitigated by thoroughly testing any obligation before it is enabled, and the latter by taking judicious care when employing obligations, preferably granting only trusted individuals the authorization to define obligations and restricting the scope of those individual's authority to well-defined groups of policy elements.

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* of the respective item in the event context of the spawning event.

▪ **PATTERN:** A finite sequence of symbols over the alphabet Σ_P, which represents the logical expression of an event pattern. PATTERN denotes a formal language over the alphabet in question. The alphabet and language grammar used to specify event patterns are an implementation choice.

$\text{PATTERN} \subseteq \text{seq}_1\ \Sigma_P$

▪ **RESPONSE:** A finite sequence of symbols over the alphabet Σ_R, which represents one or more administrative actions that constitute 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.

$\text{RESPONSE} \subseteq \text{seq}_1\ \Sigma_R$

- **Obligations:** The ternary relation OBLIG from U to PATTERN to RESPONSE. $\text{OBLIG} \subseteq \text{U} \times \text{PATTERN} \times \text{RESPONSE}$

3.6 Access Request Decisions

Authenticated users do not generate access requests directly, but instead instantiate one or more processes to generate requests on their behalf. The access decision function controls accesses in terms of processes. The user on whose behalf the process operates must hold sufficient authority over the policy elements involved in the request, in the form of at least one and possibly several privileges. That is, an access request to perform an operation on a policy element, including objects, are issued only from processes acting on behalf of some user, and is granted provided the appropriate privileges exist that allow the access, and no restriction exists that prevents the access. If a restriction does exist, the access request is denied.

The relation, $\text{AREQ} \subseteq \text{P} \times \text{Op} \times \text{Argseq}$, where $\text{Argseq} = \text{seq}_1\ \text{Arg}$ and $\text{Arg} = \{x \mid x \in \text{PE} \lor x \in 2^{\text{PE}} \lor x \in 2_1^{\text{AR}} \lor x \in \text{PATTERN} \lor x \in \text{RESPONSE}\}$, defines the set of access requests. A tuple of the access request relation, (p, op, argseq), denotes that process, p, is requesting to perform the operation, op, on the resource or policy items referenced by the argument sequence, argseq. The argument sequence is a finite sequence of one or more arguments, which is compatible with the scope of the operation. Each argument in the sequence can be either a distinct policy element, a set of policy elements, a set of access rights, an event pattern, or an event response, as is appropriate for the operation. That is, an access request comprises an operation and a list of enumerated arguments whose type and order are dictated by the operation. Note that resource operations typically take an argument sequence comprising a single element (viz., an object identifier), while administrative operations typically take an argument sequence comprising multiple elements.

To determine the disposition of an access request, the access decision function requires a mapping from the operation and argument sequence in question to the set of capabilities that the process must hold in order for the request to be granted. The required capabilities mapping is defined as the partial binary function ReqCap from (Op × Argseq) to $2^{(\text{AR}\times\text{PE})}$, such that $\forall \text{op}\in\text{Op}, \forall \text{argseq}\in\text{Argseq}: (\text{capset} \in \text{ReqCap}(\text{op, argseq}) \Rightarrow \forall \text{ar} \in \text{AR}, \forall \text{pe} \in \text{PE}\backslash\text{PC}: ((\text{ar, e}) \in \text{capset}$, if and only if (ar, e), together with the other access rights in capset, form a minimal set of requisite capabilities needed to perform the operation op on argseq)). The mapping ReqCap(op, argseq) returns a set of one or more capability sets, for which the capabilities of a set, if held by the process, would allow the process to carry out the access request.[3]

The access decision function grants a process, p, permission to execute an access request (p, op, argseq), provided the following conditions hold for each access right and policy element pair (ar,

[3] Each administrative routine described in Appendix D contains, where appropriate, a comment indicating the capabilities needed to perform the routine. The entire collection can be used to specify the results of the ReqCap mapping for administrative operations, since each administrative operation corresponds to one of the similarly named routines.

pe) in one of the capability sets returned by the required capabilities function, ReqCap(op, argseq):

- There exists a privilege (Process_User(p), ar, pe).
- There does not exist a process restriction (p, ar, pe) ∈ P_RESTRICT.
- There does not exist a user restriction (Process_User(p), ar, pe) ∈ U_RESTRICT.
- There does not exist a user attribute restriction (ua, ar, pe) ∈ UA_RESTRICT, such that Process_User(p) is contained by user attribute ua.

Otherwise, the requested access is denied.

The computation of an access decision is illustrated in Figure 5. The thick arrows depict the derivation of the privilege and restriction relations respectively from the association and prohibition relations, while the thin arrows depict the use of those derived relations in reaching an access decision.

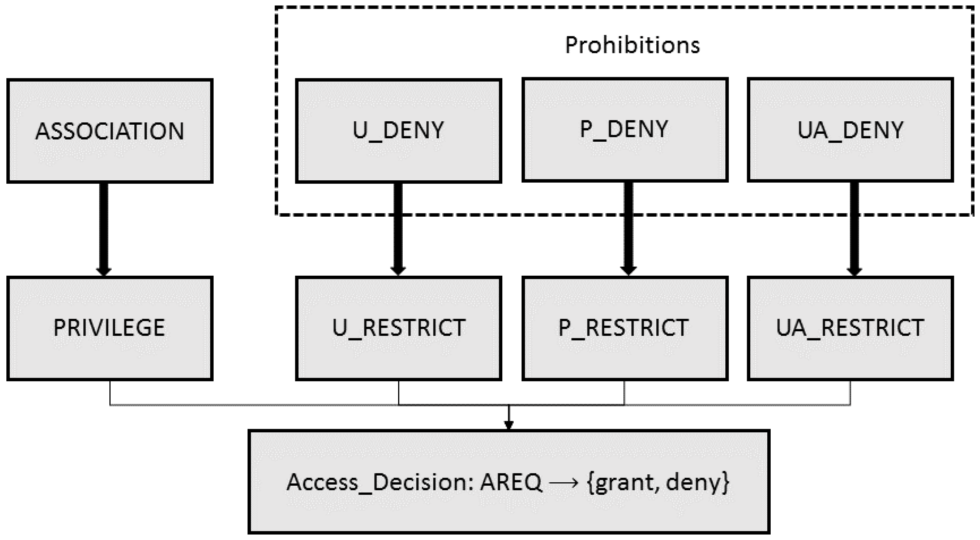

Figure 5: Computing an Access Decision

Notation for Access Request Decisions. The elements of the PM model involved in issues access requests and reaching access request decisions can be described more formally as shown below.

- **Access Request:** A finite set AREQ of possible process access requests. $AREQ \subseteq P \times Op \times Seq_1\, Arg$, where $Arg = \{x \mid x \in PE \lor x \in 2^{PE} \lor x \in 2_1^{AR} \lor x \in PATTERN \lor x \in RESPONSE\}$

- **Required Capabilities:** The partial binary function ReqCap from Op × seq_1 Arg to $2^{(AR \times PE)}$.
  - ReqCap $\subseteq$ (Op×seq_1 Arg) × $2^{(AR \times PE)}$
  - …op $\in$ Op, …argseq $\in$ seq_1 Arg: (capset $\in$ ReqCap(op, argseq) $\Rightarrow$
    $\forall$ar $\in$ AR, $\forall$pe $\in$ PE: ((ar, pe) $\in$ capset $\Leftrightarrow$ (ar, pe) is a requisite capability needed to perform the operation op on argseq))

- **Access Decision:** The function from domain AReq to codomain {grant, deny}.
  - Access_Decision $\subseteq$ AREQ × {grant, deny}
  - $\forall$p $\in$ P, $\forall$op $\in$ Op, $\forall$argseq $\in$ seq_1 Arg: ((p, op, argseq) $\in$ AREQ $\Rightarrow$
    (Access_Decision((p, op, argseq)) = grant $\Leftrightarrow$
    $\exists$capset $\in$ ReqCap(op, argseq), $\forall$ar $\in$ AR, $\forall$pe $\in$ PE: ((ar, pe) $\in$ capset $\Rightarrow$
    ((Process_User(p), ar, pe) $\in$ PRIVILEGE $\wedge$
    (p, ar, pe) $\notin$ P_RESTRICT $\wedge$ (Process_User(p), ar, pe) $\notin$ U_RESTRICT) $\wedge$
    $\nexists$ua $\in$ UA: (Process_User(p) $\in$ Users(ua) $\wedge$ (ua, ar, pe) $\in$ UA_RESTRICT)))));
    otherwise, Access_Decision((p, op, argseq)) = deny

4. Administrative Considerations

The PM model distinguishes between access request on resources represented by objects (i.e., non-administrative access) and access requests for the creation and maintenance of policy elements and relations (i.e., administrative access). The previous chapter touched mainly on policies involving non-administrative access rights to object resources, specifically, the definition of associations and various types of prohibitions, and the derivation of privileges and restrictions from them, and also the definition of obligations. This chapter expands on the material from the previous chapters and focuses on the role administrative access rights play in policy.

Many access rights categorized as administrative access rights, such as those needed to create a file and assign it to a folder, arguably seem non-administrative from a usage standpoint. Nevertheless, from a policy specification standpoint, they are considered administrative (e.g., in this case, an association with access rights for creating an object and assigning the object to an object attribute is needed). The main difference between the two types of access rights is that non-administrative actions pertain to activities on protected resources represented as objects, while administrative actions pertain to activities on the policy representation comprising the policy elements and relationships defined within and maintained by the PM. This chapter explains the principles involved in applying administrative access rights to key relations of the PM model. It also discusses the precepts to follow when conducting administrative activities.

4.1 Administrative Associations

The term administrative association refers to an association that involves administrative access rights exclusively to designate access authority. Administrative associations appear on an authorization graph, as do non-administrative associations. However, administrative associations can apply to any type of attribute, whereas non-administrative associations typically apply only to object attributes.

Administrative associations are characterized by the ternary relation from UA to 2_1^{AAR} to AT (i.e., a subset of UA $\times$ 2_1^{AAR} $\times$ AT). As mentioned in the previous chapter, when resource and administrative access rights are maintained separately within individual associations, the association relation can be correspondingly partitioned into two parts: ASSOCIATION $\subseteq$ (UA $\times$ 2_1^{RAR} $\times$ AT) $\cup$ (UA $\times$ 2_1^{AAR} $\times$ AT). Segregating associations this way presents no difficulties, since any association expressed with mixed resource and administrative access rights can be represented, without loss of meaning or function, as a pair of associations, belonging respectively to UA $\times$ 2_1^{RAR} $\times$ AT and UA $\times$ 2_1^{AAR} $\times$ AT.

The following classes of administrative access rights apply to relations 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 6 presents a simple example of an authorization graph involving both administrative and non-administrative associations, which builds on the example presented earlier in Figure 3. 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.

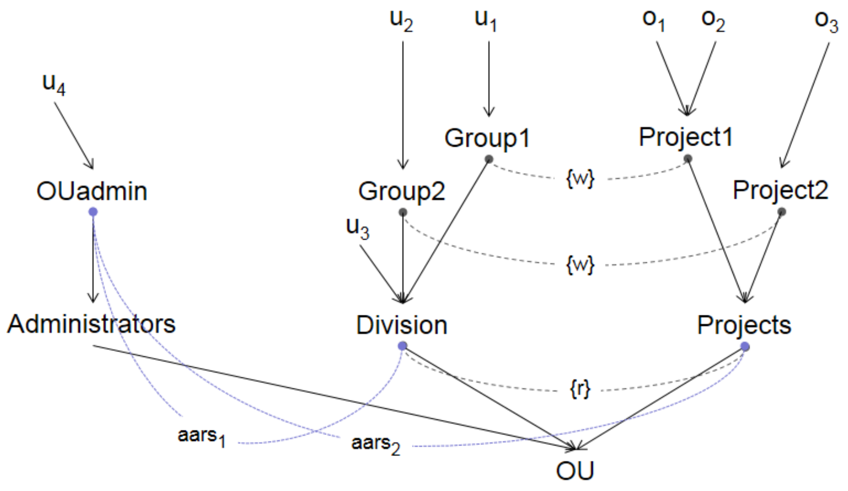

Figure 6: Simple Example Involving Administrative Associations

The authorization graph in Figure 6 contains the following two administrative associations: (OUadmin, $aars_1$, Division) and (OUadmin, $aars_2$, Projects), where $aars_1$ and $aars_2$ each represent a set of administrative access rights (i.e., each is a subset of AAR). 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, only existing objects could be viewed and modified; new users could not be created or old ones deleted and new objects could not be created without the appropriate administrative associations.

Administrative privileges are derived the same way as was described earlier for resource privileges. An administrative privilege specifies a relationship between a user, an administrative access right, and a policy element. Administrative privileges can be characterized as a subset of the PRIVILEGE relation, namely those within the subset U × AAR × (PE\PC). An individual tuple of an administrative privilege, (u, aar, e), denotes that user u holds the authority to exercise the access right aar on policy element e. For a policy comprising a single policy class, the triple (u, aar, e) is an administrative privilege, iff there exists an association (ua, ars, at) ∈ ASSOCIATION, such that user u ∈ Users(ua), aar ∈ ars, ars ∈ 2_1^{AAR}, and e ∈ Elements(at).

4.2 Administrative Prohibitions

Recall that prohibitions act antithetically to associations, 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 needed for administrative operations is distinct from the set of privileges for non-administrative, resource operations, administrative prohibitions and restrictions (i.e., prohibitions specified exclusively with administrative privileges) can be identified and discussed in detail.

The quaternary relation U_DENY_DISJ ⊆ U × 2_1^{AR} × 2^{AT} × 2^{AT}, defines the set of user-based prohibitions. Similar to associations, any prohibition tuple, (ua, {rar_i, rar_j, … , rar_m, aar_k, aar_l, … , aar_n}, atis, ates) whose access rights belong to both RAR and AAR can be represented as a pair of tuples, (ua, {rar_i, rar_j, … , rar_m}, atis, ates) and (ua, {aar_k, aar_l, … , aar_n}, atis, ates), without loss of meaning or function. When prohibitions that involve resource access rights are specified separately from prohibitions that involve administrative access rights, the user prohibition relation can be specified as U_DENY_DISJ ⊆ (UA × 2_1^{RAR} × 2^{AT} × 2^{AT}) ∪ (UA × 2_1^{AAR} × 2^{AT} × 2^{AT}).

The tuple, (u, aars, atis, ates) ∈ U_DENY_DISJ, where u ∈ U, aars ∈ 2_1^{AAR}, atis ∈ 2^{AT}, ates ∈ 2^{AT}, and atis ⋃ ates ≠ ∅, denotes that any process p, executing on behalf of user u, is withheld the the authority to exercise any of the access rights within the access right set, aars, against any policy elements that lie within the disjunctive range of the inclusion and exclusion sets of attributes, DisjRange(atis,ates). Similarly, an individual tuple (u, aars, atis, ates) of the quaternary relation U_DENY_CONJ ⊆ U × 2_1^{AAR} × 2^{AT} × 2^{AT}, denotes that any process p, executing on behalf of user u, is withheld the authority to exercise any of the access rights within the access right set, aars, over any policy element against any policy elements that lie within the conjunctive range of the inclusion and exclusion sets of attributes, ConjRange(atis,ates).

The treatment of process-based prohibitions to accommodate the bifurcation of administrative access rights from resources access rights is similar to those for user-based prohibitions. The disjunctive process prohibition relation can be specified as P_DENY_DISJ ⊆ (P × 2_1^{RAR} × 2^{AT} × 2^{AT}) ∪ (P × 2_1^{AAR} × 2^{AT} × 2^{AT}), and the conjunctive process prohibition relation as P_DENY_CONJ ⊆ (P × 2_1^{RAR} × 2^{AT} × 2^{AT}) ∪ (P × 2_1^{AAR} × 2^{AT} × 2^{AT}). The same treatment also applies to user attribute-based prohibitions. The disjunctive user attribute prohibition relation can be specified as UA_DENY_DISJ ⊆ (UA × 2_1^{RAR} × 2^{AT} × 2^{AT}) ∪ (UA × 2_1^{AAR} × 2^{AT} × 2^{AT}), and the conjunctive user attribute prohibition relation as UA_DENY_CONJ $\subseteq$ (UA $\times$ 2_1^{RAR} $\times$ 2^{AT} $\times$ 2^{AT}) $\cup$ (UA $\times$ 2_1^{AAR} $\times$ 2^{AT} $\times$ 2^{AT}).

4.3 Administrative Obligations

Conceptually, 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 administrative routines also constitutes the response for an administrative obligation, and the event context for administrative events follows the same arrangement as that for non-administrative events. The main distinction is that administrative obligations are triggered by events that occur only when an administrative access request is approved by the access decision function and successfully carried out.

4.4 Administrative Access Request Decisions

Access requests bearing an administrative operation are somewhat different from those bearing a non-administrative operation. Recall that for an individual access request, (p, op, argseq) $\in$ AREQ, op designates an operation involving the operands (i.e., resources and policy items) referenced in the argument sequence, argseq. The arguments in the sequence can be one of the following items: a distinct policy element, a set of policy elements, an event pattern, a response, and a set of access rights. The access request relation can be conveniently partitioned by the type of operation involved into two parts: AREQ $\subseteq$ (P $\times$ ROP $\times$ seq_1 Arg) $\cup$ (P $\times$ AOP $\times$ seq_1 Arg), where Arg = {x | x $\in$ PE $\lor$ x $\in$ 2^{PE} $\lor$ x $\in$ 2_1^{AR} $\lor$ x $\in$ PATTERN $\lor$ x $\in$ RESPONSE}. When the access request contains a resource operation, the argument sequence contains only a single item representing an object resource of interest. For an administrative operation, however, the access request typically involves not just a single item, but multiple items, needed to enact the desired policy.

The order of the arguments in an argument sequence for an administrative access request is significant, as is the number and type. For instance, the creation of an assignment between two object attributes, (p, create-OAtoOA, $\langle oa_i, oa_j \rangle$), is completely different from one where the order is reversed, (p, create-OAtoOA, $\langle oa_j, oa_i \rangle$). Exactly two policy elements are required by the create-OAtoOA operation: the first in the sequence corresponding to the tail of the assignment and the second to the head. In contrast, an administrative access request containing the administrative operation, create-Assoc, to create a read association between a user attribute and an object attribute, (p, create-Assoc, $\langle ua_i, \{r\}, oa_j \rangle$), requires an argument sequence of exactly three arguments: a user attribute, a set of access rights, and an object attribute.

The computation of an access decision involving an administrative operation proceeds as described in the previous chapter. For an administrative operation to be granted, the user on whose behalf the process operates must hold sufficient authority over the policy items in the argument sequence in the form of at least one, and possibly more, administrative privileges, and any restrictions to the contrary must not exist.

4.5 Administrative Commands and Routines

Administrative commands and routines are the means by which policy specifications are formed. Each access request involving an administrative operation corresponds on a one-to-one basis to an administrative routine, which uses the sequence of arguments in the access request to perform the access. As described in the previous section, Section 4.4, the access decision function grants the access request (and initiation of the respective administrative routine), only if the process holds all prohibition-free access rights over the items in the argument sequence needed to carry out the access. The administrative routine, in turn, uses one or more administrative commands to perform the access. The diagram in Figure 7 gives an overview of the role administrative routines and commands in the treatment of administrative access requests.

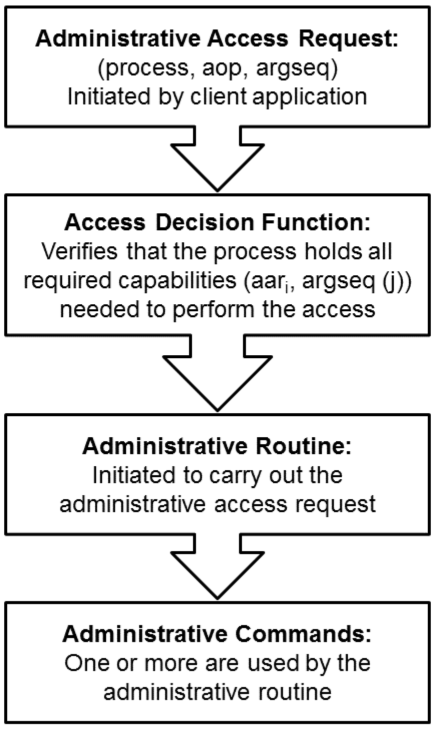
Administrative Access Request:
(process, aop, argseq)
Initiated by client application
Access Decision Function:
Verifies that the process holds all required capabilities (aar_i, argseq (j)) needed to perform the access
Administrative Routine:
Initiated to carry out the administrative access request
Administrative Commands:
One or more are used by the administrative routine

Figure 7: Administrative Routines and Commands in Access Request Processing

The syntax and notation for administrative commands and routines are discussed in detail below. The list of core administrative commands and routines for the PM, along with a description of their semantics, are presented respectively in Appendix C and Appendix D.

4.5.1 Administrative Commands

Administrative commands describe rudimentary operations that alter the policy elements and relationships of the PM model, which comprise the authorization state. An administrative command is represented as a parameterized procedure, whose body describes state changes to policy that occur when the described behavior is carried out (e.g., a policy element or relation Y changes state to Y′ when some function f is applied). Administrative commands are specified using the following format:

```
Cmdname ($x_1$: $type_1$, $x_2$: $type_2$, …, $x_k$: $type_k$)
    … *preconditions* …
        {
        $Y' = f(Y, x_1, x_2, …, x_k)$
        }
```

The name of the administrative command, Cmdname, precedes its declaration of formal parameters, x_1: $type_1$, x_2: $type_2$, …, x_k: $type_k$ ($k \geq 0$). Comments may appear anywhere in a command. Single line comments begin with double, forward slashes (i.e., // *comment*); multiple line comments begin with a forward slash and asterisk and end with an asterisk and forward slash (i.e., /* *comment* */). A set of preconditions preface the body of the command, which is delineated by left and right braces (i.e., { *body* }). Preconditions are logical expressions that must be satisfied for the command to be invoked. Complete predicate formulas that appear on consecutive lines are conjoined together by default (i.e., the logical "and" operation, $\wedge$, is implied between them). No state changes described in the body can occur unless the preconditions are satisfied. Preconditions for administrative commands are used to ensure that the arguments supplied to the command are valid and that policy elements and relationships are maintained consistently with the properties of the model.

Consider, as an example, the administrative command CreateAssign shown below, which specifies the creation of an assignment between policy elements. The x and y parameters of the command both have the type ID (i.e., identifier). The preconditions stipulate that x and y are valid members of the set of policy elements of the model and are compatible pair of policy elements, conformant to the ASSIGN relation. The stated precondition also prevent assignment loops and cycles from occurring. The body of the command describes the addition of the ordered tuple (x, y) to the ASSIGN relation, which changes the state of that relation, denoted as ASSIGN′. If the tuple (x, y) was a member of ASSIGN to begin with, the relation is unchanged (i.e., ASSIGN′ = ASSIGN).

```
CreateAssign (x: ID, y: ID)
    $x \in PE$
    $y \in PE$
    // *restrict assignments to compatible pairs of policy elements*
    $((x \in U \wedge y \in UA) \vee (x \in UA \wedge y \in UA) \vee (x \in UA \wedge y \in PC) \vee$
    $(x \in OA \wedge y \in (OA \setminus O)) \vee (x \in (OA \setminus O) \wedge y \in PC))$
    // *prevent the creation of a loop*
    $x \neq y$
    $(x, y) \notin ASSIGN$
    // *prevent the creation of a cycle from x to y*
    $(x, y \in UA \vee x, y \in OA) \Rightarrow \nexists s \in iseq_1\ PE: (\#s > 1 \wedge \forall i \in \{1,...,(\#s - 1)\}:$
    $((s(i), s(i+1)) \in ASSIGN) \wedge (s(1) = y \wedge s(\#s) = x))$
    {
        $ASSIGN' = ASSIGN \cup \{(x, y)\}$
    }
```

Each administrative command 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, the creation or deletion of other policy entities involved in relations, or the creation or deletion of an association, prohibition, or obligation relation. Compared to administrative routines, which are discussed in the next section, administrative commands are elementary. That is, administrative commands provide the foundation for the PM framework, while administrative routines use one or more administrative commands to carry out their function. Both must perform their intended function correctly and without unwanted side effects. Access to these security-critical components must be restricted, since their deletion, replacement, or modification by unauthorized parties could affect the confidentiality, integrity, and availability of the system.

4.5.2 Administrative Routines

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

```
Rtnname ($x_1$: $type_1$, $x_2$: $type_2$, …, $x_k$: $type_k$ )
  … preconditions …
      {
      $cmd_1$;
      $condition_a$ $cmd_2$, $cmd_3$;
      . . .
      $condition_z$ $cmd_n$;
      }
```

The name of the administrative routine, Rtnname, precedes the routine's declaration of formal parameters, x_1: $type_1$, x_2: $type_2$, …, x_k: $type_k$ ($k \geq 0$). Each formal parameter of an administrative routine can serve as an argument in any of the administrative command invocations, cmd_1, cmd_2, …, cmd_n ($n \geq 0$), that make up the body of the routine, and also in any condition prepended to a command. As with an administrative command, the body of an administrative routine is prefixed by preconditions, which in general, ensure that the arguments supplied to the routine are valid, and that certain properties on which the routine relies are maintained. As illustrated above, an optional condition can precede one or more of the commands.

Administrative routines are used in a variety of ways. Figure 8 gives an overview of the types of usage possible.

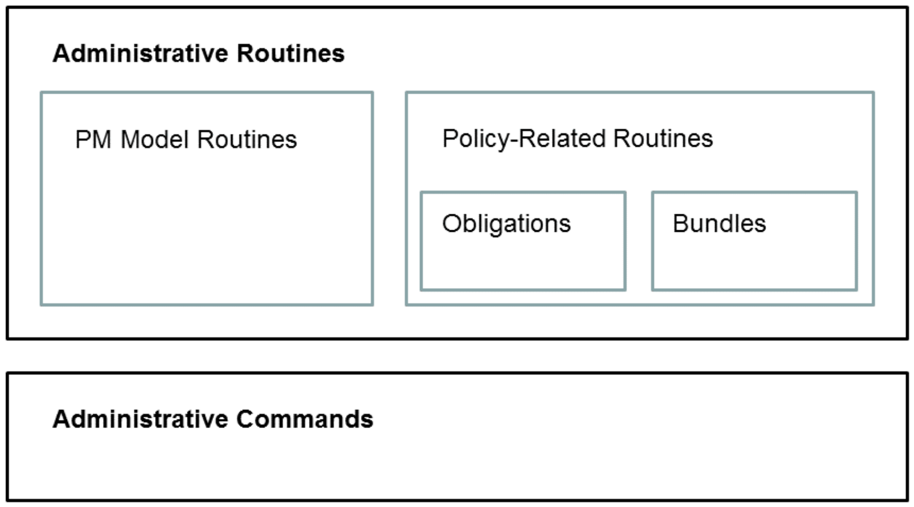

Figure 8: Administrative Routines and Commands

First and foremost, administrative routines are used to define services of the PM model. An administrative routine must be in place to carry out each valid administrative access request of the PM model on a one-to-one basis. The PM model routines form the trusted computing base of the PM framework.

Another common use of administrative routines is in the definition of an obligation, as the response to be taken whenever the corresponding event pattern is matched. It is important to note that administrative routines used to define system policy through an obligation response are distinct from those that define services of the PM model and are used to fulfill administrative access requests. Not only are they invoked differently, but the authorization requirements for each are also different.[4] Another way of explaining the situation is that PM model routines are unsuited for and not usable with obligations. The most common types of administrative routines 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 routines 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 through assignments, and through associations, grants the authority for the user to access the containers as its work space. Rather than manually repeating each step of this sequence of administrative actions for each new user, the entire sequence of actions can be defined as a single administrative routine and executed in its entirety as an atomic action.

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 routine that is capable of building

[4] The preconditions of administrative routines defined for use in obligations require that the user who defined the obligation (not the triggering process) holds sufficient authorization to execute all constituent administrative commands of the body, while the preconditions of administrative routines for the PM model require that the process attempting the access holds sufficient authorization to execute all constituent administrative commands of the body.

an entire system policy. This type of bundling would allow an established policy to be instantiated quickly elsewhere, and also allow libraries of routines containing various kinds of vetted policies or policy enhancements to be assembled and shared on a broad scale.

4.5.3 Administrative Actions

Within the PM framework, several basic precepts govern the actions that a user with administrative authority can take when using administrative routines and commands 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 Alpha creates a policy element Beta, it obtains a reference to the newly created Beta, 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 Beta, it may then assign Beta to an existing compatible policy element for which it holds authority and thereby gain additional authority over Beta through the inheritance of properties. The authority that Alpha holds over Beta and other policy elements may in turn allow Alpha to define additional relationships among them or to delete exist relationships.

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

- Automation. A user with sufficient administrative authority may define obligations that carry out a set of predefined activities on behalf of the user, based on the occurrence of specific types of events.[5]

Administrative actions may be conducted through a graphical user interface that renders the authorization graph of a policy for an administrator to view and manipulate to perform desired modifications.

[5] Specifically, the user needs to hold "obligate" access rights over all identifiable policy elements present in the pattern and response portions of the obligation.

5. Policy Specification

This chapter consolidates the concepts and material given in the previous two chapters. It provides additional details about formulating policy specifications, including the application of one or more levels of administration and policy, and the use of access rights to govern and constrain the activities of users and administrators. Examples are given to illustrate key points of 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 detailed examples are provided below to illustrate how aspects of the model can be brought together to define a specific access control policy. The examples involve a data service for electronic mail and discretionary access control for an operating system. They follow the principal mentioned earlier of not mixing resource and administrative access rights in association and prohibition relations. The examples are also 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 must first 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 9 illustrates the policy element diagram constructed so far.

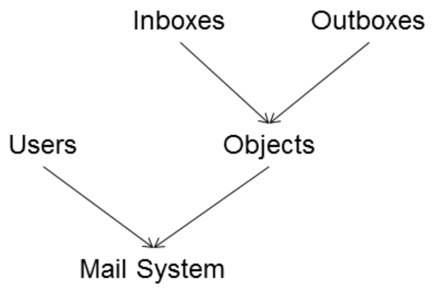

Figure 9: Partial Policy Element Diagram for the Mail System

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 the attribute 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, Out u_i is assigned to the Outboxes container, and Other u_i is assigned to the Objects container. The remaining containers, Draft u_i and Trash u_i, are assigned to Other u_i.[6]

Figure 10 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 resource operations on mail objects (i.e., messages) in its containers. Associations are represented by the dotted, downward-arcing connectors and prohibitions by the dotted, upward-arcing connectors. No mail objects are shown in the figure.

[6] The administrative activities described could be performed individually for each user, as described. Alternatively, an administrative routine bundle could be defined once and invoked for each user to carry out all the activities as an atomic action.

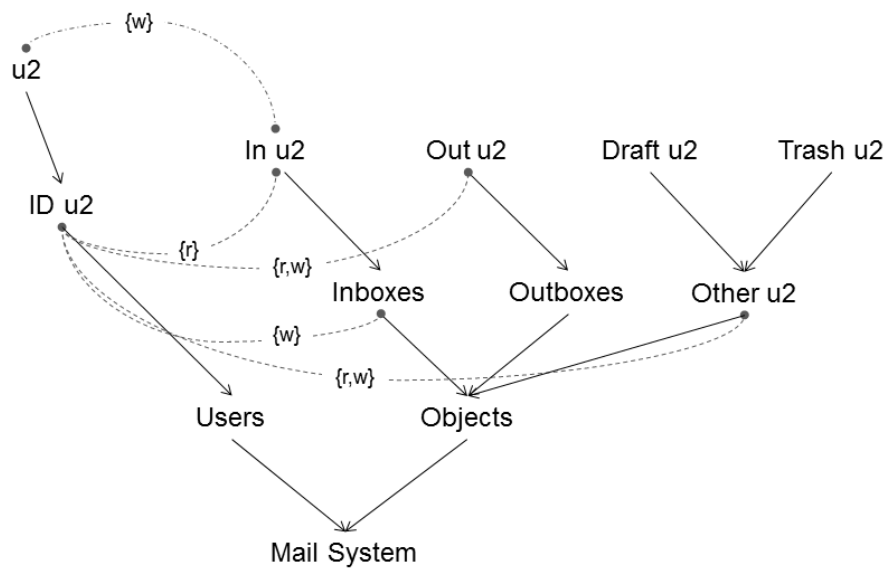

Figure 10: Authorization Graph for the Mail System

The user u_2 can read objects in its inbox, In u_2, and can read and write objects within its outbox, Out u_2, 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_2 and Trash u_2 containers), which are contained by 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.

The administrative associations and prohibitions specified for the user complement those shown above and define the remaining actions for the user. Figure 11 illustrates the authorization graph containing the additional associations and prohibitions needed.

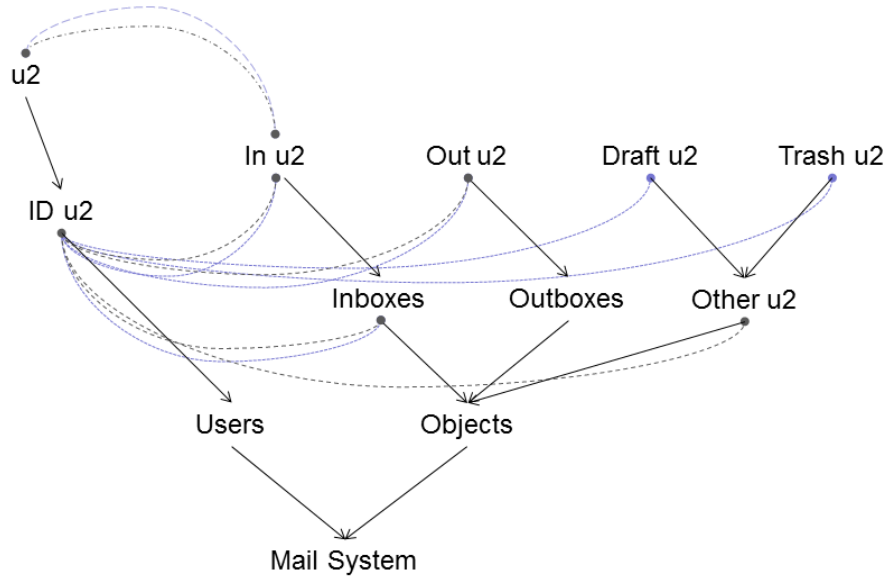

Figure 11: Authorization Graph with Administrative Associations and Prohibitions

The rationale behind the associations and prohibitions that are illustrated in Figure 11 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 to 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.

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 user's outbox, once they are written to it. The obligation presumes that draft messages are composed in the sender's drafts folder, and when ready to be sent, copied over in their entirety to newly created message objects in the sender's outbox, before being deleted from the drafts folder. The following obligation accomplishes the write-once restriction through the creation of a user prohibition:

When EC.op = write $\wedge$ EC.o ASSIGN$^+$ Outboxes **do**
CreateOblig-DisjUProhib (EC.c, EC.u, {w}, {EC.o}, ∅)[7,8]

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 there. 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

[7] 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.

[8] The semantics of the administrative routine used in this obligation is essentially the same as that for the routine C-DisjUProhib given in Appendix D, with one exception—the user that defined the obligation (viz., EC.c), not the user whose process triggered the obligation, must hold sufficient authorization to perform the body of the routine.

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 = write $\wedge$ EC.o ASSIGN$^+$ Inboxes **do**
>
> CreateOblig-DisjUAProhib (EC.c, Users, {w}, {EC.o}, ∅)[9]

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 the 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.

Note that this example presents an ideal situation in which administrative routines could be used to facilitate the specification of policy. The entire sequence described in the example: creating containers, linking them together through assignments, authorizing the user access to its work space of containers through associations and prohibitions, and imposing constraints on access through obligations, can be defined as a single administrative routine and executed in its entirety as an atomic action by the administrator for each new 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 of interest are files and folders. The latter items also serve as object attributes or containers 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 Objects container. Figure 12 illustrates the policy element diagram constructed, along with the appropriate associations, for two typical users, u_2 and u_3.

[9] As with the administrative routine in the previous obligation, the C-DisjUAProhib routine given in Appendix D has essentially the same semantics, with the caveat of the authorization imposed on the creator of the obligation, instead of the user whose process triggered the obligation.

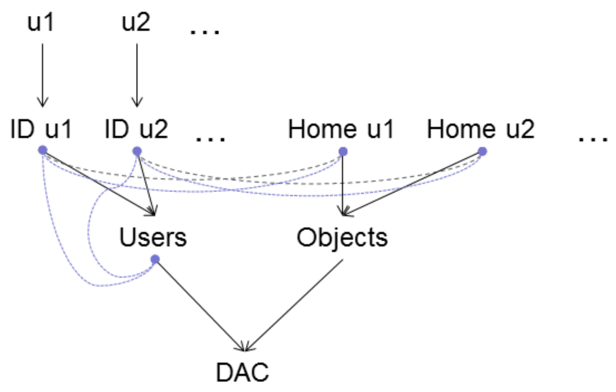

Figure 12: Authorization Graph for the DAC Operating System

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 u_1 to involve any user contained by 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}, as illustrated in Figure 13, 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}.

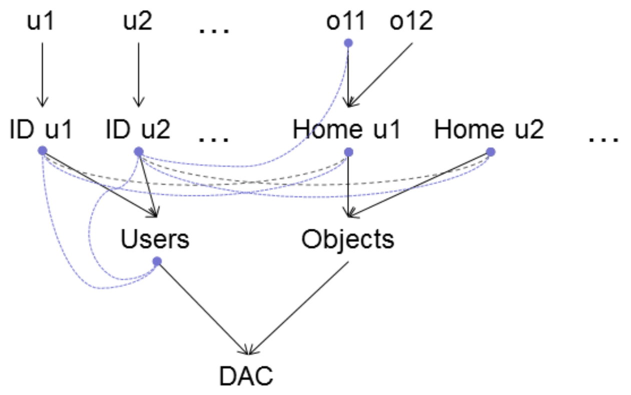

Figure 13: DAC Authorization Graph with Objects

As plainly evident, the policy defined for the users of the operating system entails discretionary control over certain objects. The policy grants a user not only the authority to perform common operating system activities, such as creating or deleting files and folders their home container, but also authority to form and rescind associations that allow other users access to files and folders under its home container. As with the previous example, the specification of the policy described here can be facilitated through the definition and use of an administrative routine that carries out the required steps for each new user.

5.2 Levels of Policy and Administration

As we have seen, associations and prohibitions are the principal means of performing delegation within the PM model. These means of delegation can be used to allow an administrator to empower other administrators or users with some of the rights the administrator hold. The definition of a single level or multiple levels of administrative authority are possible. In principle, 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 universal authorization to carry out those activities within the PM framework. The PA creates policy classes and first-level attributes that define a subordinate administrator and a domain for the subordinate administrator to manage, then allocates sufficient privileges to the subordinate administrator to perform those duties. The PA is a trusted entity and the only authority with the ability to create a policy class, which is the foundation of policy formation. Multiple domains and administrators can be created by the PA at its discretion. The PA can also forego the use of any subordinate administrators and manage a domain itself.

- The Domain Administrator (DA) is a subordinate administrator of the PA, which can create users, objects, and attributes within its domain and manage the entire domain itself, as prescribed by the PA. A DA can also define a subdomain of its domain and allocate sufficient privileges for a subordinate 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 subdomain under its control. The pattern of subdividing the subdomain, 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 establishes the policy for a system, creating the requisite domains and domain administrators, 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 policy entity acquires unintended access authorization.

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 14 below illustrates one such pattern. Consistent with existing conventions, the solid-line arrows represent assignments between policy elements. The dotted, downward-arcing 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.

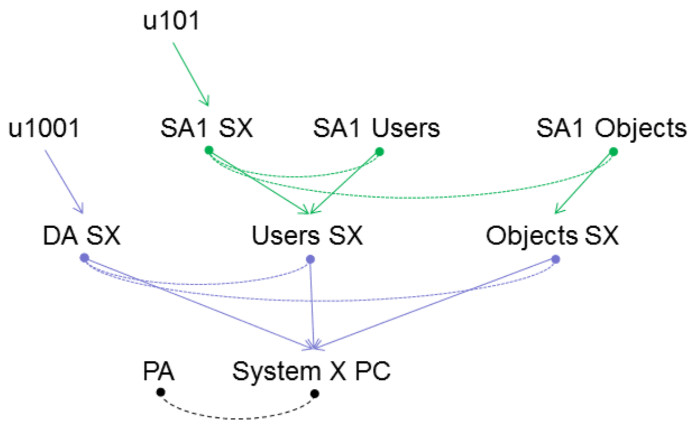

Figure 14: Pattern of Authority Levels

The assignment and association connectors are colored to convey which authority established the policy. PA's control over the System X policy class is depicted in black and denotes 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 SX), 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 14 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 15 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. 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.

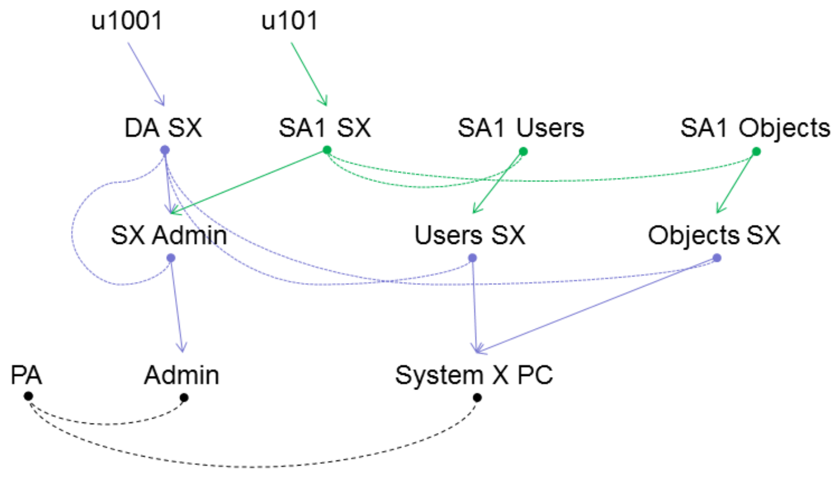

Figure 15: An Alternative Pattern of Authority Levels

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. 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 common workaround 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 chapter, Appendix C, and Appendix D. Appendix E provides a detailed discussion of three 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 the use of multiple levels of authority and policy, the examples of Section 5.1 are reexamined in light of the above discussion. For simplicity, the PA and the authorization it holds over policy creation is implied, but not depicted in the examples.

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 14. Figure 16 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.

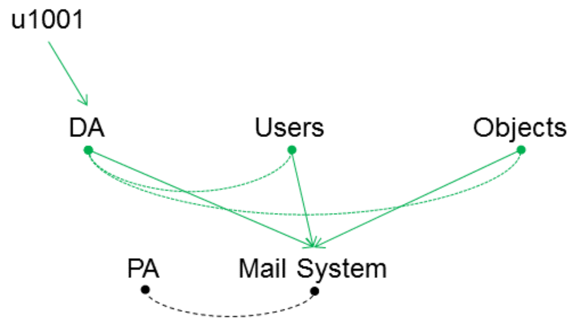

Figure 16: Policy Assignments and Associations for the Mail System DA

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 17 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.

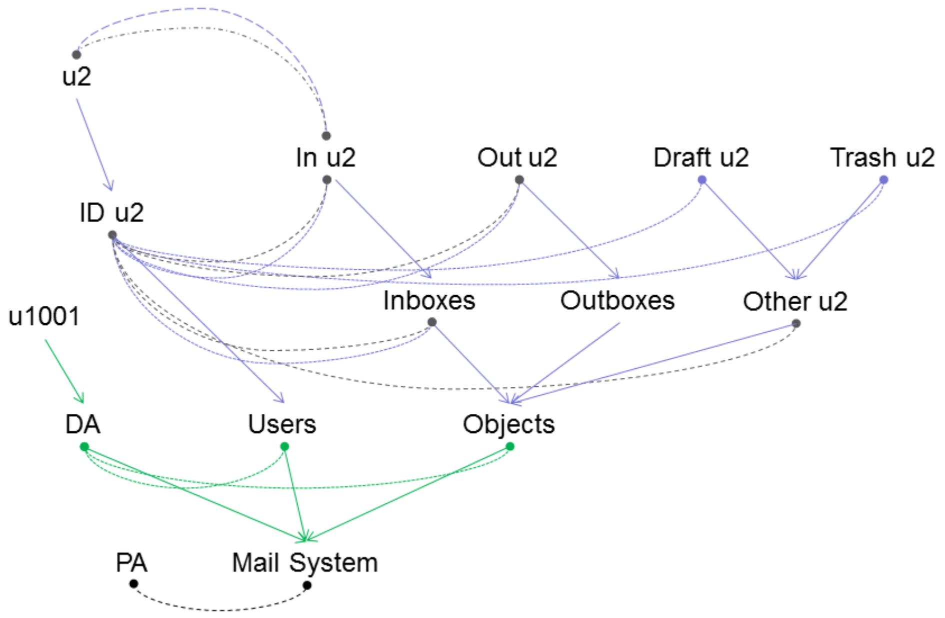

Figure 17: Authorization Graph for the Mail System

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 18 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.

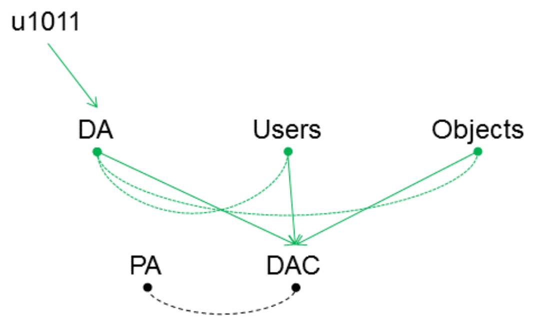

Figure 18: Policy Elements and Associations for the DAC DA

Figure 19 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 by its home container. The DA also has the discretion to create subdomains and assign an SA to them, if it chooses.

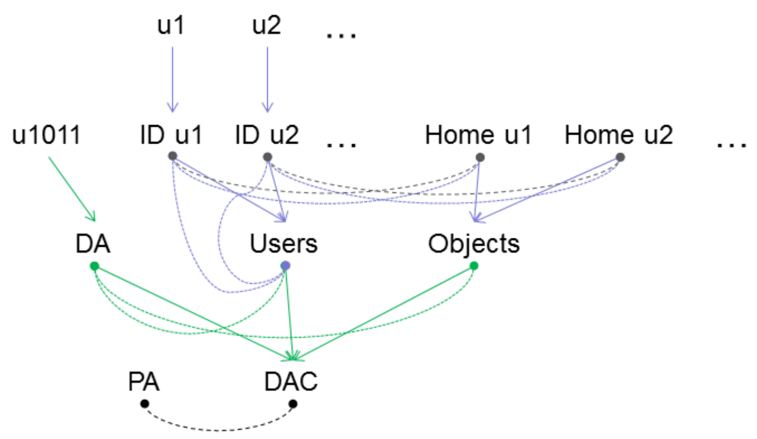

Figure 19: Authorization Graph for the DAC Operating System

5.4 Generic Access Rights

Generic access rights 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 access rights needed to structure policy for a system and establish levels of administration for governing the policy over its lifetime. These generic access rights 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 in detail a core set of access rights for the PM model and explains how such authority can be utilized to specify access control policy.

5.4.1 Classes of Access Rights

Two general classes of generic access rights exist. They are non-administrative access rights that pertain to protected resources represented by objects, and administrative access rights that pertain to a policy specification comprising the policy elements and relationships defined within and maintained by the PM. Table 1 and Table 2 enumerate the default access rights for each class, identifying the type of access mode, its designation, the type of policy element to which the mode of access applies, and the policy element or relation affected.

The first class of generic access rights, shown in Table 1, falls into one type of access mode: the input and output of data to and from protected resources represented by objects. Protected resources may be logical (e.g., files and folders) or physical (e.g., printers and networking components). As mentioned previously, for non-administrative access rights, resource operations are synonymous with the access rights needed to carry out those operations: to output data or write to an object requires "w" (i.e., write) authority, and to input data or read from an object requires "r" (i.e., read) authority.

Table 1: Generic Non-administrative Access Rights

Type	Non-admin. Access Right	Applies to	Affects
Input/Output Resources	r	Object attribute	Protected resources represented by the object attribute or an object contained by the object attribute
	w	Object attribute	Protected resources represented by the object attribute or an object contained by the object attribute

The second class of generic access rights, shown in Table 2, relate to one of four types of administrative modes of access: 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. The generic administrative access rights listed for each type of access modality follows the naming conventions for policy elements established in the previous chapters. Unlike resource access rights, the authority associated with an administrative access right is not necessarily synonymous with an administrative operation. Instead, the authority stemming from one or more administrative access rights may be required for a single operation to be authorized.

Table 2: Generic Administrative Access Rights by Type

Type	Administrative Access Right	Applies to	Affects
Create/Delete Policy Elements	c-u, d-u	User attribute	Actualization of a user policy element
	c-ua, d-ua	User attribute or Policy class	Actualization of a user attribute policy element
	c-o, d-o	Object attribute	Actualization of an object policy element
	c-oa, d-oa	Object attribute or Policy class	Actualization of an object attribute policy element
	c-pc, d-pc	The PM framework	Actualization of a policy class policy element
Create/Delete Assignments (Intra-Policy Class)	c-uua, d-uua	User attribute	Assignment from a user to the user attribute
	c-uaua, d-uaua	User attribute	Assignment from a user attribute to the user attribute
	c-uapc, d-uapc	Policy class	Assignment from a user attribute to the policy class
	c-ooa, d-ooa	Object attribute	Assignment from an object to the object attribute
	c-oaoa, d-oaoa	Object attribute	Assignment from an object attribute to the object attribute
	c-oapc, d-oapc	Policy class	Assignment from an object attribute to the policy class

Type	Administrative Access Right	Applies to	Affects
Create/Delete Assignments (Inter- or Intra-Policy Class)	c-uua-fr, d-uua-fr	User attribute	Assignment from a user element in the referent's subgraph to a user attribute
	c-uua-to, d-uua-to	User attribute	Assignment from a user to a user attribute element in the referent's subgraph
	c-uaua-fr, d-uaua-fr	User attribute	Assignment from a user attribute element in the referent's subgraph to a user attribute
	c-uaua-to, d-uaua-to	User attribute	Assignment from a user attribute to a user attribute element in the referent's subgraph
	c-uapc-fr, d-uapc-fr	User attribute	Assignment from a user attribute element in the referent's subgraph to the policy class
	c-uapc-to, d-uapc-to	Policy class	Assignment from a user attribute to the policy class
	c-ooa-fr, d-ooa-fr	Object attribute	Assignment from an object element in the referent's subgraph to an object attribute
	c-ooa-to, d-ooa-to	Object attribute	Assignment from an object attribute to an object attribute element in the referent's subgraph
	c-oaoa-fr, d-oaoa-fr	Object attribute	Assignment from an object attribute in the referent's subgraph to an object attribute element
	c-oaoa-to, d-oaoa-to	Object attribute	Assignment from an object attribute element to an object attribute in the referent's subgraph
	c-oapc-fr, d-oapc-fr	Object attribute	Assignment from an object attribute element in the referent's subgraph to a policy class
	c-oapc-to, d-oapc-to	Policy class	Assignment from an object attribute to the policy class
Create/Delete Multi-way Relationships	c-assoc-fr, d-assoc-fr	User attribute	Association involving a user attribute element in the user attribute's subgraph and a referent attribute
	c-assoc-to, d-assoc-to	Reverent attribute	Association involving a user attribute and a policy element in the referent attribute's subgraph
	c-prohib-fr, d-prohib-fr	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 a policy element
	c-prohib-to, d-prohib-to	Set of referent attributes	Prohibition involving a user, user attribute, or process, and an policy element inside or outside the referent attribute's subgraph
	c-oblig, d-oblig	Policy element	Obligation on an access request involving the user or process holding authorization

Type	Administrative Access Right	Applies to	Affects
Input/Output Resources Delegation[10]	r-del, w-del	Object attribute	Delegation of read or write access (i.e., r or w access rights) via associations or prohibitions, without possessing the ability to access the resources in question

As described above, administrative access rights convey the authority to manipulate policy elements and relations maintained by the PM, and thereby institute or update the policy specification for a system. The prefix "c-" (i.e., the designation for create) denotes the reification of a policy element or a relationship between policy elements, as designated by its stem. The prefix "d-" denotes the opposite. Two or more administrative access rights are often required to carry out a single administrative action on the policy representation. For example, some administrative access rights are explicitly divided into two parts, as denoted by the "from" and "to" suffixes (viz., "-fr" and "-to"). 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. A user must hold c-assoc-fr authority over a user attribute and c-assoc-to authority over an object or user attribute to form an association between those attributes or the attributes within the subgraph of each.

Administrative access rights are typically allocated in conjunction with one another. For example, the authority to create a user (c-u) with respect to a user attribute is not useful, if the authority to assign the user to the user attribute (c-uua) is not also held, since both access rights are needed to carry out the creation of the user (i.e., via the create-UinUA operation). Similarly, the authority to create a policy element such as a user (c-u) and assign it (c-uua) is not useful, if the authority to delete the policy element (d-u) and disassign it (d-uua) is not also held for the user attribute. The correspondence between administrative access rights and administrative activities is evidenced in the precondition comments of the administrative routines listed in Appendix D.

Because of the unique set of duties entrusted to the PA, its access rights are handled somewhat differently from other users and administrators. The PA implicitly holds universal authorization, denoted as the access right "univ," over the entire PM framework. The univ access right is treated as the union of all the access rights listed in Table 1 and Table 2. Except for c-uapc, c-oapc, c-pc, and d-pc, the PA can delegate all other access rights to subordinate authorities.

5.4.2 Other Considerations

While the lists of generic access rights in Table 1 and Table 2 are complete, they are 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 access rights to capture and designate the appropriate

[10] Note that r-del and w-del are the counterparts to the r and w access rights. While they do not impart the ability to read or write a resource, they do allow a holder to delegate r, w, r-del, and w-del to others via associations and prohibitions, provided that the holder has the required authorization to form those relations. Their intended use is to grant DAs and SAs the ability to manage resource access for the users of a subdomain for which they are responsible, but also to restrict their ability to perform the access themselves.

authority. For example, a requirement for higher level of assurance may dictate more granular access rights, or a requirement for compliance with some prevailing law, regulation, or policy may necessitate additional access rights. 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, and effect the set of access rights required. 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 to allow more granular control, and an explicit execute operation defined for this purpose (e.g., requiring an "e" access right to perform).

One other consideration that can influence the set of access rights is usability. While granular access rights 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 access rights that are usually assigned together (e.g., c- and d- access rights) into distinct sets and use those sets in lieu of individual access rights to assign a broad range of authority collectively when defining policy.

In summary, access rights are abstractions for the types of authorization possible within a computational environment to support a given policy and, as abstractions, may be adjusted to fit a unique situation. It can also be the case that only some subset of the access rights 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, involving access rights such as c-prohib-to/from or c-oblig, were not required.

6. Multiple Policy Class Considerations

Some policy specifications, such as the inter-policy class pattern for expressing levels of administration and policy 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 brought together and overlap to the extent that some policy elements 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 expressed using a single policy class.

The examples of 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 current description of PM framework are needed. These adjustments specifically involve refining the way privileges are derived for objects that are contained by 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

Recall that the association relation is defined as a ternary relation of the form ASSOCIATION $\subseteq$ UA $\times$ 2_1^{AR} $\times$ AT. Deriving privileges from a triple (ua, ars, at) $\in$ ASSOCIATION involves identifying all users that are contained by the first element of the triple, ua, all access rights that are members of the second element of the triple, ars, and all policy entities that are contained by the attribute, at, plus the attribute itself. Each combination of members from the three resultant sets forms a valid privilege of the form (u, ar, e).

A major difference when deriving privileges from associations in specifications that involve multiple policy classes is that for any overlapping objects and attributes, the privileges must hold for each policy class. More formally, the triple (u, ar, e) is a PM privilege, iff for each policy class pc that contains the target element e, there exists an association (ua, ars, at) $\in$ ASSOCIATION, such that user u ASSIGN$^+$ ua$_i$, ar $\in$ ars, ars $\in$ 2_1^{AR}, e ASSIGN* at, and at ASSIGN$^+$ pc. That is, a privilege involving a policy element is valid, iff it can be derived with respect to each of the policy classes that contain the element. This method of derivation works equally well when only a single policy class prevails, since all policy elements 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, ar, e) is a privilege, iff, for each policy class pc that contains e, there exists a user attribute ua with an assigned or inherent capability (ars, at), such that e ASSIGN* at, at ASSIGN$^+$ pc, u ASSIGN ua, and ar $\in$ ars. Similarly, the triple (u, ar, e) is a privilege, iff for each policy class pc that contains e, there exists an attribute at with an assigned or inherent access entry (ua, ars), such that e ASSIGN* at, at ASSIGN$^+$ pc, u ASSIGN$^+$ ua$_i$, and ar $\in$ ars.

6.2 Prohibition Refinements

As discussed earlier, prohibitions are used to override access to object resources or data elements and relations that constitute policy, based on whether the target entities are contained within or not contained within a set of attributes. Prohibitions in multiple policy class situations, like associations, also follow the precept that a policy element and the policy classes containing the element have relevance when determining the scope of a prohibition. The scope of a prohibition can be defined as the set of policy elements delineated by the inclusive and exclusive attribute sets of the relation. When multiple policy classes apply, wherever the scope of a prohibition overlaps with that of a policy class, and the prohibition affects some, but not necessarily all of the same entities within the policy class, difficulties in the specification and interpretation of policies can arise.

In both the conjunctive and disjunctive classes of user-based prohibitions, any member of the inclusory set of attributes, $atis \in 2^{AT}$, which is contained by a policy class does not present a problem in a multiple policy class situation, since for each inclusory attribute, the scope of the prohibition is always a subset of the scope of any policy class that contains the object attribute, and therefore, affects only the policy elements within those policy classes. Therefore, in multiple policy class situations where the exclusory set of attribute, $ates \in 2^{AT}$, is equal to the empty set and atis is not, the existing definitions for prohibitions apply, without issue. However, when the reverse is true, and atis is equal to the empty set and ates is not, issues arise in the context of multiple policy classes. That is, for each exclusory attribute that is contained by multiple policy classes, the policy entities affected include not only entities that are contained by the policy class, but also entities that fall outside the policy class into one or more other policy classes. The same issue applies as well to disjunctive and conjunctive, process- and attribute-based prohibitions.

It is possible to redefine 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 inclusory attribute in the definition that is contained solely by that policy class. Moreover, in some cases, the broader unconstrained scope of the exclusory attributes of a prohibition may effectuate the target policy more readily and produce the desired effect. For these reasons, no redefinition of prohibitions is necessary; restrictions are derived as described earlier, with the same effect on access requests. Nevertheless, caution is advised when defining prohibitions involving exclusory object attributes, to avoid unwanted effects.

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 access decision 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 PRIVILEGE from U to AR to PE\PC.
  - $\mathrm{PRIVILEGE} \subseteq U \times AR \times PE \backslash PC$
  - $\ldots u \in U, \ldots ar \in AR, \ldots e \in (PE \backslash PC): ((u, ar, e) \in \mathrm{PRIVILEGE} \Leftrightarrow \ldots pc \in PC:$ $(e\ \mathrm{ASSIGN}^{+}\ pc \Rightarrow \exists ars \in 2_{1}^{AR}, \exists ua \in UA, \exists at \in AT: ((ua, ars, at) \in \mathrm{ASSOCIATION} \wedge$ $u \in \mathrm{Users}(ua) \wedge ar \in ars \wedge e \in \mathrm{Elements}(at) \wedge at \in \mathrm{Elements}(pc))))$

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 defined. For example, rather than having redundant user policy elements to represent a user 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 duties may be allocated exclusively to certain authorities to effectuate 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 described 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 should be allocated. 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 respectively enabling and disabling the user's capability to use the mail system.

Figure 20 gives an example of a partial authorization graph illustrating the DAC segment of the integrated DAC-Mail System policy. As before, user u_{1011} 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), who 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.

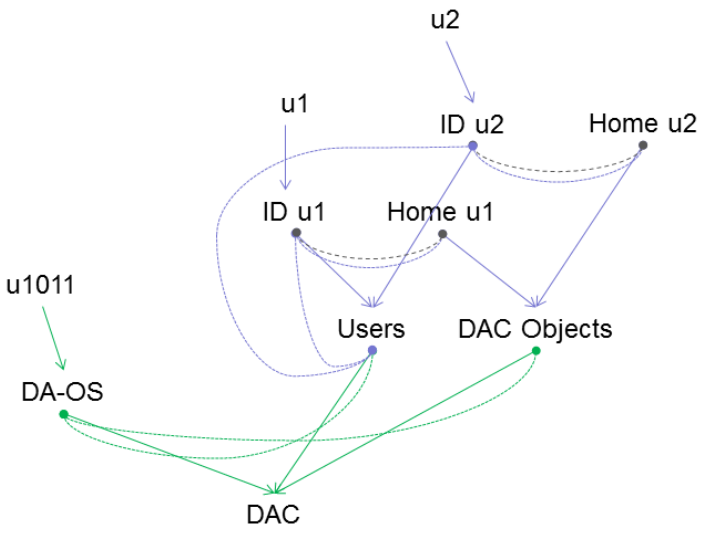

Figure 20: DAC Segment of the Integrated System Policy

Figure 21 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.

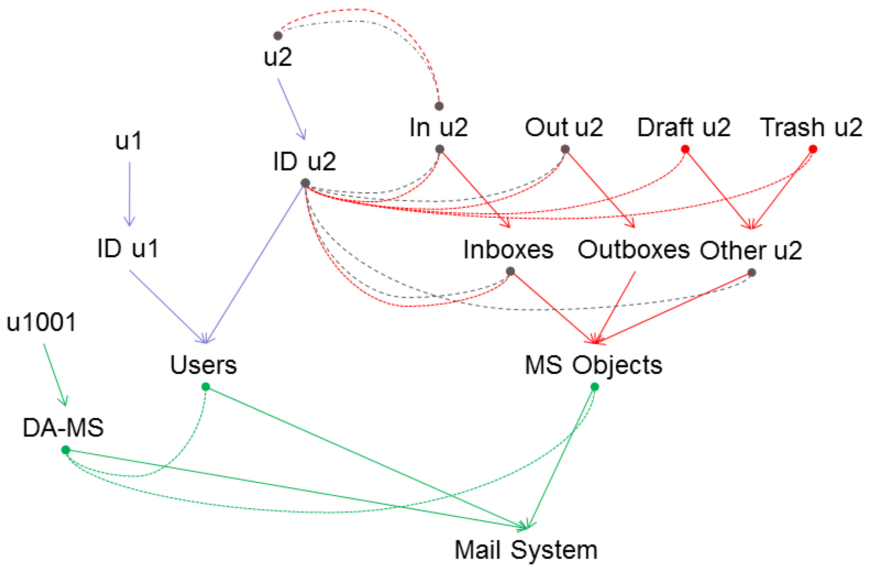

Figure 21: Mail System Segment of the Integrated System Policy

An example of an entire authorization graph for the integrated system policy is shown in Figure 22. The DAC operating system objects and relations for user u2, which were shown in a Figure 20, are omitted to avoid an overly busy illustration.

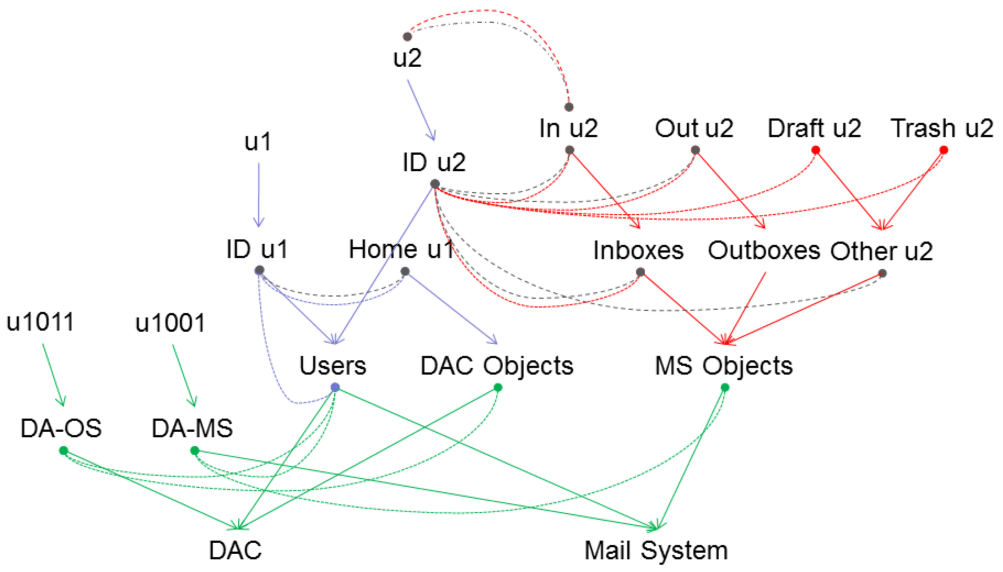

Figure 22: Authorization Graph of the Integrated System Policy

Several inferences can be drawn from this example. The first is that policies grow very quickly and can become unwieldy to view in their entirety. The second is that relationships that involve administrative access rights exceed those that involve non-administrative ones, relative 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 and compulsory; 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 23. 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. Security levels are represented by attributes within the PM policy. 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.

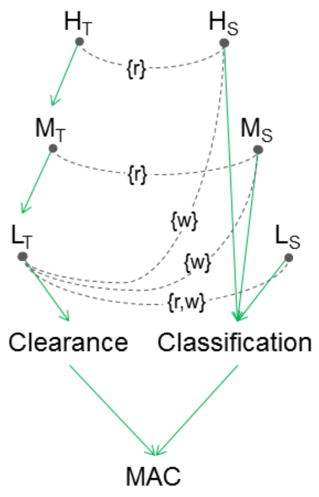

Figure 23: Authorization Graph for the MAC Policy Segment

The aforementioned 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 discussed in Section 2.3 is reflected in the policy.

Write operations are treated differently than read operations. 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 = read ∧ EC.o ASSIGN* $H_S$ do
        CreateOblig-ConjPProhib (EC.c, EC.p, {w}, ∅, {$H_S$})[11]

When EC.op = read ∧ EC.o ASSIGN* $M_S$ do
        CreateOblig-ConjPProhib (EC.c, 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 = read ∧ EC.o ASSIGN* $H_S$ do
        CreateOblig-DisjPProhib (EC.c, EC.p, {w}, {$L_S$, $M_S$}, ∅)

When EC.op = read ∧ EC.o ASSIGN* $M_S$ do
        CreateOblig-DisjPProhib (EC.c, 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 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 24. 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.

[11] The semantics of the administrative routine, CreateOblig-DisjPProhib, used in this obligation is essentially the same as that for the routine C-DisjPProhib given in Appendix D, with the exception that the user that defined the obligation (vis., EC.c) must hold sufficient authorization to perform the body of the routine.

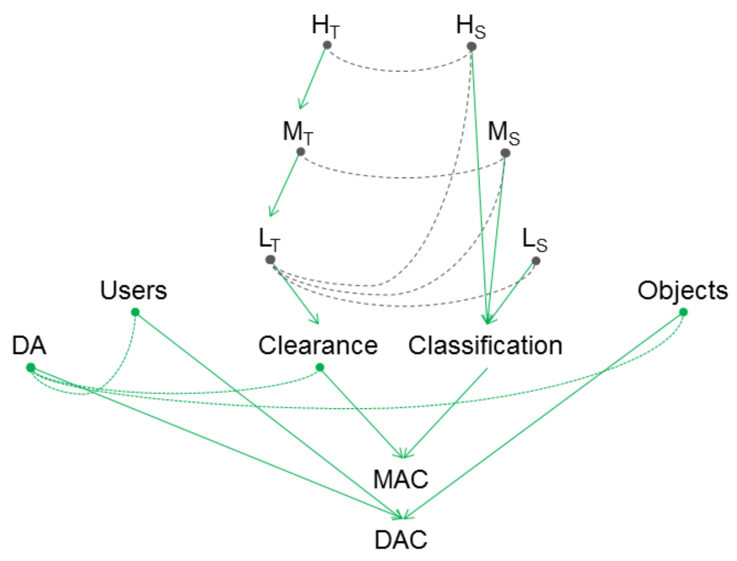

Figure 24: Authorization Graph for MAC-DAC system

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 25 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 require that the create object in object attribute routine, C-OinOA, makes a second assignment to the classification level attribute. The second way is more indirect. The second attribute assignment would be carried out through an administrative obligation that makes the assignment when triggered by the creation of an object in an object attribute event (i.e., a create-OinOA event).

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

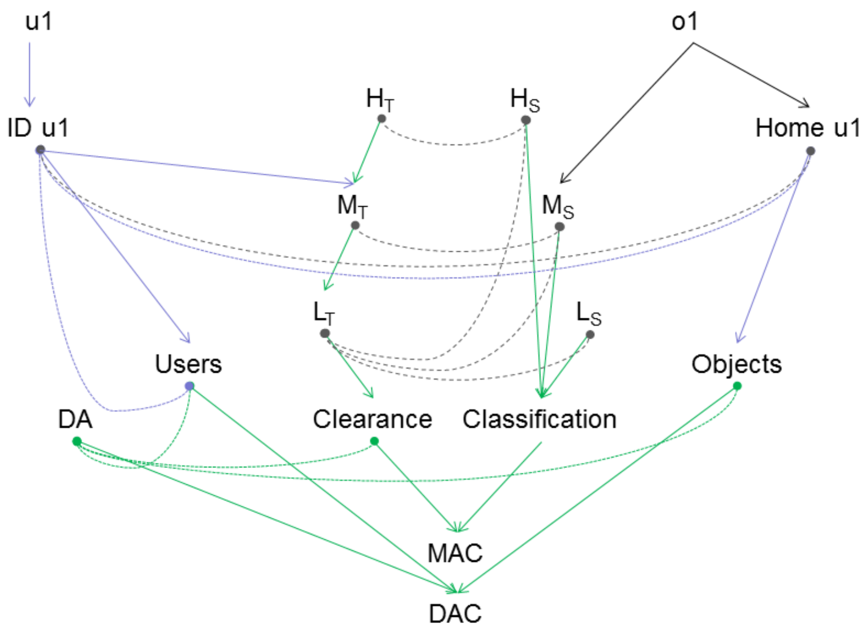

Figure 25: DAC-MAC Authorization Graph with Populated User

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 obligations is needed for this approach:

> **When** EC.op = create-OinOA ∧ EC.u ASSIGN$^+$ L_T ∧ ¬(EC.u ASSIGN$^+$ M_T) **do**
> CreateOblig-OtoOA (EC.c, EC.argseq (1), L_S)[12]
>
> **When** EC.op = create-OinOA ∧ EC.u ASSIGN$^+$ M_T ∧ ¬(EC.u ASSIGN$^+$ H_T) **do**
> CreateOblig-OtoOA (EC.c, EC.argseq (1), M_S)
>
> **When** EC.op = create-OinOA ∧ EC.u ASSIGN$^+$ H_T **do**
> CreateOblig-OtoOA (EC.c, EC.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 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's involvement is

[12] The EC.argseq of an event context for an access request involving the create-OinOA administrative operation is ⟨o, oa⟩, where EC.argseq (1) contains o, the identifier of the object that was created, and EC.argseq (2) contains oa, the identifier of the object attribute to which the object was assigned.

needed to specify part of the MAC policy. Nonetheless, 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 user, 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. But, 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 via an argument supplied by the user. If the user has read an object at a higher classification level, existing prohibitions prevent writing to the object, as would be warranted. With the second approach, 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 and the rest are dispersed across other systems, are also possible. A separate decision to use either a programming interface or network protocol for an interface may be made, as appropriate, because of the functionally equivalency between the two.

7.1 Architectural Components

The components of the PM functional architecture 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 26 illustrates these components and their interfaces. Note that in this diagram the arrows represent calls or messages between components and the bars represent component interfaces. Typically, the calling/sending component waits for a response from the called/receiving component, before continuing its operation. For simplicity, the response is not shown in the diagram.

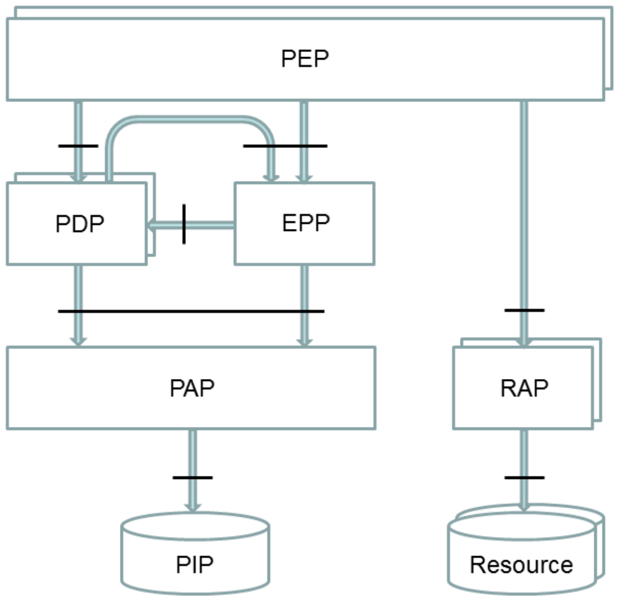

Figure 26: Architectural Components of the PM

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 the interface that it provides, which is typically a programming interface. More than one PEP may exist to service applications. The PEP ensures that access requests are verified as being in conformance with the specified policy, before the accesses in question can proceed. To accomplish this objective, the PEP works in tandem with a PDP and must not be bypassable.

  The PEP conveys access requests issued by a PM-aware application to a PDP for decisions about their disposition. An application's access request includes the identity of the requesting process, the requested operation, and the arguments of the operation, including the targeted resource(s) and optional data. Both non-administrative and administrative access requests are handled by the PEP.

  For non-administrative resource access 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 context of the event to the EPP.

  Decisions on access requests involving administrative operations 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 access decision function defined in the PM model. It also carries out all access requests that involve administrative operations for which a grant decision has been rendered. Multiple PDPs may exist in the PM environment.

  The PDP obtains the information needed to verify the access request from the PIP, via the PAP. If an access is granted which involves a resource operation on a physical resource, the PDP supplies the necessary details to the requesting PEP for locating and accessing the resource. If denied, that decision is conveyed back to the PEP in the response. If an access is granted, which involves an administrative operation on the abstract data structures of the PM, 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, which is conveyed to the EPP for eventual processing.

  The PDP also works with the EPP to perform reference mediation for the event response of any obligations that the EPP matches to an event it receives. Unlike access requests, event responses can involve multiple administrative actions that must be mediated based on the authorization held by the user that created the obligation, not the authorization held by process that triggered the obligation. The actions must be carried out in their entirety as an atomic action by the PDP via the PAP. The result is that either the response is carried out completely, or no part of the response is carried out. Note that unlike an administrative access request, the PDP does not generate events for the EPP to process for the actions carried out in an obligation response. That does not, however, mean that such accesses go unaudited.

- **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. Conceptually, the PIP embodies a reification of the administrative commands specified in Appendix C. 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 and efficiently.

  The implementation strategy and efficiency tradeoffs for derived relations (e.g., privileges) are an important performance consideration. Derived relations act as a single relation, even though they rely on information from one or more other base relations. Data structures representing derived relations may be virtual and computed as needed. However, continual reevaluation of the relation that can affect the performance of the PIP. Derived relations may also be materialized such that the tuples resulting from evaluating the relation over the current instances of the base relations are actually maintained continuously.

- **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 component 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 content within the PIP (i.e., the policy configuration), as described by the administrative routines in Appendix D, and ensures that access to the PIP 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 pattern that is matched with an event, the EPP uses a PDP to make an access request decision on the associated event response (i.e., the sequence of administrative actions defined for each obligation), and to carry out the response if the definer of the obligation holds sufficient authorization. The EPP can be viewed as a transaction processing monitor, whose performance is crucial to the overall effectiveness of the architecture. To avoid contention with active PEPs accessing the same PDP as the EPP, a specific PDP 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, decisions on event-driven administrative actions may incur delay before they are taken automatically, which in turn can affect the validity of request decisions taken concurrently with the delay. 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 27, 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.

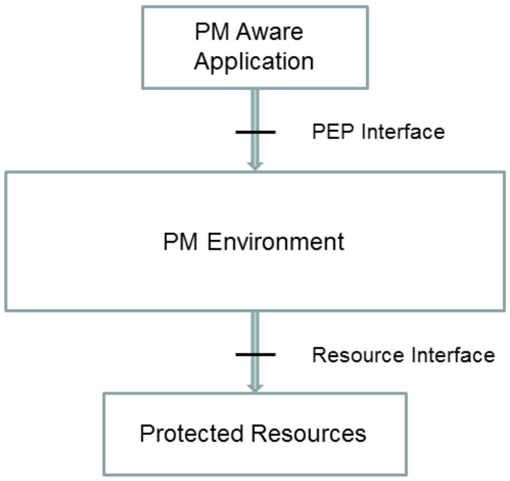

Figure 27: Application's Perspective of the PM Environment

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 at the PIP within the PM environment.

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 behave as intended and cannot be bypassed. Potential adversaries may involve more than one legitimate user working in collaboration to defeat access controls to PM protected resources, as well as involvement of non-legitimate external parties. Adversaries may have access to the data paths between components and be able to eavesdrop on exchanges.

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

Distributed entities rely on networks communications to interoperate. Messages transmitted between PM entities are potentially susceptible to attack by malicious third parties, and therefore, protocols that convey messages require adequate safeguards. Secure transmission protocols use cryptography to convey content between authenticated parties. As with authentication protocols, transmission protocol implementations may contain errors as well as design flaws, which allow 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 and EPP, can be particularly attractive targets for denial of service attacks, since they represent choke points in the access control framework.

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. Related Work

The notion of a multi-policy machine capable of enforcing multiple, possibly contradictory security policies expressed in terms of user and object attributes was envisioned in the early 90's [Hos93]. As with the PM, domains could be mutually exclusive or overlap each other, allowing subjects and objects to belong to more than one domain and fall under more than one policy. A conceptual framework was developed for the multi-policy machine, which was represented mathematically in terms of machine states, access requests, decisions, and state-transitions [Bel94]. The framework was used mainly to address strategies for resolving several key issues with multi-policies and policy conflicts that could arise. Accommodation of multiple policies is an inherent part of the PM framework, addressed through the use of policy class-based domains, the ability to derive privileges and restrictions from associations and prohibitions that span multiple policy classes, and the treatment of access requests by the access decision function.

The concept of trust management was introduced with the PolicyMaker system [Bla96]. Trust management presents a comprehensive approach to specifying and interpreting security policies, credentials, and relationships to allow access decisions to be made about security-critical actions. To provide a coherent framework for expressing interrelationships between security policies, security credentials, and trust relationships, PolicyMaker and its successor KeyNote utilize public key infrastructure environments, certificate-based trust, and binding of cryptographic keys to actions [Bla98, Bla99]. Five basic components can be identified: a mechanism for identifying principals, a language for describing security-critical actions, a language for specifying application policies, a language for specifying delegation credentials, and a compliance checker that factors all of the above in its decisions. Unlike other access control approaches focused mainly on centralized operating environments, trust management is particularly suited for situations where security policy is decentralized and distributed across a network, such as multi-system applications and data services that cross departmental and organizational boundaries [Bla99]. In this regard, it is similar to the PM framework.

A family of usage control (UCON) models was devised as a conceptual framework for access control [Par04]. The UCON models offer a broad and diverse scope of coverage that encompasses digital rights management and other modern access control models. The complete family of models is collectively termed UCON_{ABC}, since to provide a richer and finer decision capability, decision-related factors are delineated into three categories: Authorizations (A), oBligations (B), and Conditions (C). Authorizations concern the evaluation of authorization rules and usage rights together with subject and object attributes in reaching a usage decision. Attributes are used to represent key characteristics of subjects and objects, such as identities, security labels, properties, and capabilities. Obligations are treated as requirements that subjects have to perform before or during (or even after, in case of a global obligation) obtaining or exercising usage rights. Conditions are environmental or system requirements that are independent from individual subjects and objects, such as time of day and location. In comparison, the PM covers authorizations and obligations, but not conditions, and digital rights management is outside its purview.

The UCON$_{ABC}$ decision-related factors may effect subject and object attributes either before or after a usage decision, continually during and after a usage decision, or not at all. Tenable cases narrow the possible alternatives down to sixteen basic models that comprise the UCON$_{ABC}$ family. The basic models are designed to be combined together to capture subtle distinctions that arise in real-world policies, giving rise to a large number of more elaborate, composite models. Despite the broad policy coverage of the models, however, the administrative aspects of UCON$_{ABC}$ are outside its scope. Instead, usage rules for authorizations, obligations, and conditions and other policy constituents are presumed to be in place. The result is a framework that is rather complex, but can serve well as a taxonomy for classifying access control models, for which the sixteen basic models define the taxa. In contrast, the scope of PM model is more focused and covers not only wide segments of what might be termed UCON$_{AB}$, in which relevance to modern access control models has been demonstrated, but also all administrative aspects for configuring and maintaining policy.

The Extensible Access Control Markup Language (XACML) is an XML-based language standard designed to express security policies, as well as the access requests and responses needed for querying the policy system and reaching an access decision [XAC13]. XACML is similar to the PM insofar as it provides a flexible, mechanism-independent representation of policy rules that may vary in granularity, and it employs attributes in the definition of policy. Both XACML and the PM have the ability to combine policies of different authoritative domains into a single policy set that is used to make access control decisions in a widely distributed system environment. However, while the PM requires only a single policy evaluation algorithm for its policies, an XACML policy may have conflicting policies or rules, which must be resolved during policy evaluation by selectively applying one of several combining algorithms defined in the standard. Moreover, XACML does not allow the specification and enforcement of policies that pertain to processes in isolation to their users, as does the PM, thereby excluding support for a wide variety of policies in which this capability is needed [Fer11].

Because XACML is implemented in XML, it inherits XML's benefits and drawbacks. The flexibility and expressiveness of XACML while powerful, make the specification of policy complex and verbose [Hu07]. Applying XACML in a heterogeneous environment requires fully specified data type and function definitions that produce a lengthy textual document, even if the actual policy rules are trivial. Unlike XACML, the PM is not a policy specification language, which avoids the syntactic and semantic complexity in defining an abstract language for expressing platform-independent policies [Hu07]. PM policies are specified in terms of model elements that are maintained at a centralized point and typically rendered and manipulated graphically. For example, to describe hierarchical relations between attributes, the PM requires only the addition of links representing assignment relations between them.

Other areas of divergence between the two approaches are also evident. Until recently, XACML policy administration treated the access control of the administration of policies outside of the policy model. To control editing of the policy, OE mechanisms such as operating system access controls had to be used, which for large distributed systems could prove difficult to manage. This was never the case with the PM. Also, in the XACML delegation model for decentralized administration of access policies, administrative control policies that specify delegation rights are separate from other access rights, while the PM treats delegation rights and access rights uniformly.

9. References

[ANSI04] Information technology – Role-Based Access Control (RBAC), INCITS 359-2004, American National Standard for Information Technology, American National Standards Institute, 2004.

[ANSI13] Information technology - Next Generation Access Control - Functional Architecture (NGAC-FA), INCITS 499-2013, American National Standard for Information Technology, American National Standards Institute, March 2013.

[ANSI15] Working DRAFT Information technology - Next Generation Access Control – Generic Operations and Data Structures (NGAC-GOADS)), INCITS 526-201x, American National Standard for Information Technology, American National Standards Institute, June 2015.

[Bel94] D.E. Bell, "Modeling the Multipolicy Machine," *Proceedings of the 1994 Workshop on New Security Paradigms*, Little Compton, Rhode Island, USA, August 3-5, 1994, pp. 2-9. http://dx.doi.org/10.1109/NSPW.1994.656208.

[Bla96] M. Blaze, J. Feigenbaum, and J. Lacy, "Decentralized Trust Management," *IEEE Symposium on Security and Privacy*, Oakland, California, USA, May 6-8, 1996, pp. 164-173. http://dx.doi.org/10.1109/SECPRI.1996.502679.

[Bla98] M. Blaze, J. Feigenbaum, and A. Keromytis, "KeyNote: Trust Management for Public-Key Infrastructures," *The 6th International Workshop on Security Protocols*, Cambridge, United Kingdom, April 15-17 1998. In *Lecture Notes in Computer Science* 1550, Berlin: Springer, pp. 59-63. http://dx.doi.org/10.1007/3-540-49135-X_9.

[Bla99] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis, "The KeyNote Trust Management System Version 2," Internet Engineering Task Force (IETF) Request for Comments (RFC) 2704, September 1999. http://www.ietf.org/rfc/rfc2704 [accessed 9/16/2015].

[Bre89] D.F.C. Brewer and M.J. Nash, "The Chinese Wall Security Policy," *1989 IEEE Symposium on Security and Privacy*, Oakland, California, USA, May 1-3, 1989, pp. 206-214. http://dx.doi.org/10.1109/SECPRI.1989.36295.

[Fer01] D.F. Ferraiolo, R.S. Sandhu, S.I. Gavrila, D. R. Kuhn, and R. Chandramouli, "Proposed NIST standard for role based access control," *ACM Transactions on Information and System Security (TISSEC)*, vol. 4, no. 3 (August 2001), pp. 224-274. http://dl.acm.org/citation.cfm?id=501980.

[Fer05] D.F. Ferraiolo, S.I. Gavrila, V.C. Hu, and D.R. Kuhn, “Composing and Combining Policies Under the Policy Machine,” *Tenth ACM Symposium on Access Control Models and Technologies (SACMAT ‘05)*, Stockholm, Sweden, 2005, pp. 11-20. http://dx.doi.org/10.1145/1063979.1063982. http://csrc.nist.gov/staff/Kuhn/sacmat05.pdf [accessed 9/16/2015].

[Fer11] D.F. Ferraiolo, V. Atluria, and S.I. Gavrila, “The Policy Machine: A Novel Architecture and Framework for Access Control Policy Specification and Enforcement,” *Journal of Systems Architecture*, vol. 57, no. 4 (April 2011), pp. 412-424. http://dx.doi.org/10.1016/j.sysarc.2010.04.005.

[Gra72] G.S. Graham and P.J. Denning, “Protection: Principles and Practice,” *Spring Joint Computer Conference (AFIPS ’72 (Spring))*, Atlantic City, New Jersey, USA, May 16-18, 1972, pp. 417-429. http://dx.doi.org/10.1145/1478873.1478928.

[Har76] M.A. Harrison, W.L. Ruzzo, and J.D. Ullman, “Protection in Operating Systems,” *Communications of the ACM*, vol. 19, no. 8 (August 1976), pp. 461-471. http://dx.doi.org/10.1145/360303.360333.

[Hos93] H.H. Hosmer, “The Multipolicy Paradigm for Trusted Systems,” *Proceedings of the 1992-1993 Workshop on New Security Paradigms (NSPW ’92-’93)*, Little Compton, Rhode Island, USA, August 1993, pp. 19-32. http://dx.doi.org/10.1145/283751.283768.

[Hu06] V.C. Hu, D.F. Ferraiolo, and D.R. Kuhn, “Assessment of Access Control Systems,” NISTIR 7316, National Institute of Standards and Technology, Gaithersburg, Maryland, USA, September 2006, 60 pp. http://www.nist.gov/manuscript-publication-search.cfm?pub_id=50886 [accessed 9/16/2015].

[Hu07] V.C. Hu, D.F.Ferraiolo, and K. Scarfone, “Access Control Policy Combinations for the Grid Using the Policy Machine,” *Seventh IEEE International Symposium on Cluster Computing and the Grid*, Rio de Janeiro, Brazil, May 14-17, 2007, pp. 225-232. http://dx.doi.org/10.1109/CCGRID.2007.15.

[ISO02] ISO/IEC 13568:2002, “Information technology – Z formal specification notation – Syntax, type system and semantics,” International Organization for Standardization (ISO)/International Electrotechnical Commission (IEC), First Edition, July 1, 2002. http://standards.iso.org/ittf/PubliclyAvailableStandards/c021573_ISO_IEC_13568_2002(E).zip [accessed 9/16/2015].

[Kuh98] D.R. Kuhn, “Role Based Access Control on MLS Systems Without Kernel Changes,” *Third ACM Workshop on Role Based Access Control (RBAC ’98)*, Fairfax, Virginia, USA, October 22-23, 1998, pp. 25-32. http://dx.doi.org/10.1145/286884.286890. http://www.nist.gov/manuscript-publication-search.cfm?pub_id=916540 [accessed 9/16/2015].

[Lam71] B. Lampson, "Protection," *Proceedings of the 5th Princeton Conference on Information Sciences and Systems*, Princeton, New Jersey, USA, March 1971. Reprinted in *ACM Operating Systems Review*, vol. 8, no. 1 (January 1974), pp. 18-24. http://dx.doi.org/10.1145/775265.775268.

[Li05] N. Li and M.V. Tripunitara, "On Safety in Discretionary Access Control," *2005 IEEE Symposium on Security and Privacy*, Oakland, California, USA, May 8-11, 2005, pp. 96-109. http://dx.doi.org/10.1109/SP.2005.14.

[NCSC87] *A Guide to Understanding Discretionary Access Control in Trusted Systems*, NCSC-TG-003, Version-1, National Computer Security Center, Fort George G. Meade, Maryland, USA, September 30, 1987, 29 pp. http://fas.org/irp/nsa/rainbow/tg003.htm [accessed 9/16/2015].

[Osb00] S. Osborn, R. Sandhu, and Q. Munawer, "Configuring Role-Based Access Control to Enforce Mandatory and Discretionary Access Control Policies," *ACM Transactions on Information and System Security (TISSEC)*, vol. 3, no. 2 (May 2000), pp. 85-106. http://dx.doi.org/10.1145/354876.354878.

[Par04] J. Park and R.S. Sandhu, "The UCON$_{ABC}$ Usage Control Model," *ACM Transactions on Information and System Security (TISSEC)*, vol. 7, no. 1 (February 2004), pp. 128-174. http://dx.doi.org/10.1145/984334.984339.

[Per96] R.V. Peri, *Specification and Verification of Security Policies*, Ph.D. Dissertation, University of Virginia, Faculty of the School of Engineering and Applied Science, January 1996, 179 pp. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.28.4368&rep=rep1&type=pdf [accessed 9/16/2015].

[Sal75] J.H. Saltzer and M.D. Schroeder, "The Protection of Information in Computer Systems," *Proceedings of the IEEE*, vol. 63, no. 9 (September 1975), pp. 1278-1308. http://dx.doi.org/10.1109/PROC.1975.9939.

[San92] R.S. Sandhu, "Lattice-Based Enforcement of Chinese Walls," *Computers and Security*, vol. 11, no. 8 (December 1992), pp. 753-763. http://dx.doi.org/10.1016/0167-4048(92)90131-A. http://profsandhu.com/journals/csec/csec92-cwall-org.pdf [accessed 9/16/2015].

[San94] R.S. Sandhu and P. Samarati, "Access Control: Principles and Practice," *IEEE Communications Magazine*, vol. 32, no. 9 (September 1994), pp. 40-48. http://dx.doi.org/10.1109/35.312842.

[Sha13] A. Sharifi, M.V. Tripunitara, "Least-Restrictive Enforcement of the Chinese Wall Security Policy," *18th ACM Symposium on Access Control Models and Technologies (SACMAT '13)*, Amsterdam, The Netherlands, June 12-14, 2013, pp. 61-72. http://dx.doi.org/10.1145/2462410.2462425.

[Tri04] M.V. Tripunitara and N. Li, “Comparing the Expressive Power of Access Control Models,” *11th ACM Conference on Computer and Communications Security (CCS ’04)*, Washington, DC, USA, October 25-29, 2004, pp. 62-71. http://dx.doi.org/10.1145/1030083.1030093.

[Tri06] M.V. Tripunitara and N. Li, *The Foundational Work of Harrison-Ruzzo-Ullman Revisited*, CERIAS Tech Report 2006-33, Center for Education and Research in Information Assurance and Security, Purdue University, West Lafayette, Indiana, USA, 2006, 17 pp. https://www.cerias.purdue.edu/assets/pdf/bibtex_archive/2006-33.pdf [accessed 9/16/2015].

[XAC13] “eXtensible Access Control Markup Language (XACML), Version 3.0,” OASIS Standard, January 22, 2013. http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.pdf [accessed 9/16/2015].

Appendix A—Acronyms

ACL	Access Control List
ANSI	American National Standards Institute
API	Application Programming Interface
DA	Domain Administrator
DAC	Discretionary Access Control
DNS	Domain Name System
DSD	Dynamic Separation of Duty
EPP	Event Processing Point
INCITS	International Committee for Information Technology Standards
MAC	Mandatory Access Control
MS	Mail System
NGAC	Next Generation Access Control
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
RBAC	Role Based Access Control
SA	Subdomain Administrator
S^2A	Sub-SA
S^3A	Sub-S^2A
SoD	Separation of Duty
SSD	Static Separation of Duty
UCON	Usage Control
URI	Uniform Resource Identifier
XACML	eXtensible Access Control Markup Language
XML	eXtensible Markup Language

Appendix B—Notation

The mathematical notation used in this report corresponds to a subset of the Z formal specification notation defined in ISO/IEC 13568:2002 [ISO02]. Below is a summary of the notation used and, for any differences that exist, their counterpart in the Z notation.

Numbers

$\mathbb{Z}$	Set of integers
$\mathbb{N}$	Set of natural numbers { 0, 1, 2, ... }
$\mathbb{N}_1$	Set of strictly positive numbers { 1, 2, ... }
$m + n$	Addition
$m - n$	Subtraction
$m * n$	Multiplication
$m \text{ div } n$	Division
$m \text{ mod } n$	Remainder (modulus)
$m \leq n$	Less than or equal
$m < n$	Less than
$m \geq n$	Greater than or equal
$m > n$	Greater than
$m \ldots n$	Number range

Logic

$\neg p$	Logical negation, not
$p \wedge q$	Logical conjunction, and
$p \vee q$	Logical disjunction, or
$p \Rightarrow q$	Logical implication
$p \Leftrightarrow q$	Logical equivalence
$\forall x$	Universal quantification
$\exists x$	Existential quantification
$\exists_1 x$	Uniqueness quantification
$\nexists x$	Existential quantification negation (in Z, $\neg\exists$)

Sets and expressions

$x = y$	Equality
$x \neq y$	Inequality
$x \in A$	Set membership
$x \notin A$	Non-membership
$\emptyset$	Empty set
$A \subseteq B$	Set inclusion
$A \subset B$	Strict set inclusion
$\{x_1, \ldots, x_n\}$	Set display
$A \times B \ldots$	Cartesian product
2^A	Power set (in Z, $\mathbb{P}A$)
2^A_1	Non-empty subsets (in Z, $\mathbb{P}_1A$)
$A \cap B$	Set intersection
$A \cup B$	Set union

$A \setminus B$	Set difference
first x	First element of an ordered pair
second x	Second element of an ordered pair
#A	Number of elements in a set

Relations

$R \subseteq A \times B$	Binary relation (in Z, $R: A \leftrightarrow B$)
$R \subseteq A \times B \times C$	Ternary relation (in Z, $R: A \leftrightarrow B \leftrightarrow C$)
dom R	Domain of a relation
ran R	Range of a relation
$A \lhd R$	Domain Subtraction
$R \rhd A$	Range Subtraction
(a, b)	Ordered pair of a binary relation
(a, b, ... , c)	Ordered tuple of an n-ary relation
a R b	Relation holds between a and b – infix
R(a, b)	Relation holds between a and b – prefix set membership
$(a, b) \in R$	Relation holds between a and b – set membership
R^+	Transitive closure
R^*	Reflexive transitive closure

Functions

F(x)	Function application
$F \subseteq A \times B$	Partial functions (in Z, $A \rightarrow B$)
$F \subseteq A \times B$	Total functions (in Z, $A \nrightarrow B$)

Sequences

$\langle\rangle$	Empty sequence
seq A	Set of finite sequences
seq_1 A	Set of non-empty finite sequences
iseq A	Set of finite injective sequences
$iseq_1$ A	Set of non-empty finite injective sequences (in Z, iseq $A \setminus \{\langle\rangle\}$)
head s	First element of a sequence
last s	Last element of a sequence
s (i)	i^{th} elment of a sequence
$\langle s_1, \ldots, s_n \rangle$	Sequence display

The mathematical notation shown above is used to define the policy elements, entities, and relations that comprise the PM model. A summary of the core relations of the model and the elements and entities used in those relations appear in Table 3 given below. The relations are listed across the top of the table, and the elements and entities are listed along the left-hand side of the table. Note that the power set or finite sequence of the elements and entities listed may actually be used in a cross-referenced relation, and that for some relations, namely prohibitions, multiple types and variants exist (i.e., process, user and user attribute types, and conjunctive, and disjunctive varients).

Table 3: Summary of Core Model Abstractions

	Process_User	ASSIGN	ASSOCIATION	PROHIBITION[13]	OBLIG	AREQ
P	X			X		X
U	X	X		X	X	
UA		X	X	X		
O		X				X
OA		X	X	X		
PC		X				
AR			X	X		
OP						
Σ_P					X	
Σ_R					X	
Arg						X

[13] The heading PROHIBITIONS respresents the following PM relations: P_DENY_CONJ, U_DENY_CONJ, UA_DENY_CONJ, P_DENY_DISJ, U_DENY_DISJ, and UA_DENY_DISJ.

Appendix C—Administrative Commands

The PM access control model is essentially a finite state machine. The policy elements and the relations that define the access rights between these entities collectively constitute the authorization state of the policy maintained by the PM. Administrative commands define the behavior of state transitions that occur when PM policy elements and relations are created, deleted, and updated.

As noted in Chapter 4, administrative routines rely on administrative commands to define and alter policy. Changes in state, or state transitions, occur when an access request involving an administrative operation is granted by the access decision function, and the associated administrative routine is successfully carried out through one or more administrative commands. Changes in state may also occur when an obligation is triggered and its response (i.e., one or more administrative routines) is successfully carried out. Administrative commands constitute the range of behaviors that can be taken against the policy elements and relationships defined by the PM model. This appendix defines the semantics of the core administrative commands of the PM.

C.1 Semantic Definitions

The semantic description of an administrative command differs from a syntactic description or programming language representation. The semantic descriptions define the correct behavior expected of administrative commands, which is essential for maintaining the integrity of the PM as it transitions between states. The specifications should not be interpreted as programming statements, and instead should be interpreted as changes to model structures representing the authorization state, which occur when a command is correctly carried out. Behavioral aspects other than those related to access control are outside the scope of these descriptions, including other facets of security such as authentication and audit.

Preconditions are defined for each administrative command. Preconditions denote requirements. They are expressed as a logical expression that must be satisfied for the command to be carried out. The preconditions for administrative commands ensure that the arguments supplied for the formal parameters of the command are of the correct type, and that the basic properties of the model are observed.

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

- The behavior of administrative commands is atomic; their effects are indivisible and uninterruptable.
- The main body of an administrative command specifies state changes for those model elements and relations that are affected by its conduct—the state of any unspecified element or relation is unaffected and remains unchanged.
- Model elements and relations, whose states are affected by the behavior 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 change to the authorization state occurs 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 command provide either an input or output to the command, but never provide both an input and an output.

In the specification of preconditions, simplifications were made with the notation used; namely, the tuples of a relation are treated as members of a set when the predicate calculus qualifiers ∃, ∄, and ∀ are applied. For example, a triple of the relation ASSOCIATION has three elements: a user attribute, a set of access rights, and an object attribute. To specify that a triple (a, b, c) of ASSOCIATION with the property a=x does not exist would normally be done as follows:

$$\forall a \in UA, \forall b \in ARs, \forall c \in OA: \neg((a, b, c) \in ASSOCIATION \land a = x)$$

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

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

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., ∄ in the above formula replaced by ∃ or ∀ respectively) also exist.

Some foundational data elements and relations are needed to specify the behavior of administrative commands. All commands depend on a basic data type called ID, which represents opaque identifiers of the PM framework. An identifier is a finite sequence of bytes, whose characteristics and interpretation are left unspecified. The set GUID is used to maintain the set of all identifiers allocated to entities that comprise policy and to ensure that each identifier is assigned to only one entity. These items are defined formally as follows:

$$ID = seq_1 \{00000000b, \ldots, 11111111b\}$$
$$GUID \subseteq ID$$

Several other data elements and relations are also needed to maintain identifiers allocated to various types of entities used in policy formation, namely sets of access rights, sets of event patterns and responses, and sets of exclusive and inclusive attributes, and to maintain mappings from each identifier to its respective members. They are defined as follows:

$$ARset \subseteq GUID$$
$$ARmap \subseteq ARset \times 2_1^{AR}$$

$PATTERNid \subseteq GUID$
$PATTERNmap \subseteq PATTERNid \times seq_1\ \Sigma_P$

$RESPONSEid \subseteq GUID$
$RESPONSEmap \subseteq RESPONSEid \times seq_1\ \Sigma_R$

$ATIset \subseteq GUID$
$ATImap \subseteq ATIset \times 2^{AT}$

$ATEset \subseteq GUID$
$ATEmap \subseteq ATEset \times 2^{AT}$

An initial state (i.e., the state immediately after initialization of the PM framework) is defined in terms of the abstract data elements and relations and predicates that stipulate the initial conditions of the system. The PM authorization state is initialized to zero or empty as described in the administrative command below. The authorization state can then be populated by the security authority with supported elements, such as inherent administrative operations and associated access rights, and tailored to the specifics of the operating environment with other elements, such as resource operations and access rights.

```
InitialState ()
    {
        // initialize policy elements
        $U' = \emptyset$
        $O' = \emptyset$
        $UA' = \emptyset$
        $OA' = \emptyset$
        $PC' = \emptyset$
        $P' = \emptyset$
        $AT' = \emptyset$
        $PE' = \emptyset$
        // initialize relations
        $ASSIGN' = \emptyset$
        $ASSOCIATION' = \emptyset$
        $U\_DENY\_CONJ' = \emptyset$
        $U\_DENY\_DISJ' = \emptyset$
        $P\_DENY\_CONJ' = \emptyset$
        $P\_DENY\_DISJ' = \emptyset$
        $UA\_DENY\_CONJ' = \emptyset$
        $UA\_DENY\_DISJ' = \emptyset$
        $OBLIG' = \emptyset$
        // initialize other required entities
        $GUID' = \emptyset$
        $Process\_User' = \emptyset$
        $AOP' = \emptyset$
        ROP′ = ∅
        OP′ = ∅
        AR′ = ∅
        ARset′ = ∅
        ARmap′ = ∅
        ATIset′ = ∅
        ATImap′ = ∅
        ATEset′ = ∅
        ATEmap′ = ∅
        PATTERN′ = ∅
        PATTERNid′ = ∅
        PATTERNmap′ = ∅
        RESPONSE′ = ∅
        RESPONSEid′ = ∅
        RESPONSEmap′ = ∅
    }
```

C.2 Element Creation Commands

The semantic descriptions of element creation commands describe the changes in the authorization state that occur with the addition of new policy elements to the policy representation. The commands are formulated to preserve model properties. For example, a user can be created only with respect to an existing user attribute, and not independently of any user attribute. The preconditions for these commands ensure that the arguments supplied for the formal parameters of a command are valid.

CreateUinUA (x: ID, y: ID)

```
// add user x to the policy representation and assign it to user attribute y
    x ∉ U
    x ∉ GUID
    y ∈ UA
    y ∈ GUID
    {
        GUID′ = GUID ∪ {x}
        U′ = U ∪ {x}
        PE′ = PE ∪ {x}
        ASSIGN′ = ASSIGN ∪ {(x, y)}
    }
```

CreateUAinUA (x: ID, y: ID)

```
// add user attribute x and assign it to user attribute y
    x ∉ UA
    x ∉ GUID
    y ∈ UA
    y ∈ GUID
    {
        GUID′ = GUID ∪ {x}
        $UA' = UA \cup \{x\}$
        $AT' = AT \cup \{x\}$
        $PE' = PE \cup \{x\}$
        $ASSIGN' = ASSIGN \cup \{(x, y)\}$
    }
```

CreateUAinPC (x: ID, y: ID)

```
// add user attribute x and assign it to policy class y
    $x \notin UA$
    $x \notin GUID$
    $y \in PC$
    $y \in GUID$
    {
        $GUID' = GUID \cup \{x\}$
        $UA' = UA \cup \{x\}$
        $AT' = AT \cup \{x\}$
        $PE' = PE \cup \{x\}$
        $ASSIGN' = ASSIGN \cup \{(x, y)\}$
    }
```

CreateOinOA (x: ID, y: ID)

```
// add object x and assign it to object attribute y
    $x \notin GUID$
    $x \notin O$
    $x \notin OA$
    $y \in OA$
    $y \in GUID$
    {
        $GUID' = GUID \cup \{x\}$
        $O' = O \cup \{x\}$
        $OA' = OA \cup \{x\}$
        $AT' = AT \cup \{x\}$
        $PE' = PE \cup \{x\}$
        $ASSIGN' = ASSIGN \cup \{(x, y)\}$
    }
```

CreateOAinOA (x: ID, y: ID)

```
// add object attribute x and assign it to object attribute y
    $x \notin OA$
    $x \notin GUID$
    $y \in OA$
    $y \notin O$
    $y \in GUID$
    {
        $GUID' = GUID \cup \{x\}$
        $OA' = OA \cup \{x\}$
        $AT' = AT \cup \{x\}$
        $PE' = PE \cup \{x\}$
        $ASSIGN' = ASSIGN \cup \{(x, y)\}$
    }
```

CreateOAinPC (x: ID, y: ID)

```
// add object attribute x and assign it to policy class y
    $x \notin OA$
    $x \notin GUID$
    $y \in PC$
    $y \in GUID$
    {
        $GUID' = GUID \cup \{x\}$
        $OA' = OA \cup \{x\}$
        $AT' = AT \cup \{x\}$
        $PE' = PE \cup \{x\}$
        $ASSIGN' = ASSIGN \cup \{(x, y)\}$
    }
```

CreatePC (x: ID)

```
// add a policy class x to the policy representation
    $x \notin PC$
    $x \notin GUID$
    {
        $GUID' = GUID \cup \{x\}$
        $PC' = PC \cup \{x\}$
        $PE' = PE \cup \{x\}$
    }
```

C.3 Element Deletion Commands

The semantic descriptions of element deletion commands describe the changes in the authorization state that occur with the removal of existing policy elements from the policy representation. Besides ensuring that the arguments supplied for the formal parameters of a command are valid, the preconditions 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.

DeleteU (x: ID)

```
// remove user x from the policy representation
    $x \in U$
    $x \in GUID$
    // ensure no processes that operate on behalf of x exist
    $\nexists p \in P: (p, x) \in Process\_User$
    // ensure no assignments emanating from the user exist
    $\nexists (a, b) \in ASSIGN: x = a$
    // ensure no associations exist in which the user is the third element of the tuple
    $\nexists (a, b, c) \in ASSOCIATION: x = c$
    // ensure no prohibitions exist for the user
    $\nexists (a, b, c, d) \in U\_DENY\_DISJ: a = x$
    $\nexists (a, b, c, d) \in U\_DENY\_CONJ: a = x$
    // ensure no obligations exist defined by the user
    $\nexists (a, b, c) \in OBLIG: a = x$
    {
        $GUID' = GUID \setminus \{x\}$
        $U' = U \setminus \{x\}$
        $PE' = PE \setminus \{x\}$
    }
```

DeleteUA (x: ID)

```
// remove user attribute x from the policy representation
    $x \in UA$
    $x \in GUID$
    // ensure no assignments emanating from or to the user attribute exist
    $\nexists (a, b) \in ASSIGN: (x = a \lor x = b)$
    // ensure no associations exist in which the user attribute is the first or last element of the
    tuple
    $\nexists (a, b, c) \in ASSOCIATION: (x = a \lor x = c)$
    // ensure no attribute prohibitions exist in which the user attribute is the first element of the
    tuple
    $\nexists (a, b, c, d) \in UA\_DENY\_DISJ: x = a$
    $\nexists (a, b, c, d) \in UA\_DENY\_CONJ: x = a$
    // ensure no inclusive element sets exist that involve the user attribute
    $\nexists (a, b) \in ATImap: \exists i \in \{1, ..., \#b\}: (b (i) = x)$
    // ensure no exclusive element sets exist that involve the user attribute
    $\nexists (a, b) \in ATEmap: \exists i \in \{1, ..., \#b\}: (b (i) = x)$
    {
        $GUID' = GUID \setminus \{x\}$
        $UA' = UA \setminus \{x\}$
        $AT' = AT \setminus \{x\}$
        $PE' = PE \setminus \{x\}$
    }
```

DeleteO (x: ID)

```
// remove object x from the policy representation
    $x \in O$
    $x \in OA$
    $x \in GUID$
    // ensure no assignments emanating from or to the object exist
    $\nexists (a, b) \in ASSIGN: (x = a \lor x = b)$
    // ensure no associations exist in which the object is the third element of the tuple
    $\nexists (a, b, c) \in ASSOCIATION: x = c$
    // ensure no inclusive element sets exist that involve the object
    $\nexists(a, b) \in ATImap: \exists i \in \{1, ..., \#b\}: (b (i) = x)$
    // ensure no exclusive element sets exist that involve the object
    $\nexists(a, b) \in ATEmap: \exists i \in \{1, ..., \#b\}: (b (i) = x)$
    {
        $GUID' = GUID \setminus \{x\}$
        $O' = O \setminus \{x\}$
        $OA' = OA \setminus \{x\}$
        $AT' = AT \setminus \{x\}$
        $PE' = PE \setminus \{x\}$
    }
```

DeleteOA (x: ID)

```
// remove object attribute x from the policy representation
    $x \in OA$
    $x \notin O$
    $x \in GUID$
    // ensure no assignments emanating from or to the object attribute exist
    $\nexists(a, b) \in ASSIGN: (x = a \vee x = b)$
    // ensure no associations exist in which the object attribute is the third element of the tuple
    $\nexists(a, b, c) \in ASSOCIATION: x = c$
    // ensure no inclusive element sets exist that involve the object attribute
    $\nexists(a, b) \in ATImap: \exists i \in \{1, ..., \#b\}: (b (i) = x)$
    // ensure no exclusive element sets exist that involve the object attribute
    $\nexists(a, b) \in ATEmap: \exists i \in \{1, ..., \#b\}: (b (i) = x)$
    {
        $GUID' = GUID \setminus \{x\}$
        $OA' = OA \setminus \{x\}$
        $AT' = AT \setminus \{x\}$
        $PE' = PE \setminus \{x\}$
    }
```

DeletePC (x: ID)

```
// remove policy class x from the policy representation
    $x \in PC$
    $x \in GUID$
    $\nexists(a, b) \in ASSIGN: x = b$ // ensure no assignments emanating to the policy class exist
    {
        $GUID' = GUID \setminus \{x\}$
        $PC' = PC \setminus \{x\}$
        $PE' = PE \setminus \{x\}$
    }
```

C.4 Entity creation

The semantic descriptions of entity creation commands describe the state changes that occur with the addition of new entities required by the policy representation. These entities typically appear as items used within the formation of the core PM relations.

CreateP (x: ID, y: ID)

```
// add process x and map it to user y
    $x \notin P$
    $x \notin GUID$
    $y \in U$
    $y \in GUID$
    {
        $GUID' = GUID \cup \{x\}$
        $P' = P \cup \{x\}$
        $Process\_User' = Process\_User \cup \{(x, y)\}$
    }
```

CreateROp (x: ID)

```
// add a resource operation to the policy representation
    $x \notin Op$
    $x \notin GUID$
    {
        $GUID' = GUID \cup \{x\}$
        $Op' = Op \cup \{x\}$
        $ROp' = ROp \cup \{x\}$
    }
```

CreateAOp (x: ID)

```
// add an administrative operation to the policy representation
    $x \notin Op$
    $x \notin GUID$
    {
        $GUID' = GUID \cup \{x\}$
        $Op' = Op \cup \{x\}$
        $AOp' = AOp \cup \{x\}$
    }
```

CreateAR (x: ID)

```
// add an access right to the policy representation
    $x \notin AR$
    $x \notin GUID$
    {
        $GUID' = GUID \cup \{x\}$
        $AR' = AR \cup \{x\}$
    }
```

CreateARset (x: ID, y: 2_1^{AR})

```
// add a set of defined access rights to the policy representation
    $x \notin ARset$
    $x \notin GUID$
    // ensure each element of the set is an access right
    $y \subseteq AR$
    {
        $GUID' = GUID \cup \{x\}$
        $ARset' = ARset \cup \{x\}$ // maintains set of ids for defined access right sets
        $ARmap' = ARmap \cup \{(x, y)\}$ // maintains mapping from ARset ids to AR sets
    }
```

CreateATIset (x: ID, y: 2^{AT})

```
// add a set of inclusion policy elements denoted by a referent attribute to the representation
    $x \notin ATIset$
    $x \notin GUID$
    // ensure each element of the set is the same type of attribute
    $(y \subseteq UA \lor y \subseteq OA)$
    {
        $GUID' = GUID \cup \{x\}$
        $ATIset' = ATIset \cup \{x\}$
        $ATImap' = ATImap \cup \{(x, y)\}$
    }
```

CreateATEset (x: ID, y: 2^{AT})

```
// add a set of exclusion policy elements denoted by a referent attribute to the representation
    $x \notin ATEset$
    $x \notin GUID$
    // ensure each element of the set is the same type of attribute
    $(y \subseteq UA \lor y \subseteq OA)$
    {
        $GUID' = GUID \cup \{x\}$
        $ATEset' = ATEset \cup \{x\}$
        $ATEmap' = ATEmap \cup \{(x, y)\}$
    }
```

CreatePattern (x: ID, y: $seq_1\ \Sigma_P$)

```
// add an event pattern to the policy representation
    $x \notin PATTERNid$
    $x \notin GUID$
    $y \in PATTERN$
    {
        $GUID' = GUID \cup \{x\}$
        $PATTERNid' = PATTERNid \cup \{x\}$
        $PATTERNmap' = PATTERNmap \cup \{(x, y)\}$
    }
```

CreateResponse (x: ID, y: seq$_1$ Σ_R)

```
// add an event response to the policy representation
    $x \notin RESPONSEid$
    $x \notin GUID$
    $y \in RESPONSE$
    {
        $GUID' = GUID \cup \{x\}$
        $RESPONSEid' = RESPONSEid \cup \{x\}$
        $RESPONSEmap' = RESPONSEmap \cup \{(x, y)\}$
    }
```

AllocateID (id′: ID)

```
// allocate an identifier not yet in use by the PM framework
    {
        // no change of the authorization state occurs
        // a valid unused identifier is obtained and returned through the parameter id′
        $id' \notin GUID$
    }
```

C.5 Entity deletion

The semantic descriptions of entity deletion commands describe the state changes that occur with the removal an existing entities from the policy representation.

DeleteP (x: ID)

```
// remove process x from the policy representation
    $x \in P$
    $x \in GUID$
    // ensure the process maps to a user
    $\exists_1 u \in U: u = Process\_User(x)$
    // ensure no outstanding conjunctive prohibitions exist for the process
    $\nexists (a, b, c, d) \in P\_DENY\_CONJ: x = a$
    // ensure no outstanding disjunctive prohibitions exist for the process
    $\nexists (a, b, c, d) \in P\_DENY\_DISJ: x = a$
    {
        $GUID' = GUID \setminus \{x\}$
        $Process\_User' = Process\_User \setminus \{(x, Process\_User(x))\}$
    }
```

DeleteROp (x: ID)

```
// remove a resource operation from the policy representation
    $x \in ROp$
    $x \in Op$
    $x \in GUID$
    {
        $GUID' = GUID \setminus \{x\}$
        $Op' = Op \setminus \{x\}$
        $ROp' = ROp \setminus \{x\}$
    }
```

DeleteAOp (x: ID)

```
// remove an administrative operation from the policy representation
    $x \in AOp$
    $x \in Op$
    $x \in GUID$
    {
        $GUID' = GUID \setminus \{x\}$
        $Op' = Op \setminus \{x\}$
        $AOp' = AOp \setminus \{x\}$
    }
```

DeleteAR (x: ID)

```
// remove an access right from the policy representation
    $x \in AR$
    $x \in GUID$
    // ensure the access right does not belong to any access right set
    $\nexists(a, b) \in ARmap: x \in b$
    {
        $GUID' = GUID \setminus \{x\}$
        $AR' = AR \setminus \{x\}$
    }
```

DeleteARset (x: ID)

```
// remove an access right set from the policy representation
    $x \in ARset$
    $x \in GUID$
    $\exists_1(a, b) \in ARmap: x = a$
    // ensure no associations exist in which the access right set is the second element of the tuple
    $\nexists(a, b, c) \in ASSOCIATION: x = b$
    // ensure no disjunctive user prohibitions exist that involve the access right set
    $\nexists(a, b, c, d) \in U\_DENY\_DISJ: x = b$
    // ensure no conjunctive user prohibitions exist that involve the access right set
    $\nexists(a, b, c, d) \in U\_DENY\_CONJ: x = b$
    // ensure no disjunctive process prohibitions exist that involve the access right set
    $\nexists(a, b, c, d) \in P\_DENY\_DISJ: x = b$
    // ensure no conjunctive process prohibitions exist that involve the access right set
    $\nexists(a, b, c, d) \in P\_DENY\_CONJ: x = b$
    // ensure no disjunctive attribute prohibitions exist that involve the access right set
    $\nexists(a, b, c, d) \in UA\_DENY\_DISJ: x = b$
    // ensure no conjunctive attribute prohibitions exist that involve the access right set
    $\nexists(a, b, c, d) \in UA\_DENY\_CONJ: x = b$
    {
        GUID′ = GUID \ {x}
        ARset′ = ARset \ {x}
        ARmap′ = {x} ⩤ ARmap // the associated sequence of access rights is removed
    }
```

DeleteATIset (x: ID)

```
// remove a set of inclusion attributes from the representation
    x ∈ ATIset
    x ∈ GUID
    $\exists_1$(a, b) ∈ ATImap: x = a
    // ensure no disjunctive user prohibitions exist that involve the attribute set
    ∄(a, b, c, d) ∈ U_DENY_DISJ: x = c
    // ensure no conjunctive user prohibitions exist that involve the attribute set
    ∄(a, b, c, d) ∈ U_DENY_CONJ: x = c
    // ensure no disjunctive process prohibitions exist that involve the attribute set
    ∄(a, b, c, d) ∈ P_DENY_DISJ: x = c
    // ensure no conjunctive process prohibitions exist that involve the attribute set
    ∄(a, b, c, d) ∈ P_DENY_CONJ: x = c
    // ensure no disjunctive attribute prohibitions exist that involve the attribute set
    ∄(a, b, c, d) ∈ UA_DENY_DISJ: x = c
    // ensure no conjunctive attribute prohibitions exist that involve the attribute set
    ∄(a, b, c, d) ∈ UA_DENY_CONJ: x = c
    {
        GUID′ = GUID \ {x}
        ATIset′ = ATIset \ {x}
        ATImap′ = {x} ⩤ ATImap // the associated set of inclusion attributes is removed
    }
```

DeleteATEset (x: ID)

```
// remove a set of exclusion attributes from the representation
    x ∈ ATEset
    x ∈ GUID
    $\exists_1$(a, b) ∈ ATEmap: x = a
    // ensure no disjunctive user prohibitions exist that involve the attribute set
    ∄(a, b, c, d) ∈ U_DENY_DISJ: x = d
    // ensure no conjunctive user prohibitions exist that involve the attribute set
    ∄(a, b, c, d) ∈ U_DENY_CONJ: x = d
    // ensure no disjunctive process prohibitions exist that involve the attribute set
    ∄(a, b, c, d) ∈ P_DENY_DISJ: x = d
    // ensure no conjunctive process prohibitions exist that involve the attribute set
    ∄(a, b, c, d) ∈ P_DENY_CONJ: x = c
    // ensure no disjunctive attribute prohibitions exist that involve the attribute set
    ∄(a, b, c, d) ∈ UA_DENY_DISJ: x = c
    // ensure no conjunctive attribute prohibitions exist that involve the attribute set
    ∄(a, b, c, d) ∈ UA_DENY_CONJ: x = c
    {
        $GUID' = GUID \setminus \{x\}$
        $ATEset' = ATEset \setminus \{x\}$
        $ATEmap' = \{x\} \lhd ATEmap$ // the associated set of exclusion attributes is removed
    }
```

DeletePattern (x: ID)

```
// delete an event pattern from the policy representation
    $x \in PATTERNid$
    $x \in GUID$
    $\exists_1 (a, b) \in PATTERNmap: x = a$
    // ensure no obligations exist that involve the pattern
    $\nexists (a, b, c) \in OBLIG: x = b$
    {
        $GUID' = GUID \setminus \{x\}$
        $PATTERNid' = PATTERNid \setminus \{x\}$
        $PATTERNmap' = \{x\} \lhd PATTERNmap$
    }
```

DeleteResponse (x: ID)

```
// delete an event response from the policy representation
    $x \in RESPONSEid$
    $x \in GUID$
    $\exists_1 (a, b) \in RESPONSEmap: x = a$
    // ensure no obligations exist that involve the response
    $\nexists (a, b, c) \in OBLIG: x = c$
    {
        $GUID' = GUID \setminus \{x\}$
        $RESPONSEid' = RESPONSEid \setminus \{x\}$
        $RESPONSEmap' = \{x\} \lhd RESPONSEmap$
    }
```

DellocateID (id: ID)

```
// deallocate an identifier no longer needed by the PM framework
   $id \notin GUID$
        {
        // no change of authorization state occurs
        // a valid identifier no longer in use is relinquished
        }
```

C.6 Relation Formation Commands

The semantic descriptions of relation formation commands describe state changes that occur with the addition of tuples to existing relations and functions in the policy representation. Besides ensuring that the arguments supplied for the formal parameters of a command are valid, the preconditions specified below must also maintain certain model properties. An attempt to add tuple that already exists to a relation effects no actual change of state, due to the set operation involved (viz., union).

CreatePUmapping (x: ID, y: ID)

```
// add a maplet pair to the function
    $x \in P$
    $y \in U$
    $(x, y) \notin Process\_User$
    // ensure no other user already assigned to the process
    $\nexists z \in P: (x, z) \in Process\_User$
    {
        $Process\_User' = Process\_User \cup \{(x, y)\}$
    }
```

CreateAssign (x: ID, y: ID)

```
// add tuple (x, y) to the assignment relation
    $x \in PE$
    $y \in PE$
    $((x \in U \wedge y \in UA) \vee (x \in UA \wedge y \in UA) \vee (x \in UA \wedge y \in PC) \vee$
    $(x \in OA \wedge y \in (OA \setminus O)) \vee (x \in (OA \setminus O) \wedge y \in PC))$
    // prevents the creation of a loop
    $x \neq y$
    $(x, y) \notin ASSIGN$
    // prevent the creation of a chain of assignments that creates a cycle
    $(x, y \in UA \vee x, y \in OA) \Rightarrow \nexists s \in iseq1\ PE: (\#s > 1 \wedge \forall i \in \{1,...,(\#s - 1)\}:$
    $((s (i), s (i+1)) \in ASSIGN) \wedge (s (1) = y \wedge s (\#s) = x)$
    {
        $ASSIGN' = ASSIGN \cup \{(x, y)\}$
    }
```

CreateAssoc (x: ID, y: ID, z: ID)

```
// add tuple (x, y, z) to the association relation
    $x \in UA$
    $y \in ARset$
    $z \in AT$
    $(x, y, z) \notin ASSOCIATION$
    // ensure no duplicate association exists
    $\nexists(a, b, c) \in ASSOCIATION: (a = x \wedge ARmap(b) = ARmap(y) \wedge c = z)$
    {
        $ASSOCIATION' = ASSOCIATION \cup \{(x, y, z)\}$
    }
```

CreateConjUserProhibit (w: ID, x: ID, y: ID, z: ID)

```
// add tuple (w, x, y, z) to the prohibition relation
    $w \in U$
    $x \in ARset$
    $y \in ATIset$
    $z \in ATEset$
    $(w, x, y, z) \notin U\_DENY\_CONJ$
    $\neg (ATImap(y) = \emptyset \land ATEmap(z) = \emptyset)$
    // ensure no duplicate prohibition exists
    $\nexists(a, b, c, d) \in U\_DENY\_CONJ: (a = w \land ARmap(b) = ARmap(x) \land$
    $ATImap(c) = ATImap(y) \land ATEmap(d) = ATEmap(z))$
    {
        $U\_DENY\_CONJ' = U\_DENY\_CONJ \cup \{(w, x, y, z)\}$
    }
```

CreateConjProcessProhibit (w: ID, x: ID, y: ID, z: ID)

```
// add tuple (w, x, y, z) to the prohibition relation
    $w \in P$
    $x \in ARset$
    $y \in ATIset$
    $z \in ATEset$
    $(w, x, y, z) \notin P\_DENY\_CONJ$
    $\neg (ATImap(y) = \emptyset \land ATEmap(z) = \emptyset)$
    // ensure no duplicate prohibition exists
    $\nexists(a, b, c, d) \in P\_DENY\_CONJ: (a = w \land ARmap(b) = ARmap(x) \land$
    $ATImap(c) = ATImap(y) \land ATEmap(d) = ATEmap(z))$
    {
        $P\_DENY\_CONJ' = P\_DENY\_CONJ \cup \{(w, x, y, z)\}$
    }
```

CreateConjAttributeProhibit (w: ID, x: ID, y: ID, z: ID)

```
// add tuple (w, x, y, z) to the prohibition relation
    $w \in UA$
    $x \in ARset$
    $y \in ATIset$
    $z \in ATEset$
    $(w, x, y, z) \notin UA\_DENY\_CONJ$
    $\neg (ATImap(y) = \emptyset \land ATEmap(z) = \emptyset)$
    // ensure no duplicate prohibition exists
    $\nexists(a, b, c, d) \in UA\_DENY\_CONJ: (a = w \land ARmap(b) = ARmap(x) \land$
    $ATImap(c) = ATImap(y) \land ATEmap(d) = ATEmap(z))$
    {
        $UA\_DENY\_CONJ' = UA\_DENY\_CONJ \cup \{(w, x, y, z)\}$
    }
```

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

CreateOblig (x: ID, y: ID, z: ID)

```
// add tuple (x, y, z) to the obligation relation
    $x \in U$
    $y \in PATTERNid$
    $z \in \text{RESPONSEid}$
    $(x, y, z) \notin \text{OBLIG}$
    // ensure no duplicate (i.e., identical sentences, not semantic equivalents) obligation exists
    $\nexists(a, b, c) \in \text{OBLIG}: (a = x \land \#\text{PATTERNmap}(b) = \#\text{PATTERNmap}(y) \land$
    $\forall i \in \{1, ..., \#\text{PATTERNmap}(b)\}: \text{PATTERNmap}(b)\ (i) = \text{PATTERNmap}(y)\ (i) \land$
    $\#\text{RESPONSEmap}(c) = \#\text{RESPONSEmap}(z) \land$
    $\forall i \in \{1, ..., \#\text{RESPONSEmap}(c)\}: \text{RESPONSEmap}(c)\ (i) = \text{RESPONSEmap}(z)\ (i))$
    {
        $\text{OBLIG}' = \text{OBLIG} \cup \{(x, y, z)\}$
    }
```

C.7 Relation Rescindment Commands

The semantic descriptions of relation rescindment commands describe state changes that occur with the removal of tuples from existing relations and functions in the policy representation. 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 effects no change of state, due to the set operation involved.

DeletePUmapping (x: ID, y: ID)

```
// delete a maplet pair from the function
    $x \in P$
    $y \in U$
    $(x, y) \in \text{Process\_User}$
    {
        $\text{Process\_User}' = \text{Process\_User} \setminus \{(x, y)\}$
    }
```

DeleteAssign (x: ID, y: ID)

```
// remove tuple (x, y) from the assignment relation
    $((x \in U \land y \in UA) \lor (x \in UA \land y \in UA) \lor (x \in UA \land y \in PC) \lor$
    $(x \in OA \land y \in (OA \setminus O)) \lor (x \in (OA \setminus O) \land y \in PC))$
    $(x, y) \in \text{ASSIGN}$
    // ensure that if no other assignment emanates from x, no assignments emanate to x
    $\nexists z \in PE: (x, z) \in \text{ASSIGN} \Rightarrow \nexists v \in PE: (v, x) \in \text{ASSIGN}$
    {
        $\text{ASSIGN}' = \text{ASSIGN} \setminus \{(x, y)\}$
    }
```

DeleteAssoc (x: ID, y: ID, z: ID)

```
// remove tuple (x, y, z) from the association relation
    $x \in UA$
    $y \in \text{ARset}$
    $z \in AT$
    $(x, y, z) \in \text{ASSOCIATION}$
    {
        ASSOCIATION′ = ASSOCIATION \ {(x, y, z)}
    }
```

DeleteConjUserProhibit (w: ID, x: ID, y: ID, z: ID)

```
// remove tuple from user prohibition relation
    w ∈ U
    x ∈ ARset
    y ∈ ATIset
    z ∈ ATEset
    (w, x, y, z) ∈ U_DENY_CONJ
    {
        U_DENY_CONJ′ = U_DENY_CONJ \ {(w, x, y, z)}
    }
```

DeleteConjProcessProhibit (w: ID, x: ID, y: ID, z: ID)

```
// remove tuple from process prohibition relation
    w ∈ P
    x ∈ ARset
    y ∈ ATIset
    z ∈ ATEset
    (w, x, y, z) ∈ P_DENY_CONJ
    {
        P_DENY_CONJ′ = P_DENY_CONJ \ {(w, x, y, z)}
    }
```

DeleteConjAttributeProhibit (w: ID, x: ID, y: ID, z: ID)

```
// remove tuple from process prohibition relation
    w ∈ UA
    x ∈ ARset
    y ∈ ATIset
    z ∈ ATEset
    (w, x, y, z) ∈ UA_DENY_CONJ
    {
        UA_DENY_CONJ′ = UA_DENY_CONJ \ {(w, x, y, z)}
    }
```

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

DeleteOblig (x: ID, y: ID, z: ID)

```
// remove tuple (x, y, z) from the obligation relation
    x ∈ U
    y ∈ PATTERNid
    z ∈ RESPONSEid
    (x, y, z) ∈ OBLIG
    {
```

```
    OBLIG′ = OBLIG \ {(x, y, z)}
}
```

Appendix D—Administrative Routines

The administrative routines specified in this appendix correspond, on a one-to-one basis, to the set of administrative operations allowable in an administrative access request. Reference mediation must be successfully carried out by the access decision function as a prerequisite to carrying out the behavior described in the body of an administrative routine. The access decision function verifies that a process holds sufficient authorization to perform an administrative routine, based the operation used in the access request, the required permissions needed to perform that operation over the argument sequence in question, and the absence of any restrictions to the contrary.[14]

Administrative routines perform changes to the authorization state exclusively through one or more administrative commands. In addition to the mediation by the access decision function, administrative routines also rely on the preconditions of administrative commands to ensure that the arguments supplied for the formal parameters of a routine are valid and that the behavior of commands maintain the integrity of the model. Therefore, the preconditions of administrative routines are relatively sparse, especially when compared with those of administrative commands.

D.1 Relation Formation Routines

C-UinUA (u′: ID, ua: ID)

```
// create a u assigned to ua
// process must have (c-u, ua), (c-uua, ua) capabilities to reach this point
        {
        AllocateID (u′);
        CreateUinUA (u′, ua);
        }
```

C-UAinUA (x′: ID, ua: ID)

```
// create a ua, x, assigned to ua
// process must have (c-ua, ua), (c-uaua, ua) capabilities to reach this point
        {
        AllocateID (x′);
        CreateUAinUA (x′, ua);
        }
```

C-UAinPC (ua′: ID, pc: ID)

```
// create a ua in pc
// process must hold (univ, framework) capabilities to reach this point
    {
        AllocateID (ua′);
```

[14] The same is true for routines used to act upon resources. However, those routines are dependent on the computational environment, and therefore, their semantics are not described in this report.

```
        CreateUAinPC (ua′, pc);
    }
```

C-PC (pc′: ID)

```
// create a policy class
// process must hold (univ, framework) capabilities to reach this point
    {
        AllocateID (pc′);
        CreatePC (pc′);
    }
```

C-UtoUA (u: ID, ua: ID)

```
// create an assignment (u, ua)
// process must hold either (c-uua-fr, u), (c-uua-to, ua) capabilities to reach this point,
// or if $\exists x \in PC$: (u $ASSIGN^+$ x $\wedge$ ua $ASSIGN^+$ x), (c-uua, ua) capabilities
    $u \in U$
    $ua \in UA$
    {
        CreateAssign (u, ua);
    }
```

C-UAtoUA (uafr: ID, uato: ID)

```
// create an assignment (uafr, uato)
// process must hold either (c-uaua-fr, uafr), (c-uaua-to, uato) capabilities to reach this point,
// or if $\exists x \in PC$: (uafr $ASSIGN^+$ x $\wedge$ uato $ASSIGN^+$ x), (c-uaua, uato) capabilities
    $uafr, uato \in UA$
    {
        CreateAssign (uafr, uato);
    }
```

C-UAtoPC (ua: ID, pc: ID)

```
// create an assignment (ua, pc)
// process must hold (univ, framework) capabilities to reach this point
    $pc \in PC$
    $ua \in UA$
    {
        CreateAssign (ua, pc);
    }
```

Relation formation routines for creating and assigning objects and object attributes are defined similarly to those given above for users and user attributes.

C-Assoc (ua: ID, ars: 2_1^{AR}, at: ID)

```
// create an association (ua, ars, at)
// process must hold (c-assoc-fr, ua), (c-assoc-to, at) capabilities to reach this point
    arset: ID   // declaration of a local variable
    {
        AllocateID (arset′);
        CreateARset (arset′, ars);
        CreateAssoc (ua, arset′, at);
    }
```

C-ConjUProhib (u: ID, ars: 2_1^{AR}, atis: 2^{AT}, ates: 2^{AT})

```
// create a conjunctive user prohibition
// process must hold the capabilities (c-prohib-fr, u), (c-prohib-to, ati) for ∀ati ∈ atis, and
// (c-prohib-to, ate) for ∀ate ∈ ates to reach this point
    // ensure each member of the inclusive and exclusive attribute sets are of the same type
    ((atis ∪ ates) ⊆ UA ⋁ (atis ∪ ates) ⊆ OA)
    arset, atiset, ateset: ID   // declaration of local variables
    {
        AllocateID (arset′);
        CreateARset (arset′, ars);
        AllocateID (atiset′);
        CreateATIset (atiset′, atis);
        AllocateID (ateset′);
        CreateATEset (ateset′, ates);
        CreateConjUserProhibit (u, arset′, atiset′, ateset′);
    }
```

C-ConjPProhib (p: P, ars: 2_1^{AR}, atis: 2^{AT}, ates: 2^{AT})

```
// create a conjunctive process prohibition
// process must hold the capabilities (c-prohib-fr, Process-User(p)), (c-prohib-to, ati)
// for ∀ati ∈ atis, and (c-prohib-to, ate) for ∀ate ∈ ates to reach this point
    // ensure each member of the inclusive and exclusive attribute sets are of the same type
    ((atis ∪ ates) ⊆ UA ⋁ (atis ∪ ates) ⊆ OA)
    arset, atiset, ateset: ID   // declaration of local variables
    {
        AllocateID (arset′);
        CreateARset (arset′, ars);
        AllocateID (atiset′);
        CreateATIset (atiset′, atis);
        AllocateID (ateset′);
        CreateATEset (ateset′, ates);
        CreateConjProcessProhibit (p, arset′, atiset′, ateset′);
    }
```

C-ConjUAProhib (ua: ID, ars: 2_1^{AR}, atis: 2^{AT}, ates: 2^{AT})

```
// create a conjunctive user attribute prohibition
// process must hold the capabilities (c-prohib-fr, ua), (c-prohib-to, ati) for ∀ati ∈ atis, and
// (c-prohib-to, ate) for ∀ate ∈ ates to reach this point
    // ensure each member of the inclusive and exclusive attribute sets are of the same type
    ((atis ∪ ates) ⊆ UA ⋁ (atis ∪ ates) ⊆ OA)
```

```
arset, atiset, ateset: ID   // declaration of local variables
{
     AllocateID (arset′);
     CreateARset (arset′, ars);
     AllocateID (atiset′);
     CreateATIset (atiset′, atis);
     AllocateID (ateset′);
     CreateATEset (ateset′, ates);
     CreateConjAttributeProhibit (ua, arset′, atiset′, ateset′);
}
```

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

EvalPattern (p: ID, pattern: seq$_1$ Σ_P), returns Boolean

```
{
     /* A semantic function that evaluates the correctness of a logical expression of
     an event pattern involving the policy elements and relations of the PM
     (provided as an input string).  It also verifies that the process holds sufficient
     authority over each recognized policy element in the pattern.  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 (p: ID, response: seq$_1$ Σ_R), returns Boolean

```
{
     /* A semantic function that evaluates of the correctness of the syntax of an obligation's
     response (provided as an input string).  It also verifies that the process holds
     sufficient authority over each recognized policy element in the response.  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 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 returned in the event context or to policy elements and relations that may not be resolved until the time of match. However, some syntax checks can be made at the time of creation to filter out incorrect expressions and verify that a string supplied as an event pattern is well-formed according to its respective grammar. Where possible, verification should be made that the creator of an obligation holds c-oblig access rights over each recognized policy element that is referenced in the pattern and the obligation response.

The creation of obligations is modeled by the C-Oblig routine. The preconditions 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 the actions to be taken (i.e., the EvalResponse function returns True). The semantics for these routines entail 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 verification, at the time a match to the obligation occurs, that the authorization for the user is sufficient to execute the response.

```
C-Oblig (p#: ID, pattern: seq1 ΣP, response: seq1 ΣR)[15]
// create an obligation
    p ∈ P
    // verify that the grammars are well-formed and the process has sufficient authorization
    (EvalPattern(p, pattern) ∧ EvalResponse(p, response))
    {
        AllocateID (patternid′);
        CreatePattern (patternid′, pattern);
        AllocateID (responseid′);
        CreateResponse (responseid′, response);
        CreateOblig (Process_User(p), patternid′, responseid′);
    }
```

D.2 Relation Rescindment Routines

```
D-UinUA (u: ID, ua: ID)
// Delete user assigned to this user attribute
// process must have (d-u, ua) capabilities and either (d-uua, ua) or (d-uua-fr, u), (d-uua-to, ua)
// capabilities to reach this point
    u ∈ U
    ua ∈ UA
    (u, ua) ∈ ASSIGN
    {
        DeleteAssign (u, ua);
        DeleteU (u);            // routine fails if any relations exist that involve u
    }
```

```
D-UtoUA (u: ID, ua: ID)
// delete the assignment from u to ua
```

[15] Note that the pound sign (#) after the parameter is used here to denote an input variable not under the control of the entity invoking the operation, following the convention established in [Per96]. It is presumed that such a variable is assigned a value by the underlying system at the time of invocation.

```
// process must have either (d-uua, ua) or (d-uua-fr, u), (d-uua-to, ua) capabilities
// to reach this point
    u ∈ U
    ua ∈ UA
    (u, ua) ∈ ASSIGN
    {
        DeleteAssign (u, ua);
    }
```

D-UAinUA (uafr: ID, uato: ID)

```
// Delete user attribute assigned to this user attribute
// process must have (d-ua, uato) capabilities and either (d-uaua, uato) or
// (d-uaua-fr, uafr), (d-uaua-to, uato) capabilities to reach this point
    uafr, uato ∈ UA
    (uafr, uato) ∈ ASSIGN
    {
        DeleteAssign (uafr, uato);
        DeleteUA (uafr);            // routine fails if any relations exist that involve uafr
    }
```

D-UAtoUA (uafr: ID, uato: ID)

```
// delete the assignment from uafr to uato
// process must have either (d-uaua, uato) or (d-uaua-fr, uafr), (d-uaua-to, uato) capabilities
// to reach this point
    u ∈ U
    ua ∈ UA
    (u, ua) ∈ ASSIGN
    {
        DeleteAssign (uafr, uato);
    }
```

D-UAinPC (ua: ID, pc: ID)

```
// Delete user attribute assigned to the policy class
// process must have (univ, framework) capabilities to reach this point
    ua ∈ UA
    pc ∈ PC
    (ua, pc) ∈ ASSIGN
    {
        DeleteAssign (ua, pc);
        DeleteUA (ua);            // routine fails if any relations exist that involve ua
    }
```

D-UAtoPC (ua: ID, pc: ID)

```
// delete the assignment from ua to pc
// process must have either (univ, framework) capabilities to reach this point
    ua ∈ UA
    $pc \in PC$
    $(ua, pc) \in ASSIGN$
    $\exists x \in PC: (x \neq pc \land ua\ ASSIGN^{+}\ x)$     // ensures that the ua can reach some other PC
    {
        DeleteAssign (ua, pc);
    }
```

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

D-Assoc (ua: ID, ars: 2_1^{AR}, at: ID)

```
// delete an association
// process must hold (d-assoc-fr, ua), (d-assoc-to, at) capabilities to reach this point
    $ua \in UA$
    $at \in AT$
    a, b: ID  // local variable declaration
    $\exists_1(ua, a, at) \in ASSOCIATION: (\exists(a, b) \in ARmap: ars = b)$
        {
        DeleteAssoc (ua, a, at);
        DeleteARset (a);
        DeallocateID (a);
        }
```

D-ConjUProhib (u: ID, ars: 2_1^{AR}, atis: 2^{AT}, ates: 2^{AT})

```
// delete a prohibition
// process must hold (d-prohib-fr, u), (d-prohib-to, at) capabilities for $\forall ati \in atis$, and
// (d-prohib-to, ate) for $\forall ate \in ates$ to reach this point
    $u \in U$
    a, b, c, d, e, f: ID   // local variable declaration
    $\exists_1(u, a, c, d) \in U\_DENY\_CONJ: (\exists(a, b) \in ARmap: ars = b \land$
    $\exists(c, d) \in ATImap: atis = d \land \exists(e, f) \in ATEmap: ates = f)$
    {
        DeleteConjUserProhibit (u, a, c, d);
        DeleteARset (a);
        DeallocateID (a);
        DeleteATIset (c);
        DeallocateID (c);
        DeleteATeset (d);
        DeallocateID (d);
    }
```

D-ConjPProhib (p: ID, ars: 2_1^{AR}, atis: 2^{AT}, ates: 2^{AT})

```
// delete a prohibition
// process must hold (d-prohib-fr, Process_User(p)), (d-prohib-to, at) capabilities for $\forall ati \in atis$,
// and (d-prohib-to, ate) for $\forall ate \in ates$ to reach this point
    $p \in P$
```

```
a, b, c, d, e, f: ID   // local variable declaration
$\exists_1$(p, a, c, d) ∈ P_DENY_CONJ: (∃(a, b) ∈ ARmap: ars = b  ⋀
∃(c, d) ∈ ATImap: atis = d  ⋀  ∃(e, f) ∈ ATEmap: ates = f )
{
    DeleteConjProcessProhibit (p, a, c, d);
    DeleteARset (a);
    DeallocateID (a);
    DeleteATIset (c);
    DeallocateID (c);
    DeleteATeset (d);
    DeallocateID (d);
}
```

D-ConjUAProhib (ua: ID, ars: 2_1^{AR}, atis: 2^{AT}, ates: 2^{AT})

```
// delete a prohibition
// process must hold (d-prohib-fr, ua), (d-prohib-to, at) capabilities for ∀ati ∈ atis, and
// (d-prohib-to, ate) for ∀ate ∈ ates to reach this point
    ua ∈ UA
    a, b, c, d, e, f: ID   // local variable declaration
    $\exists_1$(u, a, c, d) ∈ UA_DENY_CONJ: (∃(a, b) ∈ ARmap: ars = b  ⋀
    ∃(c, d) ∈ ATImap: atis = d  ⋀  ∃(e, f) ∈ ATEmap: ates = f )
    {
        DeleteConjAttributeProhibit (ua, a, c, d);
        DeleteARset (a);
        DeallocateID (a);
        DeleteATIset (c);
        DeallocateID (c);
        DeleteATeset (d);
        DeallocateID (d);
    }
```

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

D-Oblig (p#: ID, u: ID, pattern: seq$_1$ Σ_P, response: seq$_1$ Σ_R)

```
    p ∈ P
    u ∈ U
    patternid, b, responseid, d: ID  // local variable declaration
    $\exists_1$(u, patternid, responseid) ∈ OBLIG: (∃(patternid, b) ∈ PATTERNmap: pattern = b) ⋀
    ∃(responseid, d) ∈ RESPONSEmap: response = d)
   // ensure that the process has authorization to delete the obligation
    (EvalPattern(p, pattern)  ⋀  EvalResponse(p, response))
    {
        DeleteOblig (u, patternid, responseid);
        DeletePattern (patternid);
        DeallocateID (patternid);
        DeleteResponse (responseid);
        DeallocateID (responseid);
    }
```

Appendix E—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. This situation could eventually lead to problems as policy evolves and personnel changes occur, since the linkage between identities must be maintained elsewhere, and typically would be done less rigorously than the policy specification. Therefore, it would be preferable to accommodate this type of functionality explicitly within the policy specification. Three general approaches are possible: extending the PM model, defining and triggering obligations, and applying a role-based orientation.

E.1 Via Model Extension

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. In other words, a change in personas can be done simply by deleting an assignment and creating another.

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 routine that carries out the indicated action. To grant the requisite authority, the system administrator has only to establish administrative associations between the user in question and user attributes representing its personas, which allow the user to replace the current assignment to one of the user attributes with an assignment to the other. With that authority in place, the user can initiate the administrative routine via an administrative access request to cause its assignment to change.

The administrative routine 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 routine follows that described in Appendix D.

SwitchAssignmentBetweenUAs (p#: ID, u: ID, uacurrent: ID, uanew: ID)

```
// process must hold reassign-user authorization over uacurrent and uanew,
// i.e., (reassign-user-fr, uacurrent), (reassign-user-to, uanew) capabilities, to reach this point
   u ∈ U
   uacurrent, uanew ∈ UA
   // u must be assigned to uacurrent, not uanew, and the user is that of the requesting process
   (u, uacurrent) ∈ ASSIGN  ∧  (u, uanew) ∉ ASSIGN  ∧  u = Process_User(p)
   {
       CreateAssign (u, uanew)
       DeleteAssign (u, uacurrent)
   }
```

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 several administrative modes or several 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 6 illustrates usage of the extension more concretely. For that policy, u_2 and u_4 are presumed to represent two personas for a single individual. Applying the above solution eliminates the need for u_4 to be defined, since the single user u_2 will suffice. The following steps are required to define the policy:

- Create the association (OUadmin, {reassign-user-to}, Group2) ∈ ASSOCIATION, which grants users assigned to OUadmin the authorization to change that assignment to Group2.

- Create the association (OUadmin, {reassign-user-fr}, OUadmin) ∈ ASSOCIATION, which grants users assigned to OUadmin the authorization to change that assignment away from OUadmin.

- Create the association (Group2, {reassign-user-to}, OUadmin) ∈ ASSOCATION, which grants users assigned to Group2 the authority to change that assignment to OUadmin.

- Create the association (Group2, {reassign-user-fr}, Group2) ∈ ASSOCIATION, 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 via its process to the other persona attribute by issuing the administrative access request (p, switch-assignment, ⟨Group2, OUadmin⟩), which in turn results in the execution of the administrative routine SwitchAssignmentBetweenUAs(p, u_2, Group2, OUadmin) 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 6 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.

E.2 Via Obligations

The second way to accommodate personas is through obligations. This approach would involve defining a routine 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 execution of the routine when the obligation is matched and the response triggered. In this case, however, no new access right like reassign-user would apply to authorizations; instead, an access right for an operation on a resource 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 6 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 the obligation's response 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 routine 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 via an extension 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 a single class of administrator is needed, obligations may provide a useful means to support personas.

E.3 Via a Role-based Orientation

The third approach for incorporating the functionality of personas is to treat each persona as though it were a role. This is by far the best approach, since it can be done entirely through the policy specification, requiring no extensions to the model as in the first approach (viz., new administrative routines) or triggering of obligations through special-purpose files, as in the second. To begin, a slight change is needed to the initial policy represented in Figure 6. The user u_4 would be eliminated, user u_2 would be assigned to a new attribute, persona-u_2, instead of Group2, and persona-u_2 would be assigned to Administrators. The following steps can then be used to define the base policy:

- Create the administrative association (persona-u_2, {c-uaua-fr, d-uaua-fr}, persona-u_2) $\in$ ASSOCIATION, which grants u_2 the authorization to create or delete an assignment from persona-u_2 to another user attribute for which it holds respectively c-uaua-to or d-uaua-to assignment authority.

- Create the administrative association (persona-u_2, {c-uaua-to, d-uaua-to}, OUadmin) ∈ ASSOCIATION, which grants u_2 the authorization to create or delete an assignment to OUadmin from another user attribute (viz., persona-u_2) for which it respectively holds c-uaua-fr or d-uaua-fr assignment authority.

- Create the administrative association (persona-u_2, {c-uaua-to, d-uaua-to}, Group2) ∈ ASSOCIATION, which grants u_2 the authorization to create or delete an assignment to Group2 from another user attribute (viz., persona-u_2) for which it respectively holds c-uaua-fr or d-uaua-fr assignment authority.

The above policy allows a process operating on behalf of u_2 to create and delete assignments that allow it to operate with OUadmin authorizations, Group2 authorizations, and both OUadmin and Group2 authorizations. To avoid the latter from occurring and have all u_2 processes operate under either OUadmin or Group2 authorizations, the following dynamic separation of duty obligations must be defined.

When EC.op = create-UAtoUA ∧ EC.u ASSIGN$^+$ persona-u_2 ∧
EC.argseq (2) = Group2 **do**
    CreateOblig-DisjUProhib (EC.p, {assign-to}, {UAadmin}, ∅)[16,17]

When EC.op = create-UAtoUA ∧ EC.u ASSIGN$^+$ persona-u_2 ∧
EC.argseq (2) = OUadmin **do**
    CreateOblig-DisjUProhib (EC.p, {assign-to}, {Group2}, ∅)

When EC.op = delete-UAtoUA ∧ EC.u ASSIGN$^+$ persona-u_2 ∧
EC.argseq (2) = Group2 **do**
    DeleteOblig-DisjUProhib (EC.p, {assign-to}, {UAadmin}, ∅)[18]

When EC.op = delete-UAtoUA ∧ EC.u ASSIGN$^+$ persona-u_2 ∧
EC.argseq (2) = OUadmin **do**
    DeleteOblig-DisjUProhib (EC.p, {assign-to}, {Group2}, ∅)

[16] EC.argseq of an event context for an access request involving the create-UAtoUA or delete-UAtoUA administrative operations is ⟨ua-fr, ua-to⟩, where EC.argseq (1) contains ua-fr, the identifier of the user attribute from which the assignment begins, and EC.argseq (2) contains ua-to, the identifier of the user attribute to which the assignment ends.

[17] The semantics of the administrative routine used in this obligation is essentially the same as that for the routine C-DisjUProhib given in Appendix D, with one exception—the user that defined the obligation, not the user whose process triggered the obligation, must hold sufficient authorization to perform the body of the routine.

[18] The D-DisjUProhib routine given in Appendix D has essentially the same semantics as this administrative routine, with the caveat of differences in the authorization required.