On Usage Control for Data Grids: Models, Architectures, and Specifications

Federico Stagni

Submitted in total fulfilment of the requirements
of the degree of Doctor of Philosophy

March 2009

University of Ferrara

Abstract

This thesis reasons on usage control in Data Grids, by presenting models, architectures and specifications. This work is a step toward a continuous monitoring and control of the data access and usage in a Data Grid. First, the thesis presents a background on Grids, security, and security for Grids, by making an abstraction to the current Grid implementations. We argue that usage control in Data Grids should be considered as a process composed by two black boxes. We analysed the requirements for Grid security, and propose a distributed usage control model suitable for Grids and distributed systems alike. Then, we apply such model to a Data Grid abstraction, and present a usage control architecture for Data Grids that uses the functional components of the currents Grids. We also present an abstract specification for an enforcing mechanism for usage control policies. To do so, we use a formal requirement engineering methodology with a bottom-up approach, that proves that the specification is sound and complete. With the methodology, we show formally that such abstract specification can enforce all the different typologies of usage control policies. Finally, we consider how existing prototypes can fit in the proposed architecture, and the advantages derived from using Semantic Grid techologies for the specification of policies subjects and objects.

Declaration

This dissertation is the result of my own work, except where explicit reference is made to the work of others, and has not been submitted for another qualification to this or any other university

For my father

Contents

1 Introduction

1.1 Problem Statement

Data Grids are an innovative technology taking advantage of existing computer science concepts in file systems, database systems and Grid computing. A Data Grid provides services that help users discover, transfer, and manipulate large datasets stored in distributed repositories and create and manage copies of these datasets. However, as in any resource sharing environment, robust and rigorous treatment of data security in a Data Grid is vital. Moreover, since data is being shared over multiple administrative domains over the Grid, continuous monitoring and control of the data access is required.

This thesis wants to solve some of the Data Grids security problems, by applying usage control techniques. Usage control extends traditional access control by controlling data access as well as usage.

1.2 Approach

We consider Grid security as a process composed by two black boxes: Grid Authentication and Grid Authorization. We present solutions for both.

We develop here a usage control model suitable for multi-authoritative distributed systems. We base this model on the UCON$_{abc}$ model proposed by Park and Sandhu [77]. One of the main contributions of the thesis is a Data Grid usage control architecture using the functional components of the current Grids, as presented by the Open Grid Forum (OGF) group on Grid authorization [18].

In doing so, we use formal goal-oriented methodology elaborated from KAOS [99].

We also make a proposal for extending the Grid usage control architecture with Semantic information for the specification of UCON subjects and objects. This way we can control the policy granularity.

1.3 Organization of the thesis

The thesis has been divided in three Parts. Part I is about Grids theory and systems, general security concepts and security for Grids. It includes Chapter 2, which explains what is a Grid, Chapter 3, which encapsulates general concepts found in the computer security world, and Chapter 4, which explains some basic notions about controlling the usage of Grid data. These notions will remain valid throughout the whole thesis. Part II summarizes the working instruments used for producing the results of the thesis. It is composed by two Chapters. Chapter 5 explains what is a security policy model, focussing on the UCON usage control model. Chapter 6 explains the software engineering instruments used in the proceedings of the work. Part III explains the models, architectures and specifications produced in the thesis. Chapter 7 explains the foundations for developing a software for usage control for Data Grids, including a distributed usage control model. Chapter 8 is about the re-engineering of a software, the VOMS tool, which provides an extended Grid authentication. Chapter 9 shows two architectures for Data Grid usage control, focussing on the authorization part. Chapter 10 shows a methodology for verifying a formal software specification using the KAOS formal requirement engineering methodology. Then, the methodology is applied for verifying that a formal abstract architecture specification, showed in Chapter 9, is correct. Finally, in Chapter 11 readers may find the conclusions of the thesis, and ideas for future works.

Part I

Background On Grids and Security

2
Notions About Grids

This Chapter serves as an introduction for Grids. The work produced within this thesis is done having in mind a definite vision of what is a Grid. There's definitely the need for such a vision. Throughout the years, many commercial and non-commercial institutes produced implementations of some kind of a distributed system, calling it "a Grid". It's fair to say that the term "Grid" now assumes different meanings according to the different implementors.

We don't deal with any particolar implementation, but we deal with a vision of what's a Grid and its purpose. This Chapter explains a brief Grid taxonomy, and uses abstractions for describing the Grid elements. As is our goal, we focus on Data Grids.

2.1 Grid systems definitions and abstractions

Grid systems and technologies have evolved over nearly a decade; yet, there is still no widely accepted definition for Grids [73]. Grids can be viewed as successors to *distributed computing* environments, but still no one articulated the real differences between the two. One of the most accepted common views defines the Grid as a high-performance distributed environment; some Grid practitioners take into consideration its geographically distributed, multi-domain feature, while others define Grids based on the number of resources they unify, and so on.

In [33], Foster defines the Grid as

> *Flexible, secure, coordinated resource sharing among dynamic collections of individuals, institutions, and resources.*

In [41] and in [39], Grimshaw talks of the Grid as

> *A single seamless computational environment in which cycles, communication, and data are shared, and in which the workstation across the continent is no less than one down the hall.*

and as

> *A wide-area environment that transparently consists of workstations, personal computers, graphic rendering engines, supercomputers and non-traditional devices: e.g., TVs, toasters, etc...*

In [40], the same author defines the Grid as

> *A collection of geographically separated resources (people, computers, instruments, databases) connected by a high speed network [...distinguished by...] a software layer, often called middleware, which transforms a collection of independent resources into a single, coherent, virtual machine.*

We believe that a good definition of a Grid should take in consideration abstraction and resource virtualization. According to Nemeth and Sunderam [72], the key properties that make Grid different from conventional distributed systems are:

1. resources are abstracted and form virtual pools;
2. users who require access to resources of the virtual pools are different from users who have valid accounts and login rights to physical resources;
3. mapping functions are available for entities translating from virtual to physical resources and users;
4. the number of resources in the pool is in the order of thousands or greater.

Grids are then realized by providing users and resource abstractions, and mapping functionalities.

Grid users abstraction through Virtual Organizations

A Grid users abstraction is realized through the concept of *Virtual Organization* (VO) [33]. In a VO, a varying number of participants with various degrees of prior relationships, join in order to share resources. Resource sharing is conditional: Resource Providers (RPs) make resources available subject to a number of constraints on who can use them, when, and for what reasons. Such constraints are agreed between RPs and VOs. The concept of Virtual Organizations is what distinguishes the Grid as a concept from other known domains of computing (such as distributed computing, cluster computing). VO participants are abstract users that are dynamically mapped into physical ones during the assignment of a process to a physical resource. Within a certain VO, the user can have a number of different attributes. These attributes can be considered during the mapping phase to improve the capabilities for a dynamic distribution of resources. Usually

VO managers can add or withdraw a user membership, and they can update the user attributes like group or role. In fig. 2.1 we provide an overview of the theoretical relationships that take place inside a VO between users, RPs, resources and middlewares.

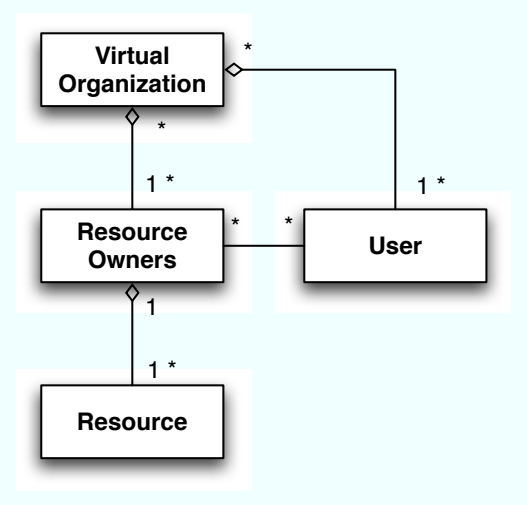

Figure 2.1: Relationships in a Virtual Organization [104]

Resource Abstraction

Resource abstraction is realized through virtual pools. Grid Information systems provide Grid users with attributes and capabilities of the underlying resource. VO users don't need a priori knowledge about the actual type, state and features of the physical resources part of the dynamic pool to which (s)he has access rights. VO users can delegate to a Grid system the selection of a suitable physical resource that matches the description of the abstract one. In this way, it is possible to optimize the usage of the available resources amongst a large community of users.

Mapping Functionalities

Mapping functionalities provide a map between physical resources and physical users for the actual execution of the user process. To do so, mapping services use informations provided by the virtual layers of abstract resources and abstract users. Such a mapping is performed as an effect of a user request.

Normally, a site administrator can decide to map a certain group of users of a certain VO to a set of resources with some kind of priority access. The mapping performed by the Grid system should be able to deal with either

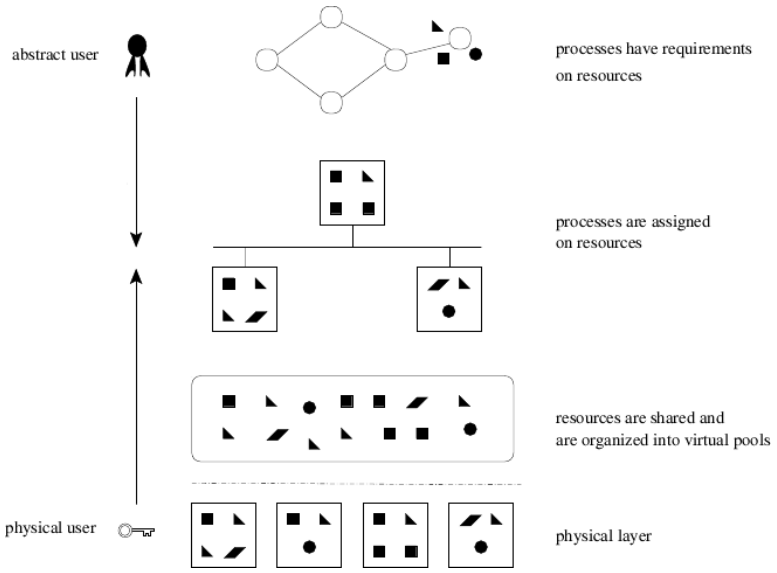

Figure 2.2: Abstraction of a Grid System [4]

the characteristics that the target resource *must* have and the characteristics that the target resource *may* have. It can also take into account the optimization of the resource distribution.

Figure 2.2, which is taken from [4], shows the whole virtualization process.

2.1.1 Grid and the power Grid

The Grid is usually compared with an electrical power Grid. For instance, electrical devices can be plugged into sockets, which provide a well-defined quantity of power. The user of the device isn't concerned as to where the power comes from, nor how it is delivered. It is also irrelevant whether the power was generated by a coal, nuclear or hydroelectric plant: this heterogeneity is masked by the power Grid.

Recently, this analogy have been expounded in [22], where the authors highlight some differences between the power Grid and the computational Grids. Computational Grids must harness not only the processing power of hardware resources such as individual Personal Computers (PCs) and site clusters, but must also deal with more complex resources such as databases. Whereas any device with a plug may draw power from the electrical power Grid, there is no 'universal adapter' for Grid computing systems. Many heterogeneous compute resources exist, and it must be possible for all to gain access to the Grid (inter-Grid compatibility must also be assured). Likewise, software applications running on the Grid need an easy way to 'plug in' to computing resources. Another important point is security: in

a Grid environment, providing a secure way for users to run applications on remote resources, which they do not necessarily own, is still not clear. Many of the issues regarding Grid security will be introduced in subsequent Sections.

It is fair to say that the added complexity of Computational Grids limits the effectiveness of the power Grid analogy. However, the idea of computing power as a utility is certainly appealing and could eventually become a reality. This potential is only beginning to be realised with Grid computing but with a steadily increasing requirement for computational power, Grid systems are set to become more prevalent through necessity.

2.2 On the construction of Grid systems

Grids are complex computing systems. For their construction, there are three fundamental Grid blocks to consider:

1. **Grid Design** Grid engineers must specify an overall design before they start work. The design identifies the fundamental components of a Grid's purpose and function. Usually a design is called a *Grid architecture* (for more in-depth explanations on the software architecture concept, see Section 6.2).

2. **Hardware** As every computing infrastructure, a Grid depends on underlying hardware. Simply put, without computers and networks, you can not have a Grid!

3. **The middleware** The middleware is the crucial software component of the Grid. Without a middleware, Grid computing becomes impossible. Since Grid resources are different and disparate, there's the need for a software stack enabling enterprise application integration to create a coherent whole: that's the middleware. The middleware concept can be found also in operating systems software theory and in some kind of applications software.

2.2.1 Grids Architectural Layers

A Grid architecture is often described in terms of "layers", where each layer has a specific function. The layers that we consider are the usual Grid Layers as described in [51].

- The bottom layer is the *network layer*, which connects Grid resources. Networks are an essential piece of the Grid hardware. Networks link the different computers that form part of a Grid, allowing them to be handled as one huge computer.

- Next in line is the *resource layer*, through which actual Grid resources, such as computers, storage systems, electronic data catalogues, sensors and telescopes are connected to the network. The physical infrastructure of a Grid is often called the Grid"fabric".

- The third layer is the *middleware layer* which, as said, provides the tools that enable the various elements to participate in a Grid. The middleware layer may be composed of several layers itself, like a layer of *resource* and *connectivity* protocols, and a higher layer of *collective* services. Protocols used inside resource and connectivity layers handle all "Grid-specific" network transactions between different computers and Grid resources. The collective services use *information protocols*, which obtain information about the structure and state of the resources on the Grid, and *management protocols*, which negotiate access to resources in a uniform way.

- The top-layer is the *application layer*; this is the layer that Grid users "see" and interact with. Grid applications can follow in a wide variety, althogh right now Grid technologies are mailny used by large scale scientific applications known to be part of the vast *e-Science* world. e-Science is a discipline that envisages using high-end computing, storage, networking and Web technologies together to facilitate collaborative, data-intensive scientific research. This requires new paradigms in Internet computing that address issues such as multi-domain applications, co-operation and co-ordination of resource owners and blurring of system boundaries. Grid computing is one such paradigm.

These layers are shown in Figure 2.3, which some readers may remind from [38]. Some authors make further differentations among the layers, but for what concern this thesis it is enough to have this simple four-layers granularity.

In Section 4.3 we will refer to this kind of classifications when defining the different policy framework models for Grids. In that case we will use an even simpler three-layers granularity.

2.2.2 The OGF and OGSA

Key to the realization of this Grid vision is standardization. Ensuring fundamental Grid requirements like interoperability, security, robustness and scalability between heterogeneous systems can be achieved using a Web Service Architecture (WSA) [13], which is an incarnation of a Service Oriented Architecture (SOA) in the context of the World Wide Web. SOA is the leading architectural style of the newly developed Grid technologies. Therefore, in order to achieve cross-Grid interoperability, the scientific community defines and implements standard interfaces for common services in the light of

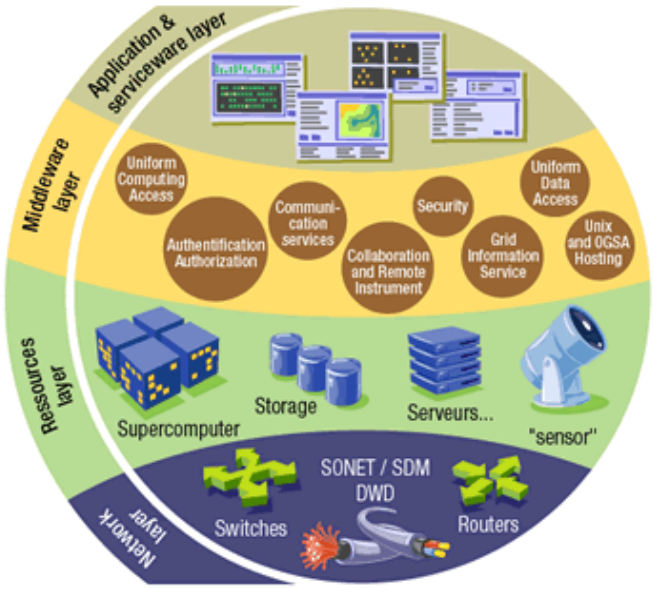

Figure 2.3: Grid layers [38]

a SOA context. Examples of such interfaces are the *Basic Execution Service (BES)* [30], that provides a Web Service (WS) interface for submitting jobs to computational resources, the *Data Access Integration Service (DAIS)* [5], and the *Resource Usage Service (RUS)* [2].

The Open Grid Forum (OGF) is a community of users, developers, and vendors leading the global standardization effort for Grid computing. The work of OGF is carried out through community-initiated working groups, which develop standards and specifications in cooperation with other leading standards organizations, software vendors, and users. The OGF produced the Open Grid Services Architecture (OGSA) [32], which describes an architecture for a service-oriented Grid environment for business and scientific use.

2.3 Grid Systems Taxonomy

After a dozen years' research on Grids, scientists are now capable to differentiate between different kind of Grids. According to [52], Grid systems can be placed into the categories shown in Figure 2.4, which are *Computational Grids*, *Data Grids* and *Service Grids*.

The *Computational Grid* category denotes systems that have higher aggregate computational capacity available for single applications than the capacity of any constituent machine in the system. These systems can be further subdivided into two more categories:

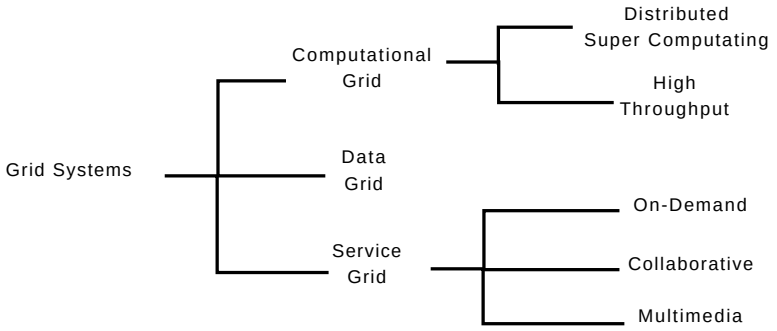

Figure 2.4: A Grid system taxonomy [105]

- a *distributed supercomputing* Grid executes the application in parallel on multiple machines to reduce the completion time of a job. Applications that require distributed supercomputing are grand challenge problems such as weather modeling.
- A *high throughput* Grid increases the completion rate of a stream of jobs and are well suited for 'parameter sweep' type applications such as the Monte Carlo simulations which are widely used by HEP (High Energy Physics) experiments.

The *Data Grid* category is for systems that provide an infrastructure for synthesizing new information from data repositories such as digital libraries or data warehouses that are distributed in a wide area network. Computational Grids also need to provide data services but the major difference between a Data Grid and a Computational Grid is the specialized infrastructure provided to applications for storage management and data access. Many Grid initiatives, like the EGEE[1] project, are working on developing large-scale data organization, catalog, management, and access technologies. Data Grid technologies are one the main subjects of this thesis. That's why a more in-deep view into data Grids can be found in Section 2.4.

The *service Grid* category is for systems that provide services that are not provided by any single machine. This category is further subdivided:

- a *collaborative Grid* connects users and applications into collaborative workgroups. These systems enable real time interactions between humans and applications via a virtual workspace.
- An *on-demand Grid* dynamically aggregates different resources to provide new services.
- A *multimedia Grid* provides an infrastructure for real-time multimedia applications.

[1] Enabling Grids for E-sciencE

Even if most ongoing research activities developing Grid systems fall into one of the above categories, it should be noted that these classifications are starting to become conceptual today, since many of the current Grid systems exhibit the functionalities of several categories. Just to make an example, the already cited EGEE project is the result of the combination of multiple past projects, and now encompasses views and functionalities from computational, data and service Grids.

2.4 Notions about Data Grids

In Section 2.2.1 we said that today many Grid applications follow in the vast e-Science world. Most of the times, these applications involve the production of large *datasets* (viz. data collections) from simulations or from large-scale experiments. Datasets are typically stored on mass storage systems, such as tape libraries or disk arrays. The datasets must be accessed by users in different locations. Users may create local copies or *replicas* of the datasets to reduce latencies involved in wide-area data transfers. A replica may be a complete or a partial copy of the original dataset. A replica management system or data replication mechanism allows users to create, register, and manage replicas, or to update new replica versions if the original datasets are modified. Some systems may also create replicas on its own. *Metadata*, or data about data, is information that describes the datasets. A metadata system could contain attributes such as name, time of creation, size on disk, and time of last modification. Metadata may also contain specific information such as details of the process that produced the data. A *replica catalog* contains information about locations of datasets and associated replicas and the metadata associated with these datasets. Users query the catalog using metadata attributes to conduct operations such as locating the nearest replica of a particular dataset.

These datasets have to be shared among large groups of researchers spread worldwide, and their analysis is highly compute-intensive requiring dedicated resources. Data Grids primarily deal with providing services and infrastructure for distributed data-intensive applications, and thus are particularly useful for running E-science applications.

Often cited examples for Data Grids are the ones being set up for analyzing the huge amounts of data that are being generated by the CMS (Compact Muon Solenoid), ATLAS (A Toroidal LHC ApparatuS), ALICE (A Large Ion Collider Experiment), and LHCb (LHC beauty) experiments at the Large Hadron Collider (LHC) [55] at CERN. These Data Grids will involve thousands of physicists spread over hundreds of institutions worldwide and will be replicating and analyzing terabytes of data daily.

Data Grids [105, 21] are an innovative technology taking advantage of existing computer science concepts in file systems, database systems and Grid

computing. A Data Grid provides services that help users discover, transfer, and manipulate large datasets stored in distributed repositories and create and manage copies of these datasets. As a minimum, a Data Grid provides two basic functionalities: a high-performance reliable data transfer mechanism and a scalable replica discovery and management mechanism. In order to enable researchers to derive maximum benefits out of the infrastructure, there are a set of functionalities that every Data Grid should be able to perform:

- ability to search through numerous available datasets for the required dataset;
- ability to discover suitable data resources for accessing the data and computational resources for performing analysis;
- ability to select suitable computational resources and process data on them;
- ability for resource owners to manage access permissions.

Thus, seamless organisation, well-defined architecture and intelligent resource allocation and scheduling are also required to ensure that users realise their utilities from the Data Grid infrastructure.

The explosion in popularity of Data Grids in scientific and commercial settings has led to a variety of systems offering solutions for dealing with distributed data-intensive applications. Unfortunately, this has also led to difficulty in evaluating these solutions because of the confusion in pinpointing their exact target areas. Also, there exist a few different mechanisms with similar properties for supporting a distributed data-intensive infrastructure.

In order to help ourselves with future reasonings on Data Grids security, following within this Section we are providing a Data Grid abstraction. Interested readers may find a good taxonomy for Data Grids in [105].

2.4.1 An abstraction of a Data Grid

Data Grids are distributed system which can contain a variety of data resources. These resources may use different data models to structure the data, different physical media to store it, different software systems to manage it, different schema to describe it, and different protocols and interfaces to access it. The data may be stored locally or remotely; may be unique or replicated; may be materialized or derived on demand. Different levels of virtualizations over these data resources should be provided. Virtualizations provide abstract views that hide these distinctions and allow the data resources to be manipulated without regard to their nature.

In a Data Grid there are two kinds of resources to be managed: *Grid Data* and *Grid storage space*:

- a *Grid Data* (GD) is any kind of data that can be located, transferred, replicated and manipulated: client services should be able to access dispersed GD, independently from its physical location, through a *Data Grid Management System* (DGMS) [66]. A DGMS is a software system used to manage Data Grids through the use of multiple abstraction mechanisms that hide the complexity of distributed data and heterogeneous resources. This naming capability allows users to refer to specific data resources in a physical storage system using a high level logical identifier.

- a *Grid storage space* (GSS) is a storage space shared between multiple VOs, and managed by a Grid *Storage Element* (SE). An SE (e.g. the Storage Resource Manager [42]) is an interface to mass storage systems, providing a uniform control interface and enabling the Grid to efficiently use the storage.

It's not necessary for a GD to be stored in GSS only, while a GSS may also contain data that can not be relocated, viz. are not GD.

DGMS implementations should follow the OGF recommendations for providing implementation guidelines and standards to implement GD location independence. Data resources have to be recognized by name without any location information. The Open Grid Services Architecture (OGSA) work on data architecture [6] identifies a scheme with the following three levels of naming:

- **Human-Oriented name** (HON): based on a naming scheme that is designed to be easily interpreted by humans, viz. human-readable and human-parsable. The HONs represent the key by which the users find the actual locations of their files. They are user friendly high-level identifiers. A DGMS could let the users organize them with a directory structure to simulate a global namespace. A same data resource could be addressed by various HONs by different users, similarly to the concept of alias.

- **Abstract name** (AN): a persistent name suitable for machine processing that does not necessarily contain location information. ANs are given to each data when it is managed by a DGMS. An AN is a unique identity to hide the data replication: a same AN can correspond to different replicas.

- **Address**: specifies the location of a data resource. An address provides an abstraction of the data namespace living into a storage resource to allow different data access paths. Each replica has its own address and it specifies implicitly which storage resource needs to be

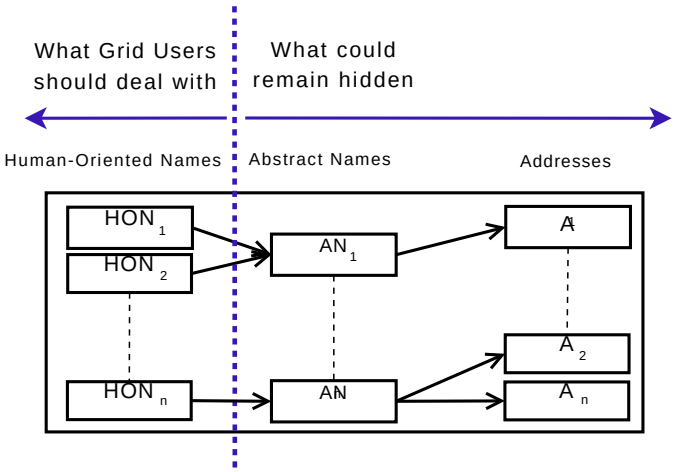

Figure 2.5: Data naming in Grid.

contacted to extract the data. Usually, users do not need to be directly exposed to addresses, but only to the logical namespace defined by HONs.

To provide the users the illusion of a single file system, a DGMS has to keep track of HONs to AN and Addresses mappings in a scalable manner. Figure 2.5 describes the relationship on terms. Within this Figure, we highlighted what Grid users should deal with, and what could remain hidden [2]. The HON level is usually organized as (virtual) distributed file systems. There's no real reason the Data Grid organization to be known to simple users, thus the AN and addresses levels does not need to be known in advance.

Anyway, as we said earlier, many implementations still let the users specificying the addresses they want to use when submitting jobs. This is the reason why we distinguish two kind of access. Figure 2.6 shows a simplified logical view of a Data Grid. The Figure shows the two kinds of data accesses:

1. clients (e.g. Grid users) may access a GD knowing just the HON by performing what we call a *Grid access*;

2. clients may access a GD and non-GD data directly accessing them on the SE when the address is known, thus performing what we call a *direct access*.

[2] Actually, some DGMS implementations let the users interrogate the Addresses directly. We personally believe that future implementations will leave this level transparent to simple users

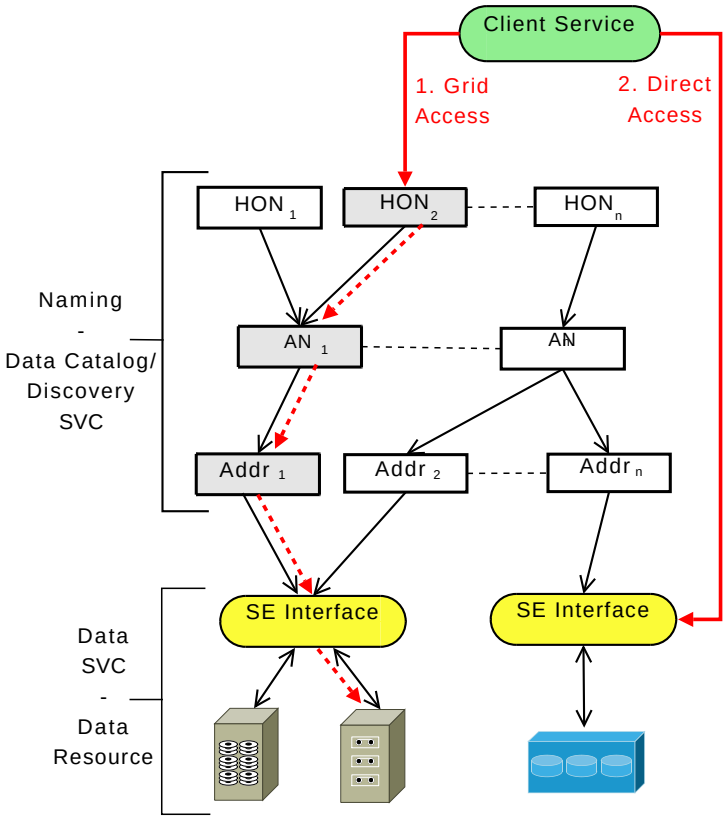

Figure 2.6: A logical view of a Data Grid

The simplicistics concepts we wrote within this Section have been expanded and implemented by Grid developers in many Data Grid projects. For example, in EGEE the most used DGMS is the LHC File Catalog (LFC) [67], where human oriented names are called logical file names (LFNs). The replicas (addresses) are identified by Site URLs (SURLs). Each replica has its own SURL, specifying implicitly which Storage Element needs to be contacted to extract the data. The SURL is a valid URL that can be used as an argument in an SRM [42] interface. Usually, users are not directly exposed to SURLs, but only to the logical namespace defined by LFNs. The Grid Catalogs provide the mappings needed for the services to actually locate the files. The illusion of a single file system is given to the users. To maintain this illusion, the Grid data management middleware has to keep track of SURL - LFN mappings in a scalable manner. Also, the identifier of a file entity has to be kept unique at all times. In order to achieve this, a Global Unique Identifier (GUID), which correspond to an abstract name, is given to each file when it is created on the Grid.

3

Notions about Computer Security

This Chapter introduces notions about Computer Security. The concepts we introduce here are somewhat general, but nonetheless useful for those readers who do not know much about the topic. They represents the cornerstones for reasoning of security for Grids.

Computer security is a branch of technology known as information security as applied to computers. General computer security includes diverse things, like controlling authorized (and unauthorized) computer usage, managing computer accounts and user privileges, copy protection, virus protection, software metering, database security, but also avoiding denial of service (DOS) attacks, or even software licensing.

In the early days of computer security, cryptography was viewed as a solution for all the computer security problems. Right now, there is a plethora of new computer security problems. This means, that, most of the time, you can't rely on just the mathematics given you by cryptography [89], which is still the base of security, but other techniques are need on top to solve other types of problems. This Chapter highlights some of them.

3.1 The CIA triad

A large amount of effort has gone into trying to define computer security. For over twenty years security practitioners has held that *confidentiality*, *integrity* and *availability* (known as the CIA Triad) are the core principles of information security.

- **Confidentiality** is not much more than privacy [89]. Confidentiality is the property of preventing disclosure of information to unauthorized individuals or systems. The bulk of computer security research has centered around confidentiality. Many people use confidentiality and security as synonyms. Breaches of confidentiality take many forms. Permitting someone to look over your shoulder at your computer screen while you have confidential data displayed on it could be a breach of confidentiality. If a laptop computer containing sensitive information about a company's employees is stolen or sold, it could

result in a breach of confidentiality. Giving out confidential information over the telephone is a breach of confidentiality if the caller is not authorized to have the information.

- **Integrity** is harder to precisely define. One good definition is: "every piece of data is as the last authorized modifier left it". Integrity is about the security of writing data. Integrity is violated when an employee (accidentally or with malicious intent) deletes important data files, when a computer virus infects a computer, when an employee is able to modify his own salary in a payroll database, when an unauthorized user vandalizes a web site, when someone is able to cast a very large number of votes in an online poll, and so on. But integrity could be violated without malicious intent, like mis-typying someone's address. Sometimes bulk updates to a database could alter data in an incorrect way, leaving the integrity of the data compromised. As it's obvious by reading these definitions, integrity is closely related to confidentiality. The latter is about unauthorized reading of data (and programs); the former is about unouthorized writing. Some security techniques want to achieve both goals.

- **Availability** is much broader than computer security. Availability has been defined as "the property that a product's services are accessible when needed and without undue delay," or "the property of being accessible and usable upon demand by an authorized entity." In the context of security, availability is about ensuring that an attacker can't prevent legitimate users from having reasonable access to their systems. High availability systems aim to remain available at all times, preventing service disruptions due to power outages, hardware failures, and system upgrades. Ensuring availability also involves preventing DoS attacks.

In 2002, Donn Parker proposed with [78] an alternative model for the classic CIA triad that he called the six atomic elements of information. The elements are *confidentiality, possession, integrity, authenticity, availability, and utility*. The merits of the Parkerian hexad are a subject of debate amongst security professionals. For what concern this thesis, we are happy with the classic CIA triad definition.

3.2 Access Control

Access control is the ability to permit or deny the access to a particular resource by a particular entity. In other words, we want to make sure that authorized people are able to do whatever they are authorized to do, and that everyone else is not. Access control *systems* want to assure that confidentiality, integrity and availability are maintained.

In a general sense, access control is a problem much bigger than just computers: access control mechanisms may be used in managing physical resources (such as a movie theater, to which only ticketholders should be admitted), logical resources (a bank account, with a limited number of people authorized to make a withdrawal), or digital resources (for example, a private text document on a computer, which only certain users should be able to read).

In all access control definitions, there is some *subject* that has to access to some *object*. The subject is often a user, but could also be a computer program or a process. The object is a file in a computer, or another computer program. More recently, an object may be a service of a Service Oriented Architecture (SoA) [13]. A same entity could be a subject in an access control relationship and an object in another.

Access control is a big deal for almost everyone. And it is difficult to do properly.

3.2.1 Authentication, authorization, accounting and auditing

In a very general sense, access control systems should include *authentication*, *authorization*, and *accounting* capabilities.

- **Authentication** deals with the verification of the identity of an entity within a network: it is the process of establishing the digital identity of one entity to another entity. Authentication is accomplished via the presentation of an identity and its corresponding credentials. Examples of types of credentials are passwords, one-time tokens or digital certificates.

- **Authorization** deals with the verification of an action that an entity can perform after authentication was performed successfully. It refers to the granting of specific types of **privileges** (including "no privilege"). Authorization may be based on restrictions, for example time-of-day restrictions, or physical location restrictions, or restrictions against multiple logins by the same user. Most of the time the granting of a privilege constitutes the ability to use a certain type of service.

- **Accounting** refers to the tracking of the consumption of network resources by users. In other words, accountability identifies what a subject (or all subjects associated with a user) did. Accountability uses such system components as audit trails (records) and logs to associate a subject with its actions. The information recorded should be sufficient to map the subject to a controlling user.

Sometimes, a fourth "A", which stands for *auditing*, transforms the AAA acronym in AAAA. *Auditing* is a manual or systematic measurable technical assessment of a system or application. All these concepts are well known and documented. Interested readers may found more notions in [107].

In the years, many *access control models* (ACM) have been proposed, and some of them implemented. These ACMs want to solve *confidentiality* and *integrity* requirements. ACMs don't usually take in consideration *authentication* and *accounting* problematics, but propose models for solving *authorization* problematics. We will make a brief comparison of ACMs in Section 5.

3.3 Usage Control

Usage control claims to provide a new intellectual foundation for access control. Usage control extends traditional access control by controlling data access as well as usage [77, 80].

Traditional access control models show limitations to cover modern digital environments. The main technical challenge stands in controlling not only data access, but data usage. Access control and trust management require enlargement of their scope to enable richer, finer, and persistent controls on digital objects regardless of their locations. Though Digital Rights Management (DRM) has opened up closed system restrictions, the discipline still lacks well-defined policies and models.

Controlling usage of sensitive information requires protection of digital information that may be critical to nations or organizations. Intelligence community and B2B (business to business) transactions are good examples for this purpose. Other relatively new goals are IPR (Intellectual Property Rights) protection or digital copyrights protection. Content providers' interest largely belongs here so they can realize maximum revenue. Privacy has been rarely studied in the context of controlling usage of digital information, but is beginning to get more public attention. The aim of usage control techniques is to cover all these purposes in a systematic way.

Until some years ago, while the fundamentals of access control appeared to be well understood, this was not the case for usage control. Some recent research helped understanding these fundamentals, and today there is quite a consensus upon the foundations of usage control mechanisms, even if this is still a widely researched topic.

This thesis is about usage control *systems*, as they can be applied to Grids and distributed systems alike. The readers will find many informations on usage control systems and techniques throughout the thesis.

From now on, we will not talk more about access control models and systems. We consider usage control as a better substitute for access control.

Usage control models and systems represent the new, and improved, access

control models and systems.

A very important result, which we will be used throughout the thesis, is represented by the UCON_{abc} usage control model [87]. A good explanation of it can be found in Section 5.2.

3.4 Security Frameworks

Security Frameworks address the application of security services in disparate environments, but we believe their utility is mainly for distributed systems. Security Frameworks provide protection for systems and objects within systems, and with the interactions between systems.

One of the first definitions for Security Frameworks came from an ITU-T technical report [47]. ITU-T recognizes the basic entities and functions involved in access (usage) control as the *initiator*, the *Access Control Enforcement function* (AEF), the *Access Control Decision Function* (ADF), and the *target*: initiators represent both the human beings and computer-based entities that use or attempt to use targets. Targets represent computer-based or communications entities to which usage is attempted or that are accessed by initiators. The AEF ensures that only allowable accesses, as determined by the ADF, are performed by the initiator on the target. When the initiator makes a request to perform a particular usage on the target, the AEF informs the ADF that a decision is required so that a determination can be made. In order to perform this decision, the ADF is provided with the usage request and some *Access Control Decision Information* (ADI), comprising the initiator ADI, the target ADI, the access request ADI and eventually some attributes helping the decision. The other inputs to the ADF are the access control policy rules. The decision is conveyed to the AEF which then either allows the access request to pass to the target or takes other appropriate actions. Figure 3.1 describes the relationship on terms, while Figure 3.2 describes the ADF interface.

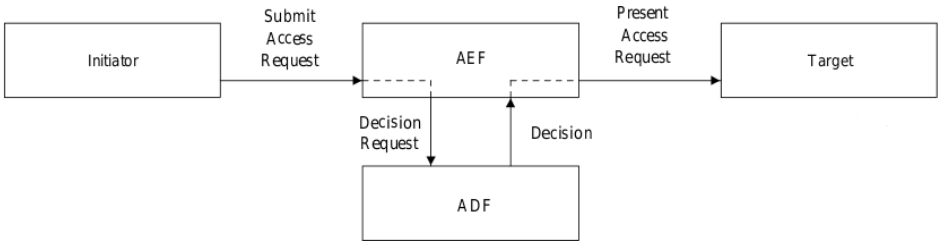

Figure 3.1: Fundamental Access Control Functions [47]

A concept that was just outlined by ITU-T is the policy concept. What happened is that recently the relationship between the Access Control Func-

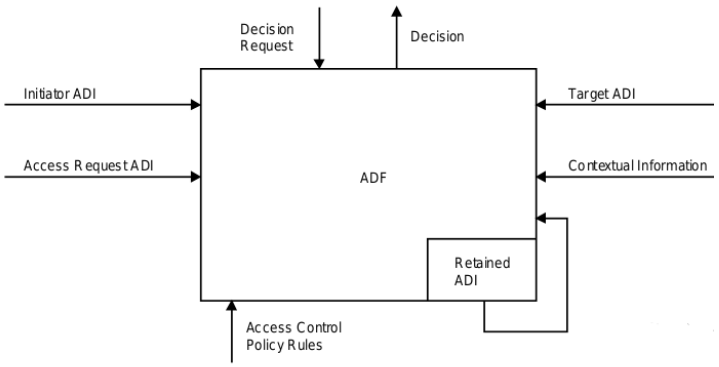

Figure 3.2: Access Decision Function [47]

tions of ITU-T has been implemented in the form of *security policies* and *policy points*.

We are not aware of commonly accepted definitions of security policies, although there exists an RFC [109] documenting policies and policy points terminology. Wheter such a definition exists or not, we may say thar a security policy is composed of a set rules that define how certain subjects may interact with certain objects. In literature, subjects are normally users, or processes acting behind a user, but subjects could also be groups of users, collective names, or services.

Security policy rules may specify that some subjects can, or cannot have an access to some objects. In literature terms like "positive authorization", "permission", "privilege" or "right" are used to indicate that a subject can have an access to an object. Instead, if a rule states that a subject cannot have an access to an object, it is called a "negative authorization", "prohibition", "negative permissions" or "denials". Another important type of rules are the "obligations", which states that a subject is obliged to perform one or more actions in order to gain a permission. Historically, the term "authorization" is used to describe equally a positive or a negative authorization, but within this thesis we will use this term in a narrow sense. Instead, we prefer the term "right". These concepts will become more clear in section 5.2.

For what concern the policy points, there are quite accepted definitions. Most of them have been popularized by the widespread acceptance behind the eXtensible Access Control Markup Lunguage (XACML) [74], whose data flow diagram is shown in Figure 3.3.

Policy Decision Point (PDP): the point where decisions about the policies are made. It evaluates *applicable policies* and renders *authorization decisions*. In a loosely coupled distributed environment a local to a resource (designated) PDP can call other PDPs requesting for evaluating

policy components related to their domain of authority to provide a final decision.

Policy Enforcement Point (PEP): the point where the policy decisions are actually enforced. This is the system entity that performs *access control*, by making *decision requests* and enforcing *authorization decisions*.

Policy Authority Point (PAP): the point that owns the authority over the PDPs. We should remind that sometimes PAP indicates the *Policy Administration Point*, which is the system entity that creates and administer the policies.

Policy Information Point (PIP): the system entity that act as a source of attribute values.

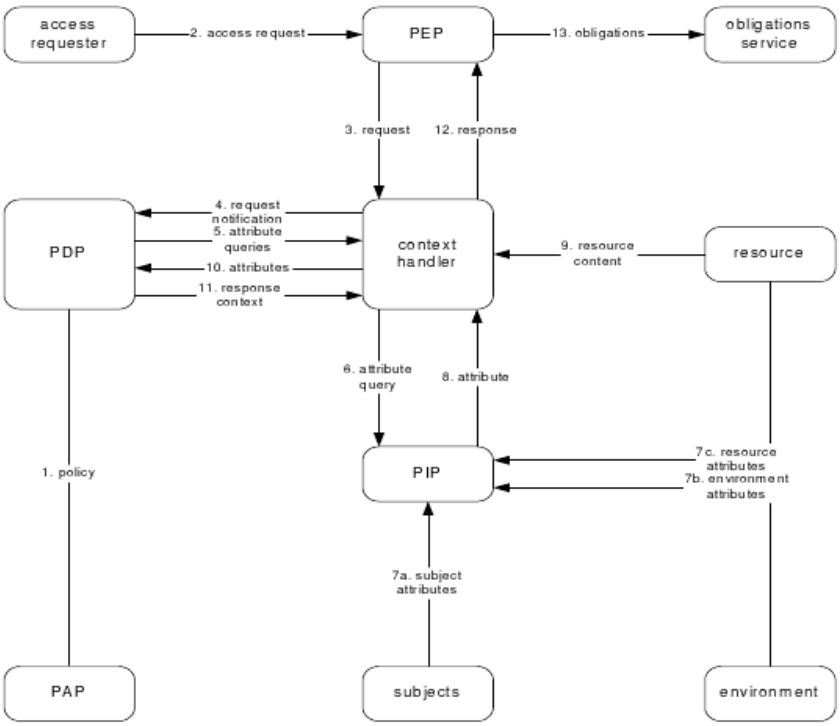

Figure 3.3: The XACML Data Flow Diagram [74]

The PDP-PEP interaction is the key for a good policy distribution. There are two possible basic implementations, the pull model and the push model. The pull model is the more used one, in which a supplicant first ask for the resource PEP to authorize himself, and then the PEP ask to an external PDP for the final decision. During the policy evaluation, the

PDP may also request specific user attributes from a Policy Information Point (PIP), or asking an authentication service for user identity confirmation. When the PDP identifies the applicable policy instance, it collects the required context information, evaluates the request against the policy, and communicate the decision back to the PEP. After receiving a PDP decision, the PEP conveys the service request to the resource, that may also have a locally determined policy implying additional restrictions on resource usage and/or access.

4

Notions About Usage Control For Grids

The goal of this Chapter is double. First of all, it introduces our general vision of security, and particularly usage control, for Grids. Secondly, shows OGF's view of Grid authorization, and some security tools used in today's production Grids.

Authentication, authorization and accounting (AAA) systems have been implemented and used in different Grid middlewares, but Grid systems in use today do not address usage controls in a systematic way. Grid communities spent a lot of time talking about security requirements. Unfortunately, until now not all the them are satisfied by any existing security infrastructure. In developing Grid security systems, some of them have not even been considered. Just to make an example, while there are many requirements concerning Grid authorization, historically in Globus [90] an authenticated user is an authorized user. This emphasizes the authentication aspect, but Grids need complex authorization mechanisms. With this thesis, one of our aim is to enable new Grid infrastructure developer to create more secure systems, capable to attract new Grid users and applications. With new security frameworks, we should be able to define "who can do what, when and where" whatever the requirements and the set of cases to satisfy. With this Section, we will look at existing security concepts for Grids, models and implementations. We will have a look in particular to existing access and usage control models for Data Grids.

We published a small portion of the work reported in this Chapter in [37].

4.1 Basic Grid usage control concepts

A Usage Control system is a security architecture, which can be defined as "a set of features and services that tackles a set of security requirements and can handle a set of cases". A Grid middleware should encompass such security architecture.

Within a Grid usage control architecture, we can distinguish two virtual black boxes: the *authentication* box and the *authorization* box:

- *authentication* deals with the verification of the identity of an entity within a network. An implementation should provide an agnostic plug point for multiple authentication mechanisms, and the means for conveying the specific mechanism used in any given authentication operation.

- *Authorization* deals with the verification of an action that an entity can perform after authentication was performed successfully. The goal of an authorization system is to provide a light-weight, configurable, and easily deployable policy-engine-chaining infrastructure that is agnostic to back-end enforcers and evaluators, as well as the run-time container infrastructure and the state model that hosts them. The framework allows for a combined and flexible decision making process, taking into account information, assertions and policies from a variety of authorities.

We can make a brief comparison between the high-level techniques besides authentication and authorization. The first link in the Grid security chain is authentication. Grid resources authenticate remote users using basically two ways: the first uses a session key, and the second, which is the mostly used too, uses the Public Key Interface (PKI). On the other hand, we need a Privilege Management Infrastructure (PMI): a PMI is to authorization what a PKI is to authentication [17]. Just to make an example, we can express some user's attributes using the X.509 Attribute Certificate (AC) [27], which maintains a strong binding between a user's name and its attributes. Certification Authorities (CAs) digitally sign a public key certificate; in a similar way, the entity that signs an AC is called an Attribute Authority (AA), while the root of trust of the PMI is called the Source of Authority (SOA), which may delegate its power to subordinate AAs. Like Certificate Revocation List (CRL), an AA could issue an Attribute Certificate Revocation List (ACRL) [45] to revoke privileges from an AC. Obviously, ACs is just one of the possible solutions to join users and their attributes.

We can roughly divide the process to reach access to a Grid resource in 3 levels: first of all there's a Grid authentication process, then authorization on a Grid-ID base, and finally local enforcement. In Figure 4.1 you can see the all-round security process. This Figure has been taken from [37].

4.1.1 Authentication

In a Grid environment the authentication model is normally based on the concept of *trusted third parties* (TTPs): the first link in the authentication chain is the certification authorities (CAs), which in practice are trust anchors for VOs. This model makes use of the Public Key Infrastructure (PKI) technology: CAs issue X.509 certificates [45], where essentially a unique

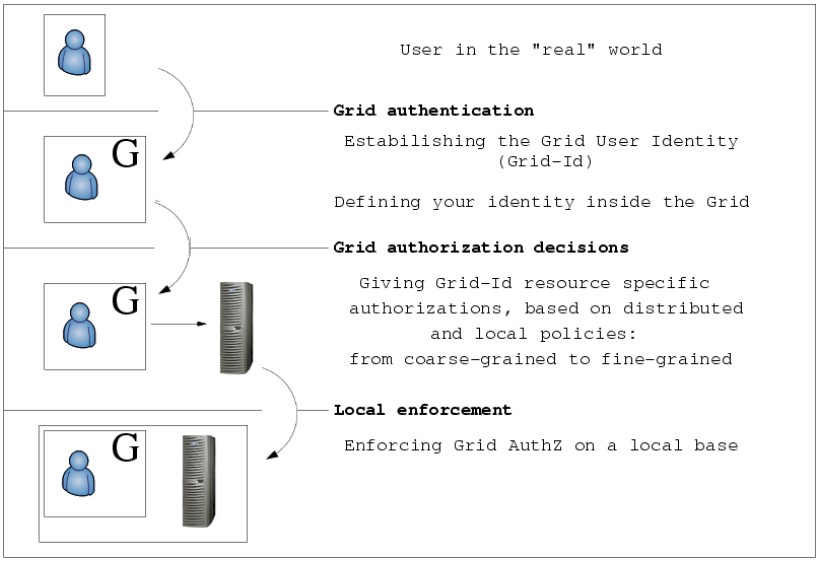

Figure 4.1: The Grid usage control process [37]

identity name and the public key of an entity are bound through the digital signature of that CA. It is possible that some Grid service may require further authentication and verification controls, but these issues are out of the scope of a Grid authentication service, because they suppose a specific contract between the user and the resource, outside the Grid security infrastructure. An authentication service must define distinctly the Grid identity of any user: this means that every user inside a Grid is given some credentials for her description and identification. With description we mean not only user's VOs, but his/her role inside every VO he is member of. We will refer to this kind of enhanced, extended, authentication as of *Grid Authentication*. A Grid Authentication is the first step for a Grid Usage Control process.

4.1.2 Authorization

Authentication frameworks should provide a coarse-grained granularity with some kind of credentials, like VO, groups and roles, or alternatively a semantic description of the Grid user. An authorization service should then make use of these information for fine-grained usage decisions, using a security policy framework that uses concepts like the ones we outlined in Section 3.4. We don't expand Grid authorization concepts here, since they represent one of the main topics of the thesis. Some existing authorization models are reviewed in the next Section.

4.2 Models for usage control on Data Grids

Doing a parallel with what is stated in [75], there are basically two ways of enforcing usage control in a Grid: the "Grid usage" and the "local usage".

4.2.1 The "Grid usage" model

With the "Grid usage" model, users can only access their data via Grid tools and services. What this means in practice is that a storage element would have a *service* `user-id`, which is assigned the all the data it stores. A second component runs under this identity and interacts with the user, acting as a server for the usage control needs of users, data, storage elements and security administrators. This server checks the user's permission from some kind of catalogs, probably using a policy server, and delivers the content from the storage element.

There is a set of motivations behind choosing this model. First of all, there is a uniform usage control semantics, identical at all Grid sites. The user should not have to know in advance about peculiarities of site usage control mechanisms: this is one of the pre-requisites for ubiquitous Grid computing.

A second motivation is in providing support for resources with weak usage control possibilities. Today very few Storage Elements provided by the Grid sites have ACL capabilities: by enforcing security through the Grid, these storages should acquire this capability indirectly. For example, the access control implementations of the SRM v1.1 specification does not include any explicit operation to manipulate permissions. Although the v2.1 interface adds some functions, only few implementations support or plan to support access control at the level of basic Unix permissions, and no implementation at this time supports POSIX [97] ACLs.

A third advantage is that there is no need to assign local users to every new user, i.e. all the user administration does not have to be repeated at every site at each SE.

With this "Grid usage" model, the data security is enforced by the Grid middleware. All the GD (Grid Data), even if are stored on a GSS (Grid Storage Space), are owned by the middleware system. The middleware access service should then be capable to access the files on the SE directly. This means that the only accepted data identifiers should be *Human-Oriented Names* and *Abstract Names*, and that there should not be the possibility for "back-door" access through any *address*, unless the SE provides ACL capabilities. The SE may be then set up to also accept the user's credentials directly by adding a line to the given file's ACL. This solution is impractital. Since many SRM interfaces does not yet properly support ACLs in a standard way, the "local knowledge" requested goes against the ubiquitous nature of the Grid. Obviously, the policy server may be distributed, as an

instance, at each SE. This way, the local administrator would retain full control over the local resources.

The main disadvantage behind this model is that data already existing in storage spaces that are going to be "Gridified" would need to go through a migration step while going "into" the Grid. This step would mean to synchronize the authorization information with the Grid catalogs and to change the ownership of these files. For new data this should not be an issue. However, if it is decided un-gridify the files, they would need to be migrated back to be owned by local users.

From the site administrator's point of view this model has the disadvantage of trusting a service affiliated with a VO. To overcome this problem this server could be deployed at each site and be run under the control of the site administrators.

4.2.2 The "local usage" model

Allowing local access in parallel to Grid access implies that the site implements mapping from Grid identities to local `user-ids`. This means that the enforcement of usage control permissions is responsibility of the local SE implementations. In this model if a Grid service has to act on the user's behalf, then it needs the user's credential to be delegated.

The main advantage behind this model is that the file access is completly controlled by the site admins, via their mapping mechanism and permission system of their SE. There is no need to consult any external service. A second advantage is that there is no need to migrate data from/to the Grid. Traditional GSS access is possible as before, since the resource owners control the system with or without the Grid the same way they did before.

This model has the disadvantage of potential inconsistencies in usage control settings, since data on the Data Grid is usually replicated on many different sites. In theory one could synchronize the usage control settings on replicated data in the whole distributed environment (the catalogs have all the necessary information), but there are a couple of problems in doing that: first of all, this synchronization would create a large amount of network traffic, and most importantly, the differences in the ACL implementations of the SEs represent a hard mountain to climb. Not mentioning the poor adaptability of the whole system. In summary, security is only as good as the local SRM can enforce it, and if data is replicated to a less secure storage elsewhere, it may be more readily compromised than originally foreseen by the user.

Within the "local usage" model, every Grid user has to be known to the SE and has to have an associated account: local user management has to be synchronized with Grid user management. This adds to the administration burden for sites who cannot do automatic user mapping updates.

Since the local resource authenticates every user individually, every level

of name mapping is accepted as file identifyers, thus not only *Human-Oriented Names* and *abstract names*, but also *addresses*.

4.3 Policy framework models for Data Grids

We now look at how the authorization problems explained in Section 4.1 can be implemented using the policy-based security frameworks explained in Section 3.4. To do so, we refer to the classification of [34].

The models are distinguished by the layering of the two basic policy points, the PDP and the PEP. The layers that we consider are three of the usual Grid Layers as described in [51], and in Section 2.2. The top-layer is the Application Layer. The bottom layer is the Resource Layer and in-between we have the Middleware Layer.

When speaking of Grid Data (GD) and Grid storage space (GSS) management and usage (see Section 2.4.1), the PDP is the logical entity that stores the authorization information for the GD and GSS, and the PEP is the actual data access service that will enforce the decision of the PDP. Of course there needs to be a strong trust relationship between PEPs and PDPs entities, i.e. the enforcement has to trust the decision maker. Policy frameworks are, at least in a logical sense, part of Data Grid Management Systems (DMGS, see Section 2.4.1), which, as said, in the OGSA work on data architecture, implements a three-level naming for GD. Thus, the level the PDP is in terms of data identification and data naming influences the subject of the authorization policies handled by the PDPs.

For the matter of the discussion, both PDP and PEP are considered as virtual entities: they could be realized as a single central instance or with a set of distributed instances. In the latter case, there is the need for synchronization across the instances.

In [34], Frohner distinguishes between six models, which are illustated in Figure 4.2. The differences between this models is only in which layer we put the policy decision and enforcement points. However, these have profound implications for the trust relationships between VOs and sites, and on the way these models are implemented. All these models have pros and cons, but we will not highlight them here, since interested readers can find these information in [34].

- In this first model, both the PDP and the PEP are on the resource layer. In this model, the SE available on the Grid sites is responsible for deciding and enforcing the authorization of the user. To do this, implementations have to use the local ACLs provided by the storage services (e.g. SRM implementations). This model can easily implements the “Local usage” model of Section 4.2.2.

- In the second model, the PDP is provided by the middleware and

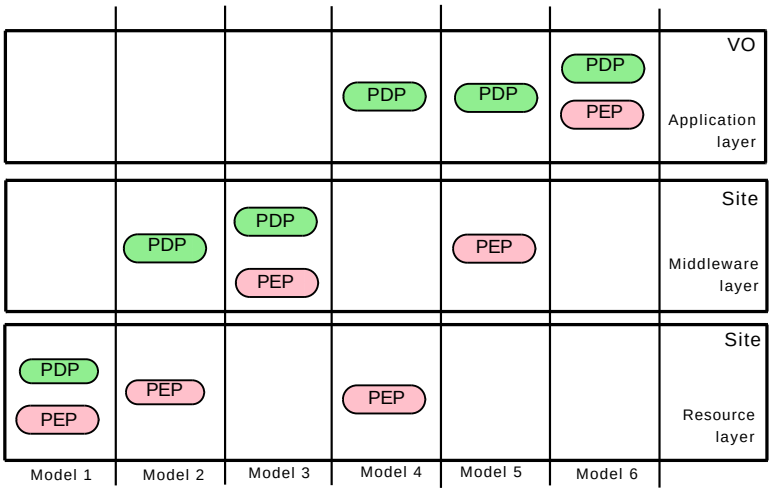

Figure 4.2: Security models for data management (inspired from [34])

not the resource. This means that there is a strong trust relationship between the resource and the middleware component that stores the authorization information.

- In the third model, both PDP and PEP are placed in the middleware layer. This means that all data access and usage has to go through the Grid middleware service. Thus, this model implements the "Grid usage" model of Section 4.2.1. If the middleware layer has both the PDP and PEP, the security semantics can be completely decoupled from the underlying data storage implementation. The middleware can then enforce any kind of semantics on top of the storage, even richer ones that the data store on the current resource can actually provide.

- In the fourth and fifth models the PDP is a VO service and is outside of the sites control. Model 4 places the PEP on the resource layer and model 5 in the middleware layer. This model has the advantage that the PDP in the application layer may be a single central instance, so the synchronization of authorization information between sites may not be an issue anymore.

- In the sixth model, both the PDP and PEP are in the application layer. This model has the downside that sites lose control over who is actually able to use their resources.

Within this thesis, we develop our own policy framework model, which slighlty differs from the ones we saw in this Section. A comparison between this models and the new one can be found in Section 7.2.2.

4.4 OGSA Recommandations

The aim of the OGF's OGSA authorization working group[1] is to define the specifications needed to allow for interoperability and pluggability of usage control components from multiple Grid authentication and authorization domains in the OGSA framework. The group leverages security work that is ongoing in the Web Service (WS) community (e.g. SAML [16], XACML [74], and the WS-Security [70] set of specifications) and defines profiles on how these should be used by Grid services.

The group provides an information document reviewing the functional components of Grid service provider authorization service middleware [18]. In the OGSA work, great attention is put on *credentials*, defined as attribute assertions digitally signed by the issuer (i.e. a security token) so that it can be cryptographically validated. Credentials can be issued by the *Credential Issuing Services* (CISs) of an *Identity Provider* (IP) or an *Attribute Authority* (AA) (e.g. the Virtual Organization Membership Service (VOMS) [3], see Section 8.1). The credentials can be embedded in an Attribute Certificate extension [27], and/or in a proxy certificate [98], or using a SAML token. Credentials can then be validated by a *Credential Validation Service* (CVS), that return the valid attributes of the subject. This first phase defines the Grid identity of every Grid user, and it's the first step of a Grid usage control process. Therefore, it corresponds to the extended "Grid Authentication" we discussed in Section 4.1.1.

Many of the other functional components follow the same terminology we wrote in Section 3.4. A *Policy Decision Point* (PDP) is the component responsible for returning an authorization decision given the user's access request and the user's valid attributes. The *Policy Enforcement Point* (PEP) enforces the results returned from a policy engine (normally a PDP). The *Context Handler* (CH) is responsible for handling the communications between PEPs, CVSs and PDPs. This second phase it's the second step of a Grid usage control process. Therefore, it corresponds to the "Grid Authorization" we discussed in Section 4.1.2.

The interactions between the functional components defined by OGSA can be constructed in four different ways, according to whether the credentials and the authorization decisions are pulled or pushed. For example, Figure 4.3 shows the case where an access requestor (a Grid User) pushes his/her credentials to a PEP. Then, after the CH obtained valid attributes from the CVS, a PDP is interrogated for an authorization decision, which in the end is returned to the PEP.

The case showed in Figure 4.3 is probably the most used and appealing one. The other three cases are shown in Figures 4.4 4.5 4.6 for completeness, but we're not going to work with them in the framework of this thesis.

[1] https://forge.gridforum.org/sf/projects/ogsa-authz

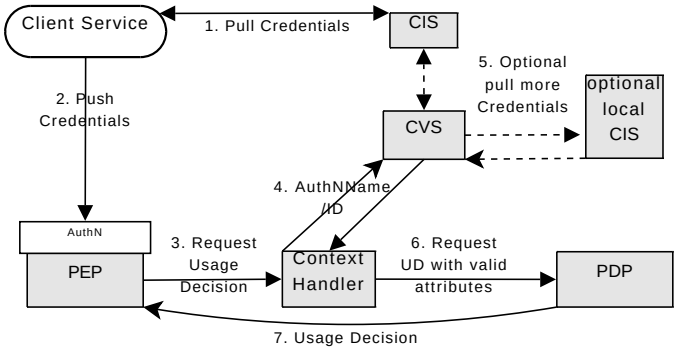

Figure 4.3: OGSA functional components. The credentials are pushed to the PEP.

Examples of all 4 modes of operation are already implemented.

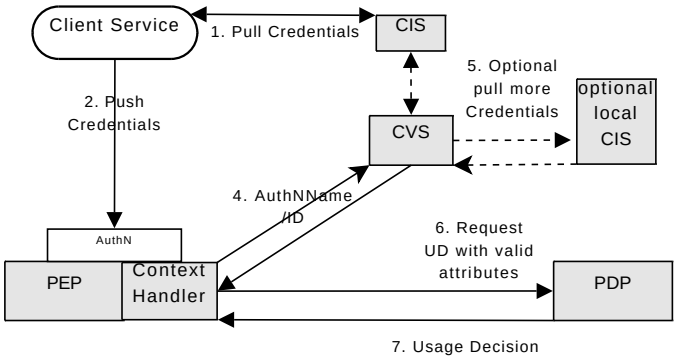

Figure 4.4: OGSA functional components. The credentials are pushed to the PEP. The CH is not separate from the PEP.

OGSA recommandations have to be taken seriously in consideration when building a security architecture for Grids and Data Grids. Within this thesis, we will propose in Section 9.1 a concrete architecture capable to handle a complete usage control process, as it explained in Section 4.1, that uses the functional components endorsed by OGSA and explained with the current Section. Moreover, the OGSA recommandations on the use of standards have been considered as a requirement in the re-engineering of a "Grid Authentication" service, which will be explained in Chapter 8.

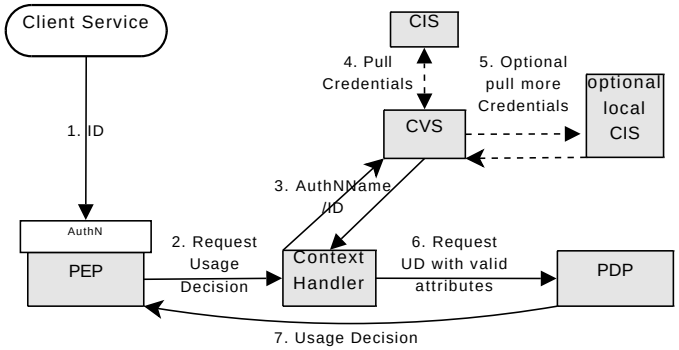

Figure 4.5: OGSA functional components. The credentials are pushed to the PEP.

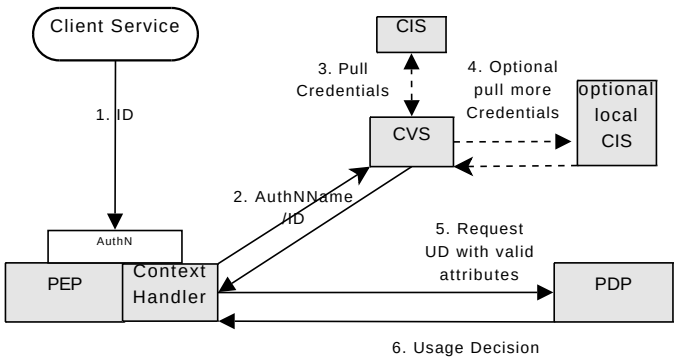

Figure 4.6: OGSA functional components. The credentials are pushed to the PEP. The CH is not separate from the PEP.

Part II

Working Instruments

5
Security Policy Models

This Chapter introduces existing and surveyed policy models in the literature, and in particular it focusses on the UCON_{abc} usage control model. UCON is a well-known usage control model we adopted as one of the cornerstone of our work. Its full understing is then necessary for all that follows.

5.1 Categorization of Policy Models

There is a great number of existing security policy models in the literature. Some address the same issues, some address different ones. For instance some models are dedicated to activity sequence control, others to the users structuring. Within this work, we don't want to review the whole bunch of security policy models that can be found in literature. A good review can be found in [64], where, for documentation reasons, they make a distinction between four families of security policy models: the *access control models*, the *flow control models*, the *administration models* and the *usage control models*. Within this work, we mainly focus on *usage control models*.

Access control models [12] enable us to specify which actions the subjects are allowed to carry out on which objects, thus protecting resources and services from unauthorized access. There is a large number of Access Control Models developed in the literature. Examples of the most historically important ones are the Discretionary Access Control (DAC) [49] and the widely used and expanded Role Based Access Control (RBAC) [88].

Flow control models [63] aim at providing an efficient response to one of the main problems of access control models: in access control, if programs more precisely processes are considered as subjects, then a malicious process might illegally transmit some unauthorized data. Therefore in flow control models, the objective is to control the access to data i.e. the contents by controlling the information flow.

Administration models can be used for managing a large scale information system that can usually not be done by a single system security officer (SSO). In good administration models, the security policy must evolve in order to always match the information system security requirements. For these reasons, appropriate administrative procedures are designed in order

to state which users are allowed, among other tasks, to add, modify or delete authorizations. A set of such procedures can be viewed as a meta-policy. Examples of Administration models are the Mandatory Access Control (MAC) [29], the The ARBAC97 model for role-based administration of roles [86], and the Organization-based Access Control (OrBAC) [64].

Usage control models [87, 80] are the result of recent research works that started in the field of digital right management (DRM), but is infact comprehensive enough to encompass traditional access control, trust management, and DRM. Usage control models like UCON$_{abc}$ [77, 111] unifies these areas systematically in a single framework and goes beyond its original scope. Even if usage control approach has ancestors in the access control frameworks, it comes out as a different and new approach. First of all, while traditional access controls utilize only authorizations for decision process, usage control methods generalize access control and cover authorizations, obligations, conditions, continuity (ongoing controls), and mutability. In access control, some permissions are granted to subjects to access "static" objects, in other words, objects that are usually stored within the subjects organization. Usage control is based on a different paradigm in which objects no longer stand in a computer system but are also shared or sent through Internet to private computers or PDAs, MP3 players, etc., owned by numerous and unknown clients.

Within the framework of the thesis, we focus on usage control models, and adapt them for the case of distributed systems with multiple SSO. We assume that programs, processes and services are trustful.

5.2 The UCON$_{abc}$ Usage Control Model

The **UCON$_{abc}$ usage control** model is a recent framework defined by Park and Sandhu [77, 87] for the specification of usage control policies. The main novelty of the UCON model lies in the fact that subjects and objects may have attributes that are mutable, thereby facilitating the continuity of the decision making and policy enforcement processes. Additionally, while decisions in standard access control models are based on policy *authorizations* only, the UCON model introduces two other decision factors, namely *obligations* and *conditions*. All of these features render the UCON model attractive for specifying security policies in distributed systems and Data Grids alike, especially considering the plethora of various security needs coming from the different Data Grid applications. Next, we describe the elements of the UCON model, first in a conceptual and informal way, and subsequently in a more formal one. The material provided in this Section has been mostly taken from [77] and [111].

5.2.1 UCON informal specification

The UCON model is made up of several components:

- *Subjects and Objects*: A subject is an entity that holds or exercises certain rights on objects, i.e. that executes access operations on objects. Both subjects and objects are defined and represented by their attributes. A subject usually represents an individual human being, but in UCON it may or may not have a unique identity. If authorization is done with a user's unique identity, accountability can be provided. If not, anonymity can be supported. An object, instead, is an entity that is accessed by subjects through access operations.

- *Rights*: Rights are the privileges that subjects can exercise on objects. Traditional access control systems view rights as static concepts, for instance access matrices, which do not change over time or have a slow rate of change. Instead, UCON determines the existence of a right dynamically, whenever a subject attempts to access and exercise a right on some object. Hence, if the same subject accesses the same object several times, the UCON policy could grant the subject different access rights each time based on changing attributes of the subject and/or the object. Even mutations is the system and environment the subjects and objects "live", can affect the decision process and therefore the right. The *usage decision functions* indicated in Figure 5.1 make this determination based on subject attributes, object attributes, authorizations, obligations, and conditions predicates. In general, rights include rights for direct use of objects (such as read), delegation of rights, and rights for administering access (such as modify subject and object attributes that in turn determine access rights). In this thesis, we do not consider delegation rights and administrative rights. Rights can be divided into many functional categories. The DRM community usually classifies rights as being in one of the two most fundamental rights categories: *view* and *modify*, sometimes augmented with *creation* and *deletion*. Another similar classification distinguishes between *create, read, update* and *delete*, also known as CRUD.

- *Attributes*: Both subjects and objects have attributes. Subject and object attributes are properties or capabilities that can be used for the usage decision process. Examples of subject attributes include identities, group names, roles, memberships, security clearance, and so on. Examples of object attributes are security labels, ownerships or classes. The general concept of attribute-based access control is commonplace in the access control literature and as such this aspect of UCON$_{abc}$ builds upon familiar concepts. UCON attributes can be

mutable, i.e. they can change over time, or *immutable*, i.e. they are constant over time. An example of a mutable attribute is the number of times that a subject accesses an object, whereas an immutable is a subject's or an object's identity.

- *Authorizations, Obligations and Conditions Predicates*: Predicates are logical statements about the subjects' and objects' attributes and the requested right. Predicates can be either *authorization*, *obligation* or *condition* predicates or any combination of these. Authorization predicates express a set rules that determine whether to grant the requested right or not. The authorization predicate could exploit both attributes of the subject and of the object. Authorizations can be either pre-authorizations (*preA*) or ongoing-authorizations (*onA*). *preA* is performed before a requested right is exercised and *onA* is performed while the right is exercised. *onA* may be performed continuously or periodically during the time span of access. In general, most traditional access control policies including MAC, DAC, RBAC, and Trust Management (TM) utilize some form of pre-authorization for their decisions. Obligations are UCON decision factors that are used to verify whether the subject has satisfied some mandatory requirements. Similarly for the case of Authorizations, we distinguish between *preB*, i.e obligations to be satisfied before performing an action, and *onB* obligations to be continuously satisfied while performing the action. Usually, Obligations refer to future requirements that must be obeyed. Traditional access control has hardly recognized the obligation concept. Finally, conditions are environmental or system-oriented decision factors, i.e. dynamic factors that do not depend on subjects or objects. Conditions are evaluated at runtime when the subject attempts to perform the access. A condition can be evaluated before (*preC*) or during an action (*onC*). Unlike authorizations or obligations, condition variables cannot be mutable, since conditions are not under direct control of individual subjects. Certain authorizations and obligations may require updates on subject attributes and/or object attributes. These updates can be either pre, ongoing, or post. Instead, evaluation of conditions cannot update any subject or object attributes.

The interactions between the described components are shown in Figure 5.1.

UCON_{abc} is actually a family of models with several parameters. In [77], Park and Sandhu introduce a classification based on the following three criteria: *decision factors* that consist of authorizations, obligations, and conditions, *continuity of decision*, being either pre or ongoing with respect to the access in question, and *mutability* that can allow updates on subject

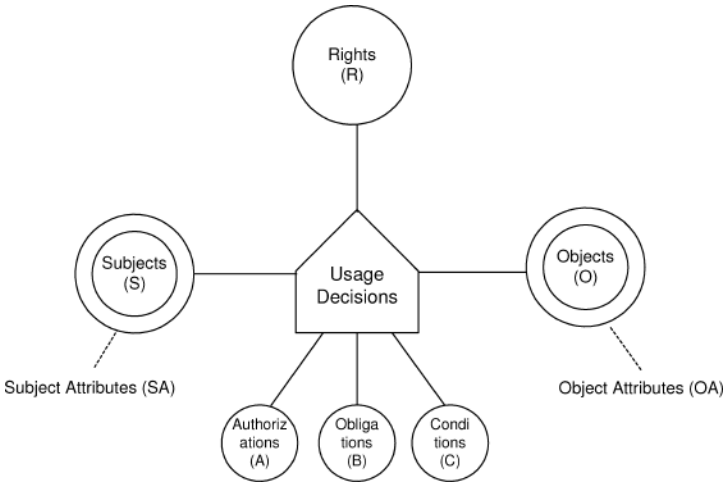

Figure 5.1: The UCON$_{abc}$ model components [77]

or object attributes at different times. If all attributes are immutable, no updates are possible as a consequence of the decision process. This case is denoted as '0.' With mutable attributes, updates are possible before (pre), during (ongoing), or after (post) the right is exercised, denoted as '1, 2, and 3,' respectively. Based on these criteria, Park and Sandhu enumerate the model space shown in Figure 5.1. Cases that are not likely to be useful in practice are marked as 'N.'

	0 (immutable)	1 (pre-update)	2 (ongoing-update)	3 (post-update)
preA	Y	Y	N	Y
onA	Y	Y	Y	Y
preB	Y	Y	N	Y
onA	Y	Y	Y	Y
preC	Y	N	N	N
onC	Y	N	N	N

Table 5.1: The UCON$_{abc}$ core models [77]

All of the UCON$_{abc}$ core models makes for a fairly complex model. From a model point of view, the resulting complexity is appropriate for modern cyberspace. We will make some considerations regarding this complexity in Section 9.1.1.

UCON$_{abc}$ is a very powerful usage control model, that is capable to embrace a vast number of previously studied and deployed security models. We now show its capabilities using some examples:

- *UCON$_{preA}$ models*: authorizations have been considered as the core of access control and extensively discussed since the beginning of access control discipline. Traditionally, access control research has focused on pre-authorizations in which a usage decision is made before a requested right is exercised. UCON$_{preA}$ models utilize these pre-authorizations for their usage decision processes. Traditional access controls such as MAC, DAC, and RBAC are likely to belong to UCON$_{preA_0}$. With attribute mutability, we introduce *update(ATT(s/o)* actions. Within UCON$_{preA_1}$, the meaning of *preUpdate(ATT(s))* is that subject attributes are updated before the access request is evaluated. UCON$_{preA_3}$ introduce *postUpdate* procedures to modify subject and object attributes after the access is ended. Exactly what values can be used in computing the update is left unspecified in the model. These could be subject or object attributes, and other variables. For example, certain authorization processes of DRM (e.g., membership-based digital library) can be expressed with policies pertaining to the UCON$_{preA_1}$ and UCON$_{preA_3}$ core models.

- *UCON$_{onA}$ models*: In an UCON$_{onA}$ model, usage requests are allowed without any 'pre' decision-making. However, authorization decisions are made continuously or repeatedly while usage rights are exercised. If certain requirements become dis-satisfied, the currently allowed usage right is revoked and its exercise is stopped. Ongoing-authorizations have been seldom discussed in access control literature. By utilizing ongoing-authorizations, monitoring is actively involved in usage decisions while a requested right is exercised. This kind of continuous control is especially useful for relatively long-lived usage rights. In UCON$_{onA}$, Park and Sandhu develop four detailed models. UCON$_{onA_0}$ is immutable ongoing-authorization model that has no update procedure included. UCON$_{onA_1}$ is ongoing-authorization model with pre-updates. UCON$_{onA_2}$ and UCON$_{onA_3}$ include ongoing updates and post updates, respectively. In many cases, ongoing-authorizations are likely to occur together with pre-authorizations. For example, suppose that an onA policy screens certain certificate revocation lists periodically to check whether the users identity certificate is revoked or not. While this is a case of ongoing authorizations, this makes sense only when the certificate has already been evaluated at the time of the request. This can be an example of UCON$_{onA_0}$ model. UCON$_{onA_1}$, UCON$_{onA_2}$, and UCON$_{onA_3}$ add pre-update, ongoing-update, and post-update procedures, respectively. With UCON$_{onA}$ models, we can express revocation policies using the usage start time (with an UCON$_{onA_{13}}$ policy), longest idle time (with an UCON$_{onA_{123}}$ policy), and so on.

- *UCON$_{preB}$ models*: UCON$_{preB}$ introduces pre-obligations that have to be fulfilled at the time of a request and before access is allowed. preB is a kind of history function that checks whether certain obligations have been fulfilled or not and return true or false for the usage decision. Suppose a user has to provide his name and email address to download a company's white papers, or suppose a user has to click 'Accept' on a license agreement to access a web portal. Here, the user has to fulfill the required actions before access is allowed. UCON$_{preB}$ models consist of 2 steps. First step is to select required obligation elements for the requested usage. This selection may utilize subject and/or object attributes. Second step is to evaluate whether the selected obligation elements have been fulfilled without any error (e.g., invalid e-mail addresses). In UCON$_{preB}$ models, a request may require multiple pre-obligation elements to be fulfilled.

- *UCON$_{onB}$ models*: UCON$_{onB}$ models are similar to UCON$_{preB}$ models except that obligations have to be fulfilled while rights are exercised. Ongoing-obligations may have to be fulfilled periodically or continuously. An example for the need of a UCON$_{onB}$ policy is a free Internet service provider requiring users to watch an advertisement while they are connected to the server. As long as the advertisement window is active, the usage is allowed.

- *UCON$_{preC}$ models*: As described earlier, conditions define certain environmental restrictions that have to be satisfied for usages. Unlike authorization and obligation models, condition models cannot be mutable. UCON$_{preC}$ introduces pre-conditions predicate that has to be evaluated before requested rights are exercised. Checking CPU-id or an IP address before a usage allowance is an example of UCON$_{preC_0}$.

- *UCON$_{onC}$ models*: The UCON$_{onC_0}$ model introduces an ongoing-conditions predicate (onC) for monitoring selected condition elements.

5.2.2 UCON formal specification

When speaking of security, it's hard to be precise in Engligh, Chinese, or whatever common language in use. Security practitioners and security engineers should try to lower the number of mistakes that can be made. A precise specification is the key to avoid a number of errors. That's why mathematics should be the language of choice when speaking about security.

> *Mathematics is nature's way of letting you know how sloppy your writing is.*
>
> Leslie Lamport

Anyhow, the mathematics written by most mathematicians and scientists isn't precise enough. More formal mathematicians, like the logicians, have developed ways of eliminating these imprecisions.

> *Formal mathematics is nature's way of letting you know how sloppy your mathematics is.*
>
> Leslie Lamport

There are many examples of logic languages in literature, but the ones most used in computer science are the *Lamport's temporal logic of actions* (TLA) [53] and the Linear Temporal Logic (LTL) [101].

A logical specification provides a tool to precisely define policies for system designers and administrators.

In [111], Park and Sandhu propose a formalization of the UCON model with a temporal logic, while in their previous works the UCON model was informal and conceptual. The formalization gives the precise meaning of the new features of UCON, such as the mutability of attributes and the continuity of usage control decisions. To build the logic model and formal specification, Park and Sandhu use an extended form of Lamport's Temporal Logic of Actions (TLA) [53]. The basic components include predicates between subject, object, and system attributes, as well as actions performed by the system or subjects. A usage control policy is a logic formula built from these components that has to be satisfied by a UCON model. This formalization is important for two reasons: first of all, it's a precise formulation of concepts that should not be ambiguous. Second, it gives us the possibility to use formal software requirement engineering methods. This last point will be clear after reading Sections 6.4 and 10.2. Within this thesis, we consider the UCON formalization as an important building block for our work. Anyhow, we're not going to make a complete review of the Park and Sandhu's work. Rather, we stress some points which we consider most important.

There are two types of actions in UCON: usage control actions and obligation actions. Figure 5.2 illustrates the different usage control actions that subjects and systems can perform in the UCON model [111]. These actions relate to the different phases of an object's usage: the subject's attempt to access an object, the system's decision regarding such attempt and the subject/object attributes updates performed by the system.

Given that the triple `(s,o,r)` represents the subject `s` requesting the right `r` for accessing the object `o`, Park and Sandhu consider the following set of actions:

- `tryaccess(s,o,r)`: performed by a subject `s` when performing a new access request `(s,o,r)`.
- `permitaccess(s,o,r)`: performed by the system when granting the access request `(s,o,r)`.

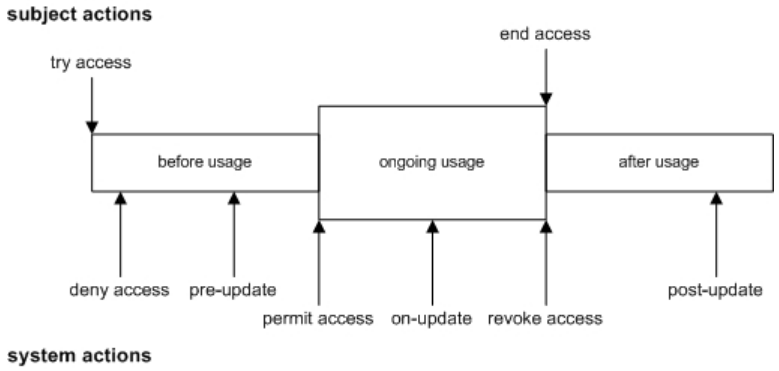

Figure 5.2: The UCON actions model [111]

- `denyaccess(s,o,r)`: performed by the system when rejecting the access request `(s,o,r)`.
- `revokeaccess(s,o,r)`: performed by the system when revoking an ongoing access `(s,o,r)`.
- `endaccess(s,o,r)`: performed by a subject `s` when ending an access `(s,o,r)`.
- `update(s,o,r)`: performed by the system to update a subject or an object attribute when performing an access request `(s,o,r)`.

Obligation actions must be performed by a subject before or during an access. For an access `(s,o,r)`, an obligation is an action described by `ob(`s_b`,`o_b`)`, where `ob` is the obligation action name and `ob(`s_b`)` and `ob(`o_b`)` are the obligation subject and object respectively. Note that in an obligation action, predicates are not used for control decisions, but for identifying what obligations are required.

The logical model of UCON is a 5-tuple: $\mathcal{M} = (S, P_a, P_c, A_a, A_b)$ where

- S is a set of sequences of system states;
- P_a is a finite set of authorization predicates built from the attributes of subjects and objects;
- P_c is a finite set of condition predicates built from the system attributes;
- A_a is a finite set of usage control actions;
- A_b is a finite set of obligation actions.

[111] provides logical specifications of the authorization, obligation and condition core models presented in Figure 5.1. We don't replicate all of

them here. A general preassumption is that all predicates and actions are computable, e.g., a predicate is a computable function of attribute values. As a matter of example, the usage control policy for the UCON_{preA_0} family of core models is:

$$permitaccess(s, o, r) \rightarrow \blacklozenge\big(tryaccess(s, o, r) \wedge (p_1 \wedge \cdots \wedge p_n)\big)$$

while the usage control policy for the UCON_{preA_2} family of core models is:

$$permitaccess(s, o, r) \rightarrow \blacklozenge\big(tryaccess(s, o, r) \wedge (p_1 \wedge \cdots \wedge p_n)\big)$$

$$permitaccess(s, o, r) \rightarrow \Diamond\big(onupdate(attribute) \wedge \Diamond endaccess(s, o, r)\big)$$

and the usage control policy for the UCON_{onB_3} family of core models is:

$$\Box\Big(\neg(\bigwedge\nolimits_i (p_{i1} \wedge \cdots \wedge p_{ik} \rightarrow ob_i)) \wedge \big(state(s, o, r) = accessing\big) \rightarrow revokeaccess(s, o, r)\Big)$$

$$endaccess(s, o, r) \rightarrow \Diamond postupdate(attribute)$$

$$revokeaccess(s, o, r) \rightarrow \Diamond postupdate(attribute)$$

These logical specifications provide a good degree of precision, but, as the reader will see in chapter 10.2, they are not able to encode all the information necessary for a really precise definition of an UCON policy. For this reason, we found these definitions helpful for understanding UCON policies, but not of real usefulness when applying formal software engineering techniques. What we're saying is that there is a lack of preciseness for what concern the exact sequence of actions needed for enforcing some types of UCON policies.

It should be noted that all the UCON authorization policies are defined for positive permissions. For an access request, if there is no policy to enable the permission according to the attribute values, then the access is denied by default. This is sometimes called the closed system assumption, whereby no policy is specified to deny an access in a system. The same holds for obligation and condition core models.

5.2.3 An UCON policy example

We now show an example of an UCON policy, that can be applied by a Data Grid application, that show the expressive power of UCON, together with its complexity. In order to specify our policy example, we include the following types:

- VO, set of Virtual Organisation (VO) identification names;

- AN, set of Abstract Names; and
- PL, set of privacy level identification names, ranging, for example, from "*free*" to "*restricted*".

The policy makes use of the following user attributes:

- $vo : VO$, the VO a user is affiliated to. Since this attribute can not be modified as a side-effect of subject's actions, but only with an administrative action, it is considered as an immutable attribute. For simplicity, we assume that an user is affiliated to only one VO. For user u, we use notation $u.vo$ to denote the VO to which s/he is affiliated.
- $assigned : VO \times AN \rightarrow \mathbb{N}^+$, a permanent, immutable attribute storing the number of maximum concurrent access to an abstract name by a user behind a VO.
- $usage : VO \times AN \rightarrow \mathbb{N}$, a temporary or permanent mutable attribute storing the number of current concurrent access to an abstract name by the user behind a VO.

The policy also uses the following abstract name attribute:

- $privacy : PL$, an immutable attribute encoding the privacy level of the AN.

The policy example states that each time a user is permitted to read an abstract name, the *u.usage* value is less than the *u.assigned* value. If this condition happens to be true, before effectively granting an access the *u.usage* value is augmented, so that subsequent access by the same user should evaluate the policy again. Additionally, the same policy states that each time a user is reading an abstract name, the *an.privacy* value must be set to "free". If this condition fails, for example when the *an.privacy* value is set to "restricted", i.e after an admistrator action or at a determined point of time, then the read access is revoked. Whether the access is revoked, or if the user simply end it, the *u.usage* value is reduced.

This policy is complex enough to show a traditional authorization policy, enforced before the effective usage, together with an on-going authorization policy, which must be enforced during the read phase. Moreover, it shows updates to one of the user attributes, performed before effectively granting the access and after the end of it. Put this in UCON terms, the policy is a combination of an UCON$_{preA_1}$ policy and of an UCON$_{onA_3}$.

Figure 5.3 shows an UML diagram of the sequence of operations corresponding to our example policy. We refer to Chapters 9 and 10 for a clarification of the objects and the actions used in the Figure.

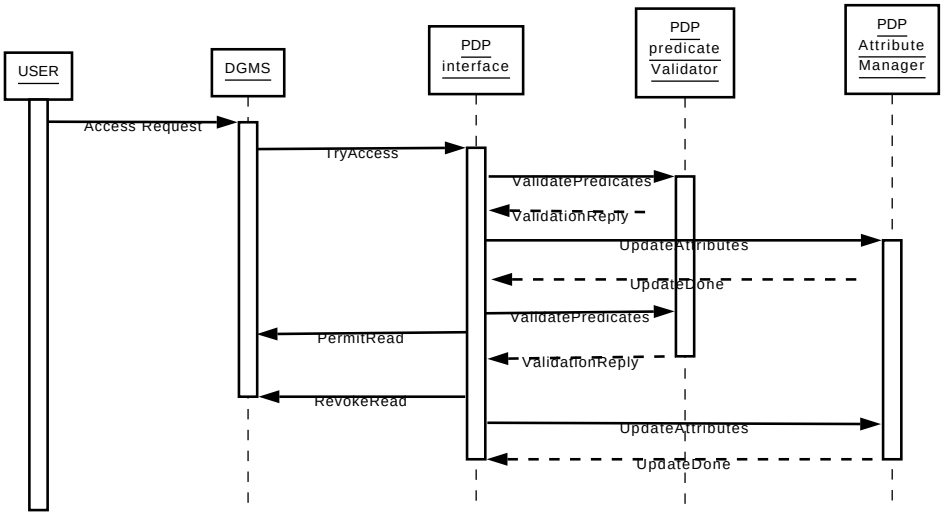

Figure 5.3: Sequence diagram corresponding to our example UCON policy

6

Software Engineering Instruments

One of the objectives of this theses is helping software developers to produce a highly capable framework for managing access and usage control, to be used by Data Grids, and by distributed systems with similar security requirements. It's easy to say that a security software like this should be the result of a proper software engineering process.

Whithin this Chapter, fist of all we overview the software development process in use. After that, we look more carefully at some parts of the software development process we used within this theses.

6.1 Notions about Software Development Process

A software development process is a structure imposed on the development of a software product. In the last dozen years, software engineers proposed different repeatable and predictable processes for improving software productivity and quality. As a result, there are several models for software development, each describing approaches to a variety of tasks or activities. Software development processes are by either using some formalization of the seemingly unruly task of writing software, or by the application of project management techniques to writing software.

A software development process is composed by a set of tasks. The sequence these tasks are executed, and the interactions among them differentiates a model from another. Basically, the tasks are: *requirement analysis, architecture, specification, design, implementation, testing, deployment and maintenance*

- The most important task in creating a software product is doing the **requirements analysis**. We call stakeholders the people affected by the systems. And stakeholders have requirements that need to be satisfied. Once the general requirements are gleaned from the stakeholder, an analysis of the scope of the development should be determined and clearly stated.
- The architecture of a software system, or **software architecture**, is the structure or structures of the system, which comprise software

elements, and the relatioships among them. Architecture is concerned with making sure the software system will meet the requirements of the product, as well as ensuring that future requirements can be addressed.

- **Specification** is the task of precisely describing the software to be written, possibly in a rigorous way. Safety-critical software systems are often carefully specified prior to application development. Specifications are most important for external interfaces that must remain stable.

- **Design** software means using a specification and an architecture for designing a candidate software solution.

- **Implementation** is the part of the process where software engineers actually program the code for the project.

- Software **testing** is an integral and important part of the software development process.

- **Deployment** starts after the code is appropriately tested, is approved for release and sold or otherwise distributed into a production environment.

- **Maintenance** is about enhancing software to cope with newly discovered problems or new requirements of the software.

A software development process use all or most of the above tasks, sometimes repeating some or all of them more than once for the same software. Some notorious development processes can be categorized as prescriptive or agile.

The *prescriptive processes* represent the older software engineering paradigm. The oldest of all is the *waterfall* process, where the developers follow the above steps in order, thus after each step is finished, the process proceeds to the next step. This approach is usually used in high risk projects. It is not a "fast" development process. Other prescriptive processes are the *incremental* model, the RAD *(Rapid Application Development)* model, or the *spiral* model.

The *Agile software development* models are built on the foundation of iterative development. Agile models have a more people-centric viewpoint than traditional approaches, by using more feedbacks rather than planning, as their primary control mechanism.

There is a rich literature on software engineering process, but we have no interests in it. Our interests are in some more specific part of the software engineering landscape, namely software architectures and formal software engineering methods.

6.2 Software Architectures

In Chapter 9 we will show two software architectures. They are an important result of the work contained in this thesis. But what's exactly a software architecture? There are many definitions of it, but we like the one given Rozansky and Woods in [84]:

> *The **architecture** of a software-intensive system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them*

The key parts of these definitions are system's *structure* and *externally visible properties*.

- The system *structures* may be static or dynamic. The static structures define the system internal design-time elements and their arrangements. The dynamic structures define the system runtime elements and their interacions.
- The *externally visible properties* are *externally visible behavior* (what the system does) and *quality properties* (how the system does it). While the former defines the functional interactions between the system and its elements, the latters identify non-functional properties like performance, security or scalability.

We strongly believe in the importance of a well structured software development process. Producing a well-documented and understood software architecture is a big part of such a process.

6.3 Formal methods

Formal methods are mathematical approaches to solving software (and hardware) problems at the requirements, specification and design levels. Formal methods are most likely to be applied in environments and applications where the software is safety critical.

There is plenty of support for formal methods and notations in the literature. Examples of formal methods include the B-Method [1], Petri nets [68] and VDM. Various formal specification notations are available, such as the Z notation [91]. Formal software development is starting to grow, with the application of the Object Constraint Language (OCL) [83]. Another emerging trend in software development is to write a specification in some form of logic and then to directly execute the logic as though it were a program.

Formal specification is simply a matter of being more explicit and specific in defining the requirements of software. At the simplest level this can

take the form of Design by Contract, where functions and procedures specify pre- and post-conditions and loops and objects include a set of invariant properties. In the more rigorous case formal specification involves building an explicit mathematical definition of the requirements of the software. Using such a definition one can prove the correctness of the system, or simply prove theorems about properties of the system. An implementation can also be checked against such a formalised specification, verifying that the implemented code does indeed do precisely what the requirements claim. At the most rigorous level the initial formal requirement specification can be expanded, through (mathematically rigorous) refinement, to ever more specific and detailed specifications resulting eventually in executable code.

All of these different levels allow a significantly greater degree of analysis of the software, be it improved static and runtime checking with contracts, to more complex data flow analysis and proof with more complete specifications. In the same way that static types allow more rigorous checking at compile time, catching a lot of simple errors, contracts and specifications allow even more analysis and checking, catching even more errors at the early stages of development when they are most easily and efficiently fixed.

Not every project really needs formal specification, but there are a great many software projects that could benefit greatly from some level of formal specification - a great many more than make use of it at present. Formal specification doesn't need to be used for a whole project, only those parts of it that are sensitive to error. Likewise any network services could easily benefit from formal specification on the network facing portions of the code to significantly reduce the possibility of exploits: it is far easier to audit and verify code that has been properly specified. Security software, and implementations of cryptographic protocols, are far safer if formally specified: with cryptography the protocols are often rigorously checked, and many exploits relate to errors where the implementation fails to correctly follow the protocol. Finally mission critical business software, where downtime can costs millions of dollars, could most assuredly benefit form the extra assurances that formal specification and verification can provide.

Formal specification isn't a silver bullet, and it isn't the right choice for every project. On the other hand you can use as much specification and verification as is suitable for your project. You can apply formal specification only to the portions that are critical, and develop the rest normally. More importantly, the techniques, languages and tools for formal specification continue to improve. The more powerful the methods and the tools, the more projects for which formal specification becomes a viable option. In an increasingly network oriented computing world where security and software assurance is becoming increasingly important, the place for formal specification is growing.

6.4 Goal-oriented Approaches and KAOS

Goal-oriented approaches have been recognized to be essential components in many formal software engineering approaches. For example, one of the fields where they are used most stands in Requirement Engineering.

A goal is a prescriptive description of system properties, formulated in non-operational terms. A system includes not only the software to be developed but also its environment. Goals are refined and operationalised in a top-down manner as the system is designed, or with a bottom up approach while re-engineering existing systems. The approach also supports adverse environments, composed of possibly malicious external agents trying to undermine the system goal rather than to collaborate in the goal fulfillment. As a Grid system is typically composed of a large number of nodes interacting in an open and possibly adverse environment, this approach fits our needs well.

One of the requirement engineering methodology that makes use of goals is **Knowledge acquisition in automated specification** (KAOS). KAOS is a generic methodology based on capturing, structuring and precise formulation of system goals [99].

A KAOS model is composed of a number of sub-models, these include:

- The **goal model** captures and structures the assumed and required properties of a system by formalising a property as a top-level goal which is then refined to intermediate subgoals and finally to low-level requirements representing goals that can be operationalised. Goals may be organized in AND/OR refinement-abstraction hierarchies, where higher-level goals are generally strategic, coarse-grained and involve multiple agents whereas lower-level goals are technical, fine-grained and involve fewer agents. In such structures, AND-refinement links relate a goal to a set of sub-goals possibly conjoined with domain properties or environment assumptions; this means that satisfying all subgoals in the refinement is a sufficient condition in the domain for satisfying the goal. OR-refinement links relate a goal to a set of alternative refinements.

- The **agent model** assigns goals to agents in a realizable way. Agents include software components that exist or are to be developed, external devices, and humans in the environment. Discovering the responsible agents is the criterion to stop a goal-refinement process.

- The **object model** is used to identify the concepts of the application domain that are relevant with respect to the requirements and to provide static constraints on the operational systems that will satisfy the requirements. The object model consists of objects from the domain

and objects introduced to express requirements or constraints on the operational system.

- The **operation model** details, at state-transition level, the actions an agent has to perform to reach the goals it is responsible for.

The KAOS language has a two-layer structure: an outer conceptual modelling layer for declaring concepts (such as goals, objects, agents, etc) and links between concepts (such as goal refinements, responsibility assignments of goals to agents, etc.); and an inner formal assertion layer for formally defining concepts.

The rigor of the KAOS methodology stems from the fact that any concepts defined within its sub-models incorporate formal definitions using Linear Temporal Logic (LTL) [101] formulae. LTL formalae consist of combinations of the usual first-order predicate logic operators ($\wedge \vee \neg \rightarrow \leftrightarrow$) along with the following temporal operators about the predicate P and Q:

- $\Box P$, which says that P is always true from now on;
- $\Diamond P$, which says that P will be true sometime in the future;
- $\circ P$, which says that P will be true in the next state;
- $\blacksquare P$, which says that P was always true till now;
- $\blacklozenge P$, which says that P was true at sometime in the past;
- $\bullet P$, which says that P was true in the previous state;
- $P \mathcal{S} Q$, which says that Q has been true since a time when P was true;
- $P \mathcal{U} Q$, which says that P will be true until a time when Q will be true.

We also write $(P \Rightarrow Q)$ to mean $\Box(P \rightarrow Q)$ and $(P \Leftrightarrow Q)$ to mean $(P \Rightarrow Q) \wedge (P \Leftarrow Q)$.

Figure 6.1 shows an overview of the KAOS models and their interrelations. This example has been widely taken from [56], and shows excerpt of the goal, object, agent and operation models for a meeting scheduling problem. Within the goal model of Figure 6.1, the top-level goal [ConvenientMeetingHeld] is AND-refined into the subgoals [PrtcptsCnstrKnown], [ConvenientMeetingPlanned], and [PrtcptsInformed]. The goal [PrtcptsCnstrKnown] has two alternative, OR-refinements. The goals are further refined until the leaf goals are identified as either assumptions of the system, or requirements. Then, analyzing the goal model, we can identify the objects of the system, their relations, and the agents involved. Figure 6.1 shows that, for the case of the meeting scheduling problem, two objects are identified and shown in the object model. The leaf goal [PrtcptsCstrRequested] is a requirement of the system and can be assigned alternatively to the *Scheduler*

software agent or to the *Initiator* agent. From the agent model and the goal model, we can identify the operations of the system. Figure 6.1 shows how the operation model uses *domain pre-* and *post-* conditions of the operation [SendCstrRequest] to capture what any sending of a constraint request is about in the application domain.

Within this thesis, we will use a methodology for validating that an abstract specification for enforcing UCON_{abc} policies is sound and complete. Such methodology, which will be explained in Section 10.1, is liberally inspired from KAOS.

6.5 Engineering a Security Software

> *The biggest problem in computer security today is that many practitioners don't know what the problem is. Simply put, it's the software! You may have the world's best firewall, but if you let people access an application through the firewall and the code is remotely exploitable, then the firewall will not do any good (not to mention that the firewall is often a piece of fallible software itself). The same can be said of cryptography. In fact, 85% of CERT security advisories*[1] *could not be prevented by cryptography*
> Bruce Schneier

In Section 6.1 we made a very brief survey on the software development processes in the literature. One of the questions that need an answer is: what is, if there is one, a software development process for a security software? And why is a good software development practice so important?

The most obvious answer is that we want to help producing good software. A good software is a highly capable product, with few bugs and vulnerabilities. As stated in [106], the major source of vulnerability has been recognized to be poor-quality software. Internet-enabled applications present the larger category of security risks. Obviously, Grid middlewares and applications follow in this category.

Secure software practitioners advocates to think of software security as risk management. Software security risks come in two main flavours: architectural problems and implementation errors. Even if we produced some working code, we can not help much with the second kind of problems. There are good books on programming good software with security in mind out there, like [106] or even the more general [62]. Anyway, we did our best for integrating security in the architectures shown in Chapter 9.

[1] CERT is an organization that studies Internet security vulnerabilities. See http://www.cert.org

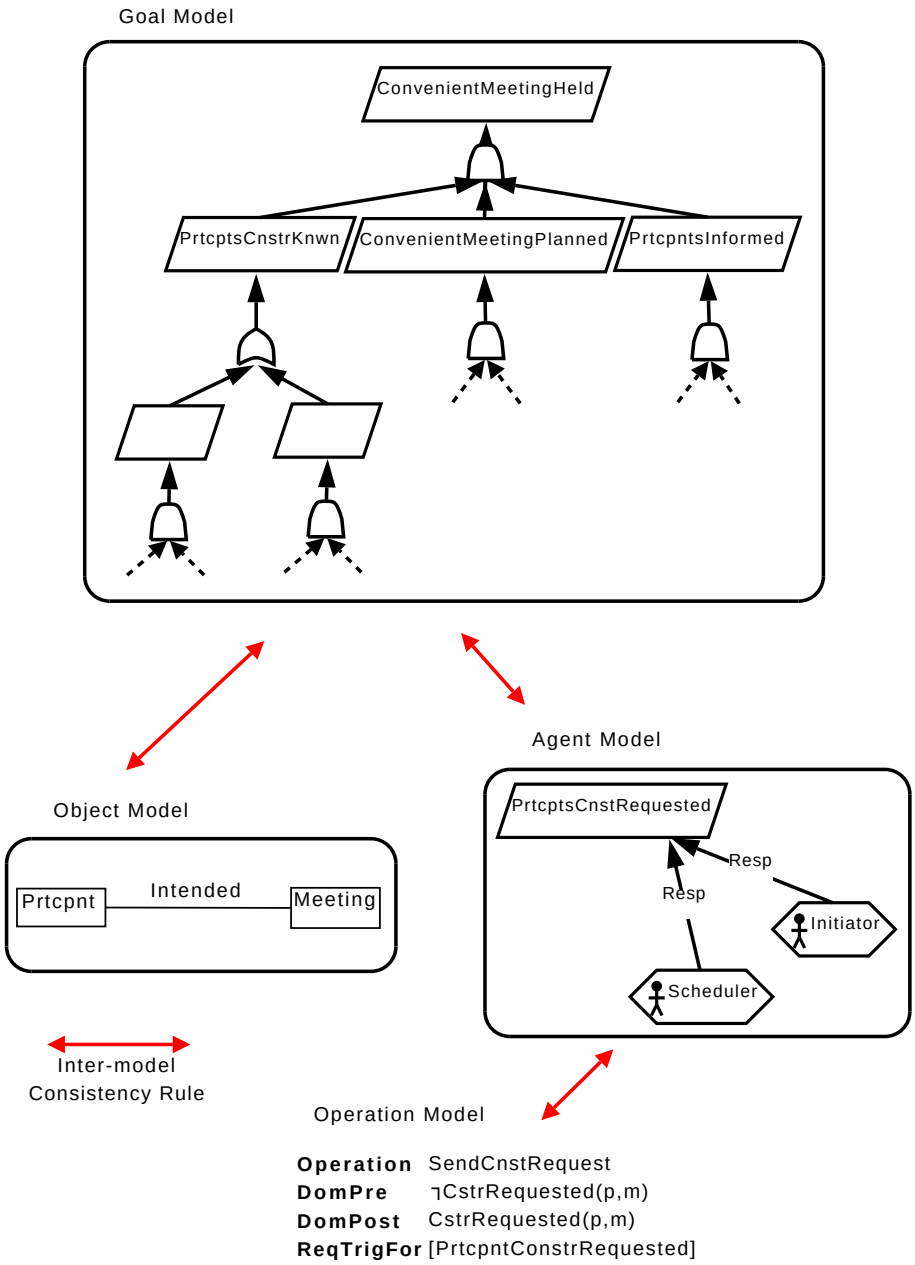

Figure 6.1: Overview of the KAOS models

The most important prerequisite to software risk management is adopting a high-quality software engineering methodology. Software that is properly engineered goes through a well-structured process from requirements design, through detailed specification, to actual implementation. A compressed development schedule means that software engineering processes are often thrown out of the window. Sometimes software specifications are very poorly written (if they exist at all). We think this is just awful. That's the reason why, within this theses, we produced a formal software specification for one of the most critical software systems to be produced. Such architecture may be found in Section 9.2. For its production, we used a formal software requirement engineering methodology we will review in Section 6.4, while the next Section overviews the formal methods of software engineering.

Part III

Models, Architectures, and Specifications

7

Foundations for Usage Control on Data Grids

This Chapter shows the foundations for studying and developing usage control techniques on data Grids. First of all, we review old and new requirements concerning Grid security, with a focus on Data Grids. Such requirements guided the work that can be found in subsequent Chapters. The second part of this Chapter presents a distributed security model suitable for Grids and distributed systems alike. Such model, although linear and simple, couldn't be found in any precedent literature on policy security models. Then, we show how to apply such model to Data Grids.

7.1 Security Requirements for Grids

In distributed systems design, security cannot be orthogonalized as a concept: securing the system cannot be done separately, since security may have a profound impact on the actual architecture and design. In Grid infrastructures there are many constraints due to security considerations.

On the other hand, the Grid infrastructure, and the Grid applications, have an influence on the way the security models, sytems, frameworks and services are designed, developed and deployed.

In this Section first of all we will outline the basic security concepts that concern Grid developments and deployements. Then, we will concentrate on Usage Control requirements for Data Grids.

7.1.1 General Security Requirements

In section 2, we've made a really brief history of the evolving of the concept of Grid, concerning Computing Grids and Data Grids, and we also showed a simplified taxonomy for Grids. To understand all the security requirements, we have to think to who is using these Grid typologies now, who is going to use Grid technologies soon, and who wish to use them, but can not trust them for some security reasons [37]. At the present time, the majority of Grid middlewares and tools are growing behind some specific needs, mainly

HEP (High Energy Physics) experiments. HEP applications produce and consume a considerably high amount of data with heavy impact on the bandwith, but probably they don't need a high security system, because the main purpose of this activities is to be fast. At the same time, other Grid middlewares grow for chemicals or bioinformatics necessities, with different, tighter, security requirements.

A growing number of researchers and VOs will born. They will use Grids, peer-to-peer systems, or whatever distributed paradigm will be in place that could help with their computing needs. These VOs may pose new security requirements. Just to make the simplest example, in the next generation of Grids file sharing, a user will want to give access to his/her files only to a limited set of people, identified by some kind of property. To do this, there's the need for a high control over who is authorized to view or modify the data [44].

A problem comes from the fact that security is, indeed, a cost: performance, scalability, manageability, maintainability, usability of a Grid have an effective relation with the security service. For this reason, some applications (and specifically their users) could consider strict security checks as an annoyance, an aggravation.

Every Grid application may have a specific set of security requirements, and a Grid middleware should be capable to deal with a vast number of those. In other words, a Grid security framework should deal with different protection functionalities. Different Grid applications should be able to determine the way the Grid guarantees data integrity and confidentiality. Different Grid authentication and authorization capabilities need to be in place. The solution is to conceive a really flexible system, with no explicit bindings with a specific application.

The next generation Data Grid elements should be able to publish the Quality of Protection (QoP) they can assure to the data they own. The QoP exposed by DGMS, network, SEs and other software elements would be decisive for the entire Data Grid system. Security negotiations should be used to establish secure sessions between the endpoints. A security infrastructure featuring support for negotiations and establishment of end-to-end and/or hop-to-hop security associations has broader applicability to general networked environments like Grids. Security negotiations require some brokering agent to mediate between the endpoints.

In the literature, or simply by surfing the web, the readers may also find many requirements which we consider too technical to be listed here: we are not interested in design-time or development-time requirements. We are interested in those requirements that helps us identifying suitable security models for Grids, and for the specification of Data Grid security architectures.

That being said, we recognize in the CIA triad (*Confidentiality*, *Integrity* and *Availability*) of Section 3.1 the building block behind a Grid security

infrastructure. Another important property should be *data lifecycle management*, which is the process of managing data throughout its lifecycle. *Fault tolerance* is a desirable feature especially when transfers of large data occur. Grid services should also be able to *trust* each others.

7.1.2 Requirements for Data Grids Usage Control

We don't have a single source of requirements for Data Grids Usage Control frameworks. Some of the requirements that are listed here can be found in [102] or in [34], but we believe they do not represent a comprehensive list.

1. **A Grid Security Model.** A Data Grid usage control framework should implement a Grid security and policy model. Referring to what we wrote in Sections 4.2.1 and 4.2.2, the Grid security model of choice should be the "Grid usage" model, since relying on local (SE) security implementations makes impossible creating a flexible system.

   The other downside of using local ACL implementations is the lack of standardization. The only attempt for a standardization of ACLs has been made within POSIX. The POSIX.1e ACLs [97] are an estension of the POSIX.1 permission model, the standard 9-bit access permissions of the UNIX systems. The extended ACLs support more fine-grained and complex permission scenarios, that are difficult or impossible to implement with the minimal model. Unfortunately, the work behind ACLs never became a POSIX formal standard, and at the time of writing there's a wild mix of implementations with subtle differences and incompatibilities. We aren't going to explain how they work, we just say that they can be applied to files and directories, increasing flexibility and security. For our purposes, the worst problems come when we have to preserve permissions in a distributed system: it's very difficult to implement a system able to preserve as much information as possible. There are a number of complications that make the operation prone to implementation errors, especially when we have different kernels and file systems. The semantics of ACLs differ widely among UNIX systems alone, not to speak of non-UNIX ones.

   That being said, it's now clear that the model of choice from the Grid user's point of view should be the Grid security mode since this integrates the site peculiarities into a uniform security model where every grid site looks the same.

   For what regards a policy model, we believe that none of those presented in Section 4.3 can be used effectively. For more information, see the "Multiple Authoritative Points" requirement in this Section.

2. **Separating the Policy from the Infrastructure.** As a concequence of the previous requirement, a future authorization service will

separate authorization infrastructure from the policy itself, providing only secure environment and mechanism for site-authority controlled policy enforcement. The policy evaluation engine will be implemented as a separate service that will be able to call external separate decision points.

3. **Multiple Authoritative Points.** In a distributed system like a Grid, there may be small to larger number of different resources, each one controlled by a different policy officer. Each policy officer is a *Source of Authority* (SoA) for an *authoritative point*, viz. authoritative sources of authorizations and usage control. When a client service is requesting the permission to access a single remote resource, a number of policies maintained by different SoAs may have to be evaluated. What this means is that the VO should be able to decide users' membership policies and users' authorization policies for which it is authoritative, and, in parallel, that resource owners should be able to decide the users' authorization policies for which they are authoritative.

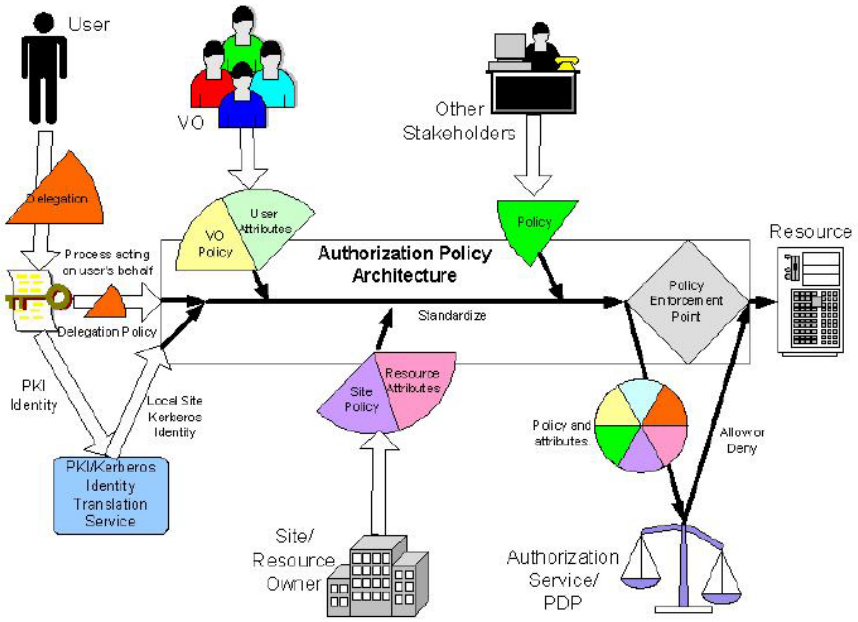

Figure 7.1: Framework for combining authorization policy from different parties. [96]

This requirement was historically advocated by the Globus and EGEE security teams. A picturial illustration of it can be found in Figure 7.1. This Figure shows that VOs, sites, and other stakeholders all participate in the definition of policies to be evaluated, i.e. they are au-

thoritative points. VO policies, user attributes, site policies, resource attributes, together with policies from other stakeholders and/or delegation policies, are to be evaluated by at least one (logical) authorization service, and enforced by at least one (logical) enforcement service. Up to now, there is no existing Grid usage control framework coping with this requirement. This is the main reason why in Section 7.2 we present a usage control model for Grids and distributed systems that is capable to cope with this requirement.

4. **Security Policy Characteristics.** The policy model should be capable to encompass traditional access control models. Moreover, the policy model should not pose constraint on the degree/level of granularity of usage control, ranging from storage space level to individual data access restrictions.

5. **Standard Languages and Protocols.** A usage control framework should be based on a recognized policy expression language and exchange format, and should use a Request/Response protocol to allow intra-site and multiple site scalability. This implies the investigation for the use of "standard" format languages and protocols.

   Some security experts advocate that some (not all) of the proposed security "standards", which are mostly XML-based languages, still suffers from instability and poor expressibility. This is the case, for example, of P. Gutmann [43]. Going beyond these kind of discussions is not one of the purposes of this thesis.

   Personally, we agree that some of the proposed standards are not mature enough to be used effectively. For example, we believe that the adoption of standard policy languages, like for example XACML, could represent a limit to the number of policy models such a language could be able to express. Within this thesis, we don't make an analysis of existing security policy languages. Some others have tried [26, 71], even if we have no notions about comparative studies including XACML.

   Even if standard languages may pose limitations, we still consider them as parts of the requirements for a Data Grid usage control implementation. For example, in Chapter 8, we show how we used a security standard language while re-engineering a Grid Attribute Authority.

6. **Domain Trust.** A usage control framework should be able to explicitly trust or not trust usage control decisions coming from other domains. This means that a trust service should be in place. Within this work, we don't plan to solve this requirement. Instead, we consider the existence of a trust service as a pre-requisite for a usage control service.

7. **Other Requirements**

   - authorization should be largely based on the VO internal structure;
   - the dinamicity level should not affect the performance of the whole system;
   - the security service should be mostly a black box from a user perspective, even if there is a minimum the user should be aware and need to know, especially for what regards the management of credentials.
   - users could be members of any number of VOs;
   - users could have any number of roles within a VO;
   - it should not be possible to successfully carry out an action or successfully submit a job where the authorization is not valid;
   - it should be possible to determine the list of resources to which a user has access and what actions they are allowed to carry out in the VO(s) and role(s) set for the current session;
   - it should be possible to determine if a certain user in the current session has access to a certain resource and what actions they may carry out on that resource;
   - the authorization method should be application independent;
   - the authorization decision making process should be the same/-consistent within a VO;
   - it should be possible for a VO manager to add/remove a user and/or a role from a VO;
   - the authorization requirements on Grid Data (GD) access shall hold regardless of replication;
   - and, most of all, the whole service must be scalable; regarding this last requirement, we will make some further consideration in Section 11.2.

7.2 A Grid Usage Control Model

In this section we define a usage control model for Grids and distributed systems. Then, we will apply this model to the Data Grid abstraction of Section 2.4.

7.2.1 The Distributed Usage Control Model

Up to now, there's no existing security model that can cope with the inner nature of Grids. Some of the security requirements of a (Data) Grid can not

be satisfied by simply applying the policy models we reviewed in Section 5 to a Data Grid. For example, one of the most important requirements for Data Grids Usage Control that can be found in Section 7.1.2 is about ensuring the presence of multiple authoritative points (number 3 of the list). The model we propose in this Section can deal with such requirement.

Within the model we propose, that we call *Distributed Usage Control Model* (D-UCM), policy officers could impose the evaluation of local policies. We say that a single usage decision comes from the evaluation of a *workflow* of local usage control steps. For example, when the workflow of a complete usage control is made of three separate usage control steps, each one of the three must be satisfied.If one of the usage control steps can not be satisfied, the entire usage is not permitted.

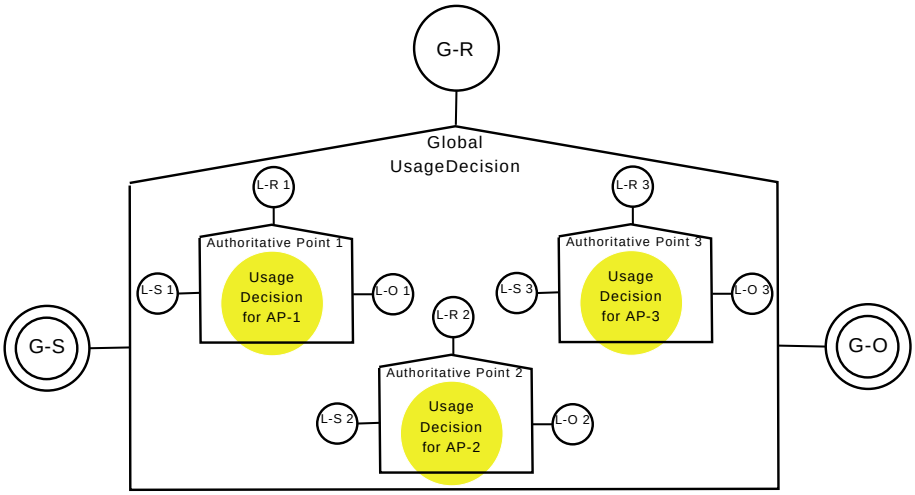

Figure 7.2: The Distributed Usage Control Model

Figure 7.2 shows a pictorial overview of D-UCM. Within this Figure, we show that three distict authoritative points each impose the evaluation of a local usage decision (L-UD) step. Each step have to be satisfied for the enforcing of a global usage decision (G-UD). A central workflow orchestrator, with responsibility for the G-UD, is needed.

Within this model, the evaluation of a L-UD step is seen as an atomic action. The model doesn't pose any constraint neither on the way authoritative points enforce usage control steps, nor on the nature of the security policies that have to be evaluated to reach a L-UD. For example, a L-UD may require the evaluation of a vast number of distributed and concurrent policies, but all this machinery is under the responsibility of the local Source of Authority (SoA).

A G-UD is based on a global subject (G-S), a global object (G-O) and a requested global right (G-R). To reach a L-UD, each SoA encode G-S, G-O and G-R respectively in a local subject (L-S), local object (L-O) and local

right (L-R). The relation between the global and local subjects, objects and rights is dependent from the application using the model.

We believe we can apply this model not only to Data Grids, but to all those distributed applications with the same requirements of distributed, multiple authoritative points. Therefore, the challenge for controlling the resource usage in Grids and distributed systems is knowing which are the authoritative points involved in a usage request.

7.2.2 D-UCM for Data Grids

We now apply the *distributed usage control model* (D-UCM) of Section 5.2 to the Data Grid. For what concern the Data Grid environment, we choose the UCON$_{abc}$ usage control model as policy model because of its high capabilities. UCON$_{abc}$ satisfies the requirement on policy models of Section 7.1.2 (number 4 of the list).

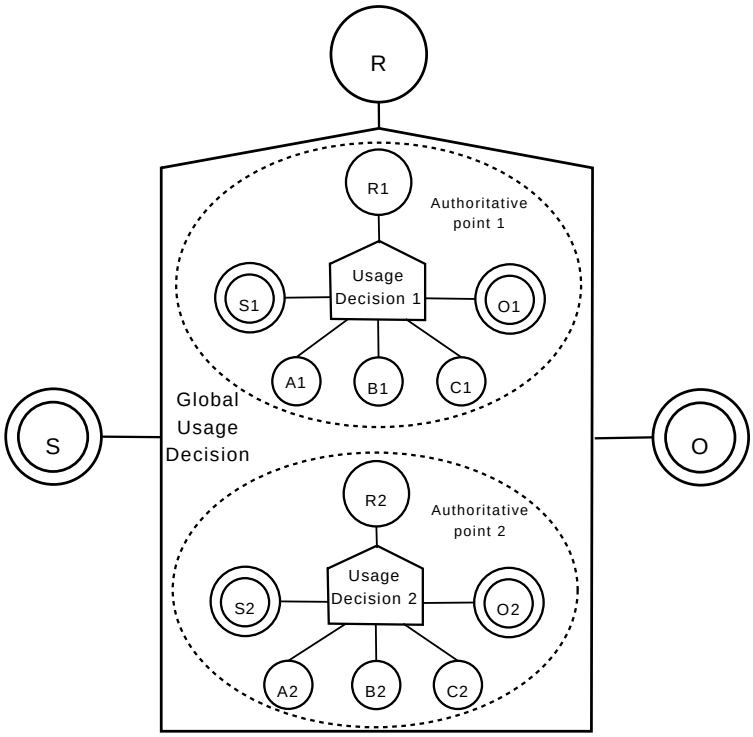

Figure 7.3: A 2-step D-UCM, where each step enforces a policy pertaing to the UCON$_{abc}$ model

We now consider the terminology introduced in Section 2.4. In a Data Grid, GDs (Grid Data) are stored (and transferred and replicated) in GSSs (Grid Storage Speces) by the SEs (Storage Elements). A client performing an access to a GD should be authorized to access the data itself, and to

use the GSS. Therefore, the policies of the single steps should be written by those policy officers which are SoA for the GDs (e.g. VO admins or simply VO participants), and by those policy officers which are SoA for the GSSs (e.g. SE admins). Therefore, we identified two authoritative points, which are DGMS and SEs. A Complete usage control in Data Grid then follows a two-steps workflow. From now on, we refer to each of these steps as *data usage control* (D-UC) and *storage usage control* (S-UC). Each step corresponds to the enforcing of (at least) a UCON_{abc} policy.

We now pose some constraints on the relation between G-S, G-O, G-R and L-S, L-O and L-R of D-UC and S-UC. Each single L-S represents the G-S as it is recognized by respectively the DGMS and the SE. Similarly, the L-R represents the G-R as it is recognized by the DGMS and by the SE. The object of the D-UC is the unique identifier of a GD, i.e. the *abstract name*. The object of S-UC is, instead, the GSS itself. By doing this neat separation between the objects of D-UC and S-UC, we highlight the role of the authoritative points. Moreover, by doing this separation, the policies of the different steps will never overlap. Figure 7.4 shows this two-step usage control.

Figure 7.5 compares security models for Data Grids as seen in [34], and summarized in Section 4.3, with the distributed security model we are proposing within this thesis. As it's clear by looking at this Figure, the real difference stands with the usage of two (UCON) PDPs, one for each usage control step.

We believe that, by applying this simple distributed security model, we are capable to solve one of the most challenging security problem that affects Grids, but also many distributed models, that is the centralization of security policy management and enforcement. This model logically split the authorization decision function by using multiple, non-overlapping, authorization services. For what regards its application to Data Grids, as will be made clear in Section 9.1, such separation is achieved with two UCON PDPs connected by an *orchestrator*.

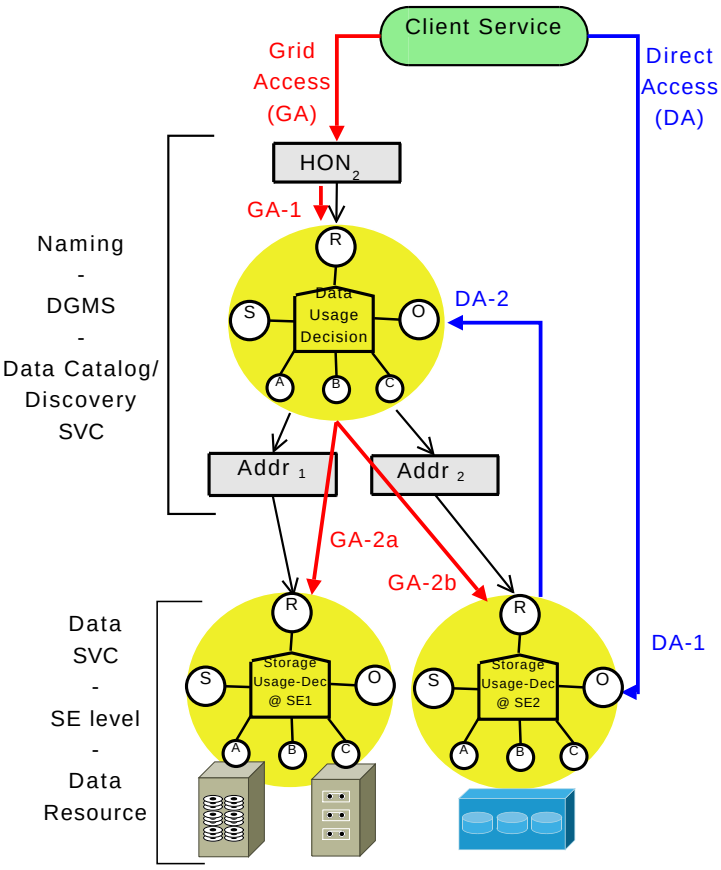

Figure 7.4: The two-steps Grid usage control

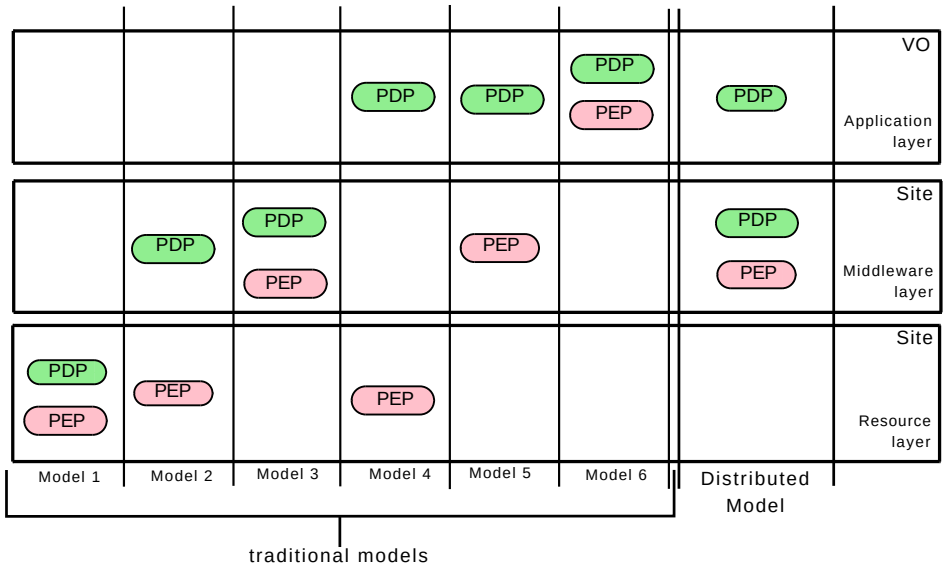

Figure 7.5: Comparison of security models for Data Grids (inspired from [34])

8

Grid Authentication for Usage Control systems

This thesis wants to help solving some of the Data Grids security problems by applying usage control models and techiques. As we said in Section 4.1, we consider usage control as a process composed by two steps, i.e. the "Grid Authentication" and the "Grid Authorization" boxes. Even if most of the thesis is dedicated to the Grid authorization phase, this Chapter shows our work for a Grid Authentication service.

An authentication service defines the Grid identity of Grid users, with a set of credentials for her description and identification. Using the OGSA words, the "Grid Authentication" is achieved through the use of one or more CIS (Credential Information Service) of an Attribute Authority (AA).

Within this Chapter, we show the re-engineering of the Virtual Organization Membership Service (VOMS), an AA for Grids we already mentioned in Section 4.4. VOMS is a *de facto* standard when coming to AA for Grids. The re-engineering wants to achieve the "standard languages and protocols" requirement we mentioned in Section 7.1.2 (number 5 of the list), through the use of the SAML Security Assertion Markup Language.

We published a good potyion of the work reported in this Chapter in [104] and [103].

8.1 The Virtual Organization Membership Service

The Virtual Organization Membership Service (VOMS) is an AA focussing on Virtual Organization Management. It releases signed assertions containing attributes expressing a user membership and position in a VO. Such assertions can be used by Grid Services to drive usage control decisions. By using VOMS, users can be organized in a hierarchical structure.

VOMS is composed by a number of software elements, and the most important is the AA-element, which is the main VOMS server. It works as a net server, one instance per VO, auditing for incoming connections requesting attributes. All connections between the server and its clients are mutually authenticated, therefore each client is authorized to retrieve

informations only for herself.

Currently, three different types of attributes may be returned by a VOMS server:

- **groups**: they represent an organizational structure. There is a root group, representing the whole organization, and a series of subgroups, up to any depth. By convention, the root group has always the same name as the managed VO. A user may belong to any number of groups, and group membership information will always be present in the assertions, without the possibility for the users to hide it.
- **Roles**: since not all the users within a group have equal rights, it is a common situation when some of the members of a particular group need some additional privileges. This situation is represented by roles, which are specific of a group and are granted only if explicitly requested by a user.
- **Generic attributes**: these attributes are in the form `name = value`, where both name and value could be freely chosen by a VO. They are also called *Tag* attributes, and like groups, they will always be present in the released credentials.

In its current version, used in production Grids like EGEE, VOMS releases its *groups* and *roles* attributes using the Fully Qualified Attribute Name (FQAN) attribute [23] to encode the position of a subject in a VO. Each FQAN contains informations regarding the VO, groups and roles the user is member of. A FQAN has the following format:

```
<group>/<sub-group(s)>/Role=<role>
```

This syntax means that the user holds the role `<role>` in the group `<group>`, where the first group is always the name of the VO. Examples of valid FQANs are:

```
/cms
/cms/Role=VO-Admin
/cms/production/montecarlo/Role=writer
```

The order in which the FQANs are presented is significant, since it is the order in which the holder wished the FQANs to be evaluated. In the present service, VOMS encodes FQANs using X.509 Attribute Certificates (ACs). [45, 27]

8.2 Recent VOMS enhancements

In the context of the OMII-Europe project, we re-engineered VOMS [104] to support authorization standards emerging from the OGF. We give here a brief explanation of the work.

VOMS is widely used in the Grid community, thus the aim of our effort is to retain the functionalities of the current service, and extend it with a standard Web Service interface that uses the Security Assertion Markup Language (SAML). Besides the protocol, the new service uses SAML Assertions to contain the subjects' attributes. The service is not meant to be a replacement of the old one, but aim at making the VOMS framework supporting the wider possibile range of use patterns. A driving use case has been those Grid middlewares not using proxy certificates [45], which right now is the delegating mechanism used by many middlewares.

The main interactions between a client and the re-engineered VOMS service are shown in fig. 8.1.

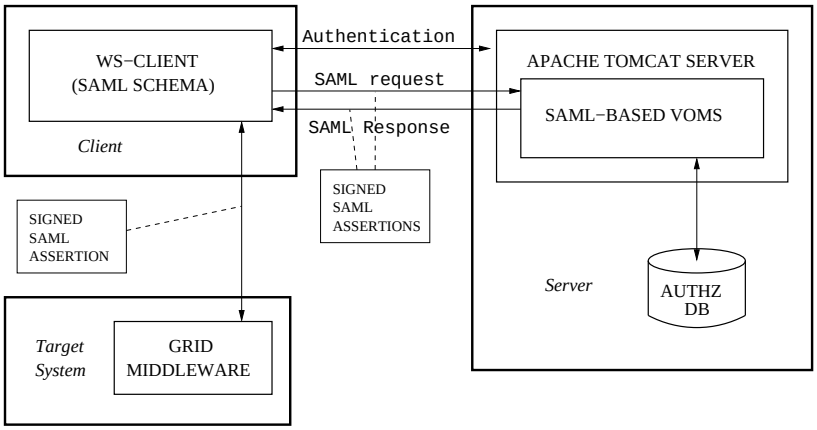

Figure 8.1: Interactions between a client, a re-engineered VOMS service and a target system [104]

8.2.1 The OASIS Security Assertion Markup Language

The Security Assertion Markup Language (SAML) is developed by the Security Services Technical Committee of OASIS. It is an XML-based framework that allows business entities to make assertions regarding the identity, attributes, and entitlements of a subject (an entity that is often a human user) to other entities, such as a partner company or another enterprise application.

We are interested in its capability of releasing attribute-based information to be used for authorization purposes, where, for example, one web site can communicate identity information about a subject to another web site, in order to support some authorization decisions.

SAML defines *Assertions*, packages of information that supply one or more statements by a SAML authority, among which are *Attribute Assertions*. It also defines protocols to request *Assertions* from SAML authorities, and bindings into standards messaging or communication protocols [15].

8.2.2 Service Interface

The VOMS SAML service exposes an interface according to SAML protocols [16] and bindings [15]. The service supports a single operation, whose input is a `<samlp:AttributeQuery>` element and the output is a `<samlp:Response>` element. The `<samlp:AttributeQuery>` element contains the subject for which the requestor wants to retrieve attributes, and eventually which attributes she is interested in. A successfull `<samlp:Response>` contains a `<saml:Assertion>` element with the requested attributes. The elements `<samlp:AttributeQuery>` and `<saml:Assertion>` are used according to the SAML profile for X.509 subjects [82].

8.2.3 Asserting Attributes Using SAML Assertions

When the service authorizes a request, a `<samlp:Response>` is used to return a `<saml:Assertion>` containing a `<saml:AttributeStatement>` with the subject's attributes. In the following, we sketch an example of a SAML assertion (some XML tags and attributes are omitted for brevity). Following the VOMS logic, an assertion must identify the VOMS server that released it, the entity (normally a user) whose assertion is addressed, and the VOMS attributes.

```
1 <saml:Assertion ... >
    <saml:Issuer>
3     CN=omii002.cnaf.infn.it,L=...
    </saml:Issuer>
5   <saml:Subject>
      <saml:NameID
7       Format="urn:..:x509SubjectName"
        CN=Federico Stagni, OU=... >
9     </saml:NameID>
    </saml:Subject>
11  <saml:AttributeStatement>
      ...
13  </saml:AttributeStatement>
  </saml:Assertion>
```

Line 7 must be an uri in conformance with [82], while line 3 must indicate the distinguished name of the issuer host. Line 8 identifies the distinguished name of the assertion's subject. The `<saml:Subject>` element may contain a `<saml:SubjectConfirmation>` element to confirm the identity of the involved subject. The assertion may contain a signature.

Expressing the VOMS attributes using SAML

The core functionality of VOMS is expressing attributes, but the Fully Qualified Attribute Name attribute used by VOMS and the generic "tag" attribute don't map naturally to SAML, thus we need to use the following

`<saml:Attribute>` elements: *vo*, *group* and *role* for the FQAN, and a fourth *tag* attribute.

Concerning the `<saml:AttributeVaue>` containing the VOMS attributes, we defined a new `complexType` type, the `FQANType`, in order to carry the priority attribute of *group* and *role*. Such a type is simply an extension of the `<xs:token>` type, which is itself a built-in type over `<xs:string>` that represents a tokenized string in the W3C recommendation of XML [108]. The following XML schema fragment defines the FQANType complex type:

```
<complexType name="FQANType">
  <simpleContent>
    <extension base="xs:token">
      <attribute name="priority"
      type="xs:positiveInteger"/>
    </extension>
  </simpleContent>
</complexType>
```

Similarly, we defined two new `complexType` types to carry a *tag* attribute. The `TAGType` type contains a sequence of `TAGValue` types, which are extension of the `<xs:string>` XML type. Both the XML schema follow:

```
<complexType name="TAGType">
  <attribute name="TAGname"
    type="xs:string" use="required"/>
  <attribute name="TAGdescription"
    type="xs:string" use="optional"/>
  <sequence>
    <element ref="TAGValue"
      minoccours="1"
      maxoccours="unbounded"/>
  </sequence>
</complexType>

<complexType name="TAGValue">
  <simpleContent>
    <extension base="xs:string">
      <attribute name="qualifier"
        type="xs:normalizedString"
        use="required"/>
    </extension>
  </simpleContent>
</complexType>
```

The qualifier attribute of `TagValue` may be empty, indicating that its content should be assigned directly to the user.

8.2.4 Sending SAML Assertions to Grid Services

In the AC based VOMS, there are command line tools that allow an AC retrieved from a VOMS AA to be embedded in a proxy certificate: over the years, this has proved a very convenient way of sending AC to Grid Services. Some tools using SAML Assertions are using the same proxy-based logic. For example, the Globus Toolkit 4 [1] Community Authorization Service (CAS) [31] binds authorization Assertions to a proxy. GridShib [10] does the same with authentication Assertions. The Web Service Security specification [70] defines a way to send SAML security tokens as part of the SOAP Header [69]. We have preferred this solution, being based on an already consolidated standard and allowing support for services not supporting the use of proxy certificates for authentication. Following is an example of a SOAP message carrying a SAML Assertion (omitted for brevity):

```
<soap:Envelope xmlns:soap="...">
 <soap:Header>
  <wsse:Security wsse="...">
   <saml:Assertion xmlns:saml="...">
    ....
   </saml:Assertion xmlns:saml="...">
  </wsse:Security>
 </soap:Header>
 <soap:Body>
  ...
 </soap:Body>
</soap:Envelope>
```

At the moment of writing, we are discussing with other similar VO management tool implementors a common SAML Attribute profile, that will define the format for SAML Attributes of interest for VOs.

8.3 Scenarios

In VOs, participants join in order to share resources, and essential is the ability to share any resources, independently from the middleware the resources are using. However, such a situation is very seldom seen nowadays, since VOs tend to be confined to a specific middleware distribution, and sharing resources using different middlewares raises a number of problems. Grid Authentication is great issue when trying to share resources that use different middlewares. As it is shown in the next Chapter, Grid authorization is a complex task, since RPs should be able to enforce authorization policies defined by the VOs, by external stakeholders and by the RPs themsevels. In order to develop a consistent authorization framework, such policies could use environment informations and context data, and must use user attributes

[1] http://www.globus.org/toolkit

like the ones provided by VOMS. Therefore, attributes defined by VOMS must be available and understood on any shared resource.

The use of standard languages, protocols, and bindings aims at wider and easier support of VOMS on more middlewares. This is going to increase the range of resources that VOs using VOMS will be able to share. Also users in the VO will be able to choose their favourite client applications. An architectural sketch of such scenario is shown in fig. 8.2.

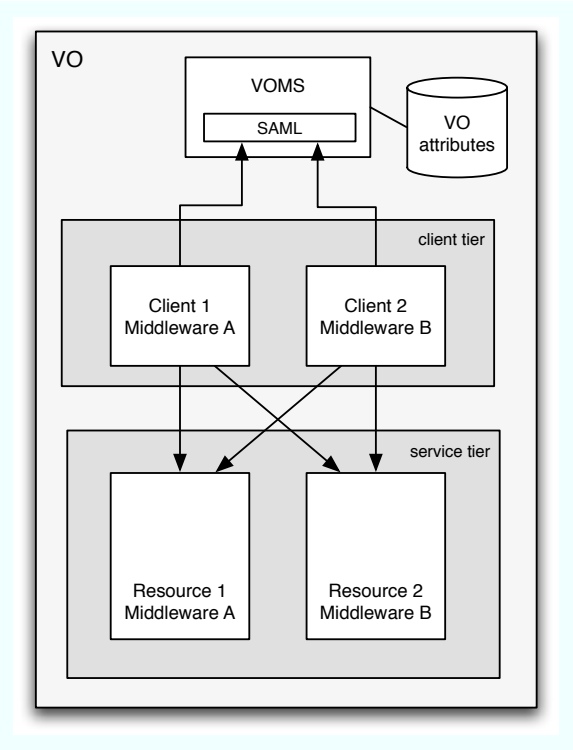

Figure 8.2: Sharing resources using different middlewares [104]

8.4 Related Works and VOMS Alternatives

In today's Grids, there are not many competitors with VOMS, which as said is a de-facto standard when coming to Grid Authentication services. One of the possible alternative is represented by **GridShib**. The GridShib [10] project aims at integrating PKIs with both site authentication infrastructure and Shibboleth [46]. Shibboleth, which is based on SAML is an identity management system designed to provide federated identity, acting as an Attribute Authority. Shibboleth is used as the attribute source.

Another possible alternative can be represented by the Semantic Grids, which are an evolutionary approach for Grids with roots in the Semantic

Web [11] vision. Since the preliminary Tim Berners-Lee's vision of the web evolution, the Semantic Web has become a field receiving great attention. Technologies, specifications, data interchange formats and notations studied and developed for the Semantic Web have recently attracted the scientific community.

Projects like OntoGrid[2] proved the interest of the Grid community in Semantic Web technologies. The challenge is the sharing and deployment of knowledge to be used for the development of innovative Grid infrastructure, and for Grid applications: the Semantic Grid. As stated in [24], the Semantic Grid is an extension of the Grid in which rich resource metadata is exposed and handled explicitly, and shared and managed via Grid protocols. The layering of an explicit semantic infrastructure over the Grid infrastructure potentially leads to increased interoperability and greater flexibility.

In a Semantic Grid, following the terminology introduced in [24], each *Grid Entity* is associated to a *Knowledge Entity* (KE) through a *Semantic Binding*. KEs are special types of Grid Entities that represent or could operate with some form of knowledge. Examples of KEs are ontologies, rules, knowledge bases or even free text descriptions that encapsulate knowledge that can be shared. Semantic Bindings are the entities that come into existence to represent the association of a Grid Entity with one or more KE.

Example of KEs representing the Grid Entity *Grid User* and the Grid Data (GD) are shown in Figure 8.3. A semantic-aware DGMS could associate a data KE like this one to each of the managed *abstract names*. The Grid User KE graph is inspired from [24], while GD graph has been derived from the CCLRC scientific metadata model [95]. These examples are not meant to be complete. We also note that a GD description normally makes use of application-dependent metadata, thus in a real system, a KE of a GD could be much more complicated than the one shown here. Each Grid User is simply described through the use of three fields: the *Institution* he/she is affiliated with, the *Investigation* he/she takes part in, and the *Job or Role* he/she is doing as part of the *Institution*. Instead, each GD is described not only by the *Type* (e.g. file, or stream), but also by the *Program* of work, the supported *Study*, and by an *Investigation*. The interested reader should refer to [95] for an complete explanation of these fields. Examples of valid values for the *Institution* field could be "INFN" or "STFC", while values for the *Investigation* field could be "measurement", "simulation" or "experiment".

User Knowledge Entities like the one we put in Figure 8.3 can be used, for example, to represent who is a Grid User in a Grid, what's her role inside a VO, and what are the experiments she takes part in. KEs can be considered as credentials to better identify users in a Grid, thus implicitly realizing the Grid Authentication phase.

[2] web address: `http://www.ontogrid.net`

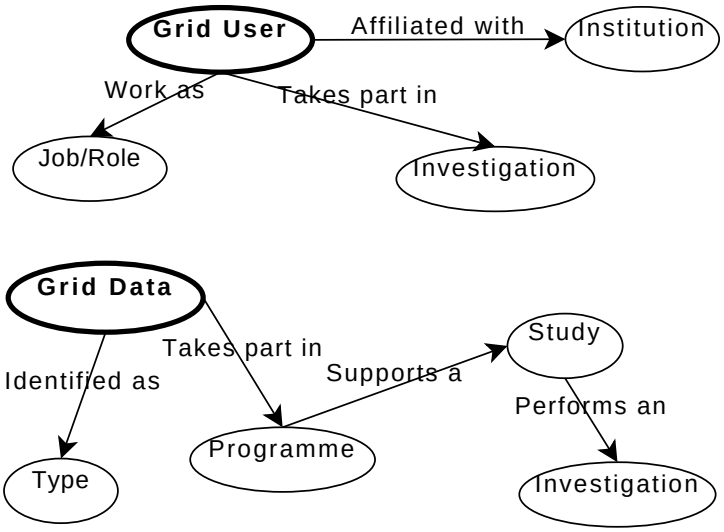

Figure 8.3: An example for a Grid user and a GD Knowledge Entity

9

Controlling the Grid Authorization: a Usage Control Architecture for Data Grids

In this Chapter, we define two software systems architectures: the first is a concrete architecture, the second an abstract one. The concrete architecture outlines the different software elements of a usage control system for Data Grids. The abstract architecture provides the KAOS agent and operation models view of the UCON PDP component of the concrete architecture. The reason we propose an abstract specification is because by doing so we can abstract some important aspects and do the validation using them. In fact, this abstract architecture will be validated in the next Chapter. This way, we will show that our architecture indeed enforce UCON policies; such a validation would be too complicate in the case of a concrete architecture, since there are too much details important for implementation, but not necessary for the validation.

A good portion of the work reported in this Chapter is going to be published in [92].

9.1 A Concrete Data Grid Usage Control Architecture

In this Section, we use notions extracted from the OGF's OGSA authorization working group presented in Section 4.4 to introduce a concrete architecture of a usage control system for Data Grids. The proposed architecture is an implementation of the *Distributed Usage Control Model* (D-UCM) as applied to Data Grid that we introduced in Section 7.2 and with the Figure 7.4.

Figures 9.1 and 9.2 show the architecture. Each usage control step of the D-UCM uses the authorization functional components defined by OGSA. Both the Figures show exactly the same architecture, but while 9.1 uses the standard UML component diagram, 9.2 is instead drawed using the "box and lines" style. We show both because we feel that non-UML expert readers would find the second Figure easier to read and understand.

The architecture encompasses the extended "Grid authentication" and the "Grid Authorization" steps we explained early in this theses in Section

4.1. The "Grid authentication" step has been explained with good details in Section 8. This Section encompasses both, but we concentrate mainly on the "Grid Authorization" step, which, as will become clearer, is an UCON policy-based framework.

Policy-based security mechanisms adopt an almost standard terminology when defining authorization architectures, which distinguishes between different kinds of *Policy Points*. In Section 3.4 we already explained them. However, since usage control extends access control in many ways, we provide more UCON-specific definitions of the software elements of Figures 9.1 and 9.2:

- an **UCON Policy Decision Point (UCON PDP)** is the component responsible for returning an usage decision given the user's usage request (i.e. the right requested), the user's valid attributes, the object's valid attributes, and the satisfaction of authorizations, obligations and conditions predicates. A UCON PDP should be capable to interpret, i.e. enforce, policies pertaining to the UCON_{abc} usage control framework.

- An **UCON Policy Enforcement Point (UCON PEP)** enforces rights returned from an UCON PDP: while UCON PDPs interpret UCON_{abc} policies, UCON PEPs enforce the results coming from an UCON_{abc} policy interpretation (enforcement) process. PEPs enforce rights and right revocations translating these ones in an application-dependent format.

- An **UCON Policy Information Point (UCON PIP)** is a repository for attribute values, both mutable and immutable. Attributes are used for the evaluation of UCON policies

- An **UCON Policy Administration Point (UCON PAP)** is a repository where UCON policies are stored and managed.

- A **Context Handler (CH)** is responsible for handling the communications between PEPs, PDPs and possible PIPs.

- A **System Information/Accounting Service (SIAS)** acts as a source for system attributes. This component, in addition to the immutable attributes and the meta-data repositories, acts as a source of information for the PDP.

These definitions highlight a common confusion with the term "enforcement". From now on, within this thesis, the terms *policy interpretation* and *policy enforcement* are used in an interchangeable way.

As can be seen within the Figures, a *Client Service* is an access requestor (normally, a Grid User) that pushes the credentials obtained from a VO

CIS, the Credential Issuing Service explained in Section 4.4, to a Data Grid Service. As explained in Section 8, a good example of a VO CIS is VOMS. Using the terminology introduced in Section 2.4.1, a Data Grid Service could be either a DGMS (when performing a Grid access) or a SE (when perfoming a Direct Access).

DGMS and SE are clients to a *super-PEP* software element, which can communicate with the CHs (OGSA's Context Handler, again see Section 4.4) located at the DGMS and SEs. Each CH obtains valid attributes from the CVS (OGSA's Credential Validation Service). Then, the local UCON PDP is interrogated for an usage decision. From a UCON point of view, valid attributes released by a CVS are examples of *immutable* (persistent) attributes.

The *super-PEP* is the software element responsible for performing both the usage control steps requested. Among the possible solutions for this element, a centralized service or a collaborative one. For instance, one could consider POLPA [61], a policy language suitable for expressing sequence of actions as well as conjunctions and disjunctions of such sequences. These policies could be useful to *orchestrate* other usage control steps in a workflow (as well as to model single access actions in a usage control step). A possible initial solution in this line of thought is envisaged in [7]. Due to the fact that a super-PEP may be located at DGMS or at SE level, we consider it as a mobile agent.

A complex UCON PDP should be able to evaluate policies where the predicates are statements about the subjects' and objects' attributes. Five software elements make up the UCON PDP:

- the **Reference Monitor** (RM) is a gateway for all the authorization decisions. Referring to the actions shown in Figure 5.2, the RM can receive `TryAccess` and `EndAccess` invocations, and is responsible for issuing the `PermitAccess`, `DenyAccess` or `RevokeAccess` operations.

- The **Authorization Predicate Validator** (PV) takes care of validating the authorization policy predicates. It can perform the `AuthzPredicateValidation` operation.

- the **Obligation Monitor** (OM) checks if subject fulfilled the obligations. It can perform the `ObligationsSat` operation.

- the **Condition Monitor** (CM) takes care of validating the condition policy predicates. It can perform the `CondsPredicateValidation` operation.

- the **Attribute Manager** (AM) updates the UCON mutable attributes and return their values. It can perform the `AttributeUpdate` operation.

In Section 9.2, which follows, each of these software elements is described in an abstract specification using a subset of the KAOS models, which are the agent and operation models.

External components are needed to supply the UCON PDP with the needed information:

- an *UCON policy repository* provides the PDP with the UCON policies to be evaluated;
- a *meta-data repository* provides the PDP with the optional immutable object attributes;
- a *mutable attributes repository* stores the UCON mutable attributes of the subjects and objects;
- the *Grid/SE Accounting SVC* is a System Information/Accounting Service (SIAS) acting as a source for system attributes.

For an access, the PDP collects the immutable subject and object attributes, as well as search for the UCON policies to be enforced. The policy is selected using the access requestor ID (the UCON subject), and the UCON object requested. Mutable subject and object attributes, as well as system attributes, are pulled by the PDP from the mutable attribute repository, and from the Grid accounting service. The updates of mutable subjects' and objects' attributes are performed by the AM sub-component.

For the *data usage control* step, we apply the following restrictions:

- an **UCON subject** is represented by a DGMS user ID, which is the way the access requestor Grid user ID is recognized by the DGMS;
- an **UCON object** is represented by the *abstract name* requested by the DGMS user ID;
- an **UCON right** always follows in one of the fundamental rights categories, which are *view* (read) and *modify* (write), possibly augmented with *creation* and *deletion*;
- **subject attributes** are mutable or persistent security descriptors of the *Client Services* (e.g. the number of data accessed);
- **object attributes** are mutable or persistent security description of the *abstract name* (e.g. the *privacy* level, or the maximum number of contemporary access).

For the *storage usage control* step, we apply the following restrictions:

- an **UCON subject** is an SE user ID, which is the way the access requestor Grid user ID is recognized by the SE;

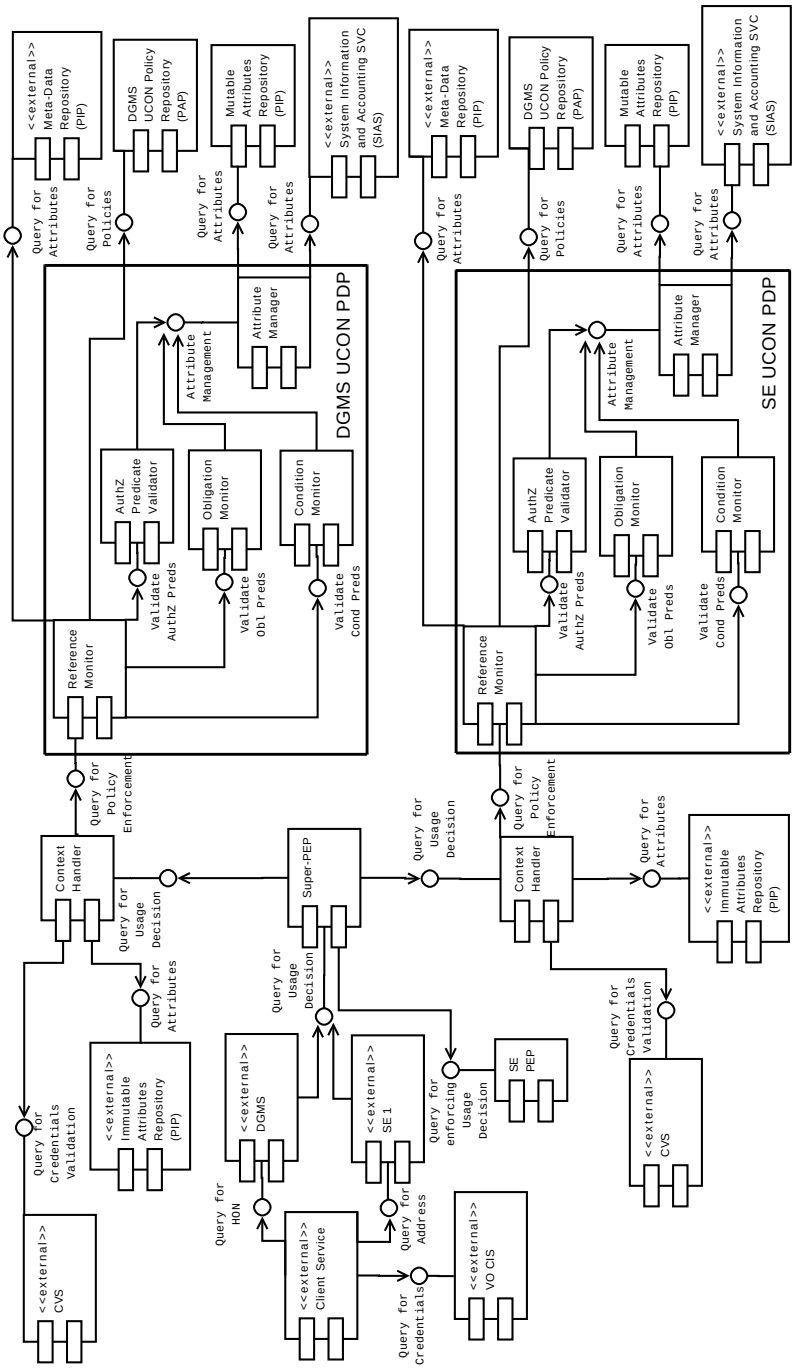

Figure 9.1: Data-Grid usage control architecture (drawed using the UML component model)

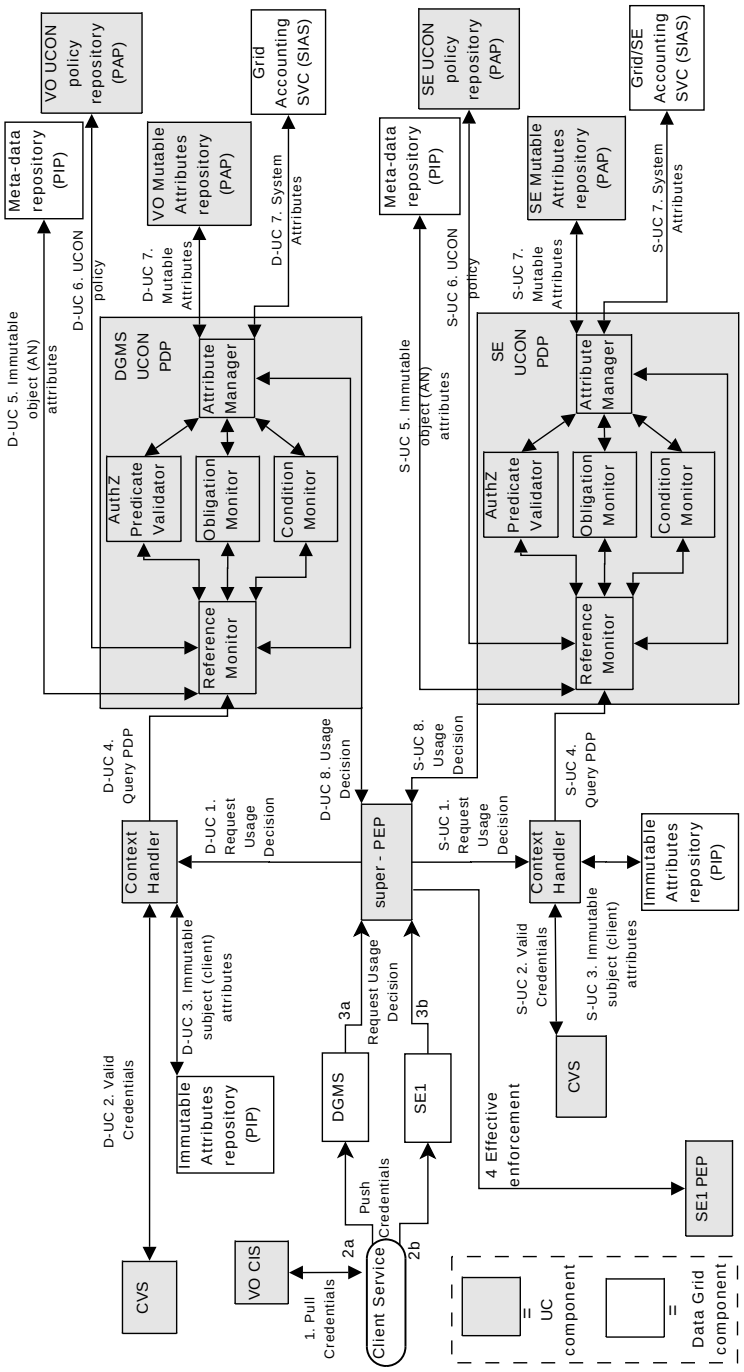

Figure 9.2: Data-Grid usage control architecture (drawed using the "box and lines" style

- an **UCON object** is the GSS where the GD is located;
- an **UCON right** depends from the SE interface implementation in use;
- **subject attributes** are security descriptors of the *Client Services*;
- **object attributes** are security descriptors of the GSS.

9.1.1 Architecture Analysis

Main pros

- The whole architecture is **modular**, **flexible**, and presents a **high capability** level. A number of policy officers are capable of specifying policies pertaining to a vast number core models, and these policies will never overlap. Moreover, each SoA maintain a local authority over its resources, and there's no need for policy syncronization.

Main Cons

- **Complexity.** The proposed architecture has a high degree of complexity. We are aware of the fact that

  > *Complexity is the worst enemy of security.*
  > Bruce Schneier

  There are reasons for such complexity, and simplification possibilities. All the software elements composing the UCON PDP have been recognized as requirements for enforcing UCON_{abc} policies. To do so, we used notions that are partially extracted from the KAOS requirement engineering methodology to produce an abstract specification of all the UCON PDP architectural elements and operations. Such work can be found in Chapter 10, and is also partially available in [93]). We also demonstrated that such specification is capable to enforce all the UCON_{abc} types of policies, as they are formally specified in [111].

  An overall simplification is possible: since UCON is a family of core models, simpler UCON PDPs would enforce not all, but a number of UCON core models. For example, the *Obligation Monitor* component is not necessary if there are no needs for enforcing UCON_b policies.

- **Performance and Trust.** Other big problems may be represented by the performance of an implementation, and by the trust relationships between the sites, but since right now there's not a single complete implementation of the architecture, we leave this problem to future works on the topic.

Issues

- **Policy strategy.** At the end of Section 5.2, we mentioned that an enforcement mechanism for UCON policies should be able to enforce not only the single operations, but the sequence these operations are invoked. In order for an UCON PDP to be an enforcement mechanism for all the UCON core policy models, a way to encode the policy *strategy* (i.e. the sequentiality of the operations) is needed. A possibility lies in the use of an operational policy language like the already cited PoLPA, where the policy specification itself encodes the strategy. Otherwise, an external scheduler can be used for the particular UCON_{abc} sub-model to which the policy pertains.

- **Obligations.** Checking the obligations satisfaction is still an issue. An introductury work on usage control obligations can be found in [81]. We don't plan to solve such issue within this thesis.

We believe this concrete architecture can be of real use for implementors and developers.

9.2 An Abstract Specification of Enforcement Mechanism for Usage Control

We define here an abstract specification of the UCON PDP component defined in the previous Section using the agent and operation models as they are provided by KAOS. The operations are partially inferred from the UCON formal representation defined in [111] and in Section 5.2.2. Figure 9.3 illustrates the abstract version of the UCON PDP components using the KAOS agent/operation models.

We published an initial version of what can be found in this Section in [93].

The *Reference Monitor* (RM) element can receive `TryAccess` and `EndAccess` operation invocations, and is responsible to itself make the `PermitAccess`, `DenyAccess` or `RevokeAccess` operation invocations. The *Authorization Predicate Validator* (APV) element can be invoked for the validation of the authorization predicates by performing the `AuthZPredicateValidation` operation. The *Obligation Monitor* (OM) can be invoked for the validation of the obligation predicates by performing the `OblPredicateValidation` operation. The *Condition Monitor* (CM) can be invoked for validating authorization predicates by performing the `CondPredicateValidation` operation. The *Attribute Manager* (AM) can be invoked for the update of the UCON attributes with the `AttributeUpdate` operation.

Next, we give a written operational software specification of most of the operations shown in Figure 9.3 using the KAOS operation model. `TryAccess`

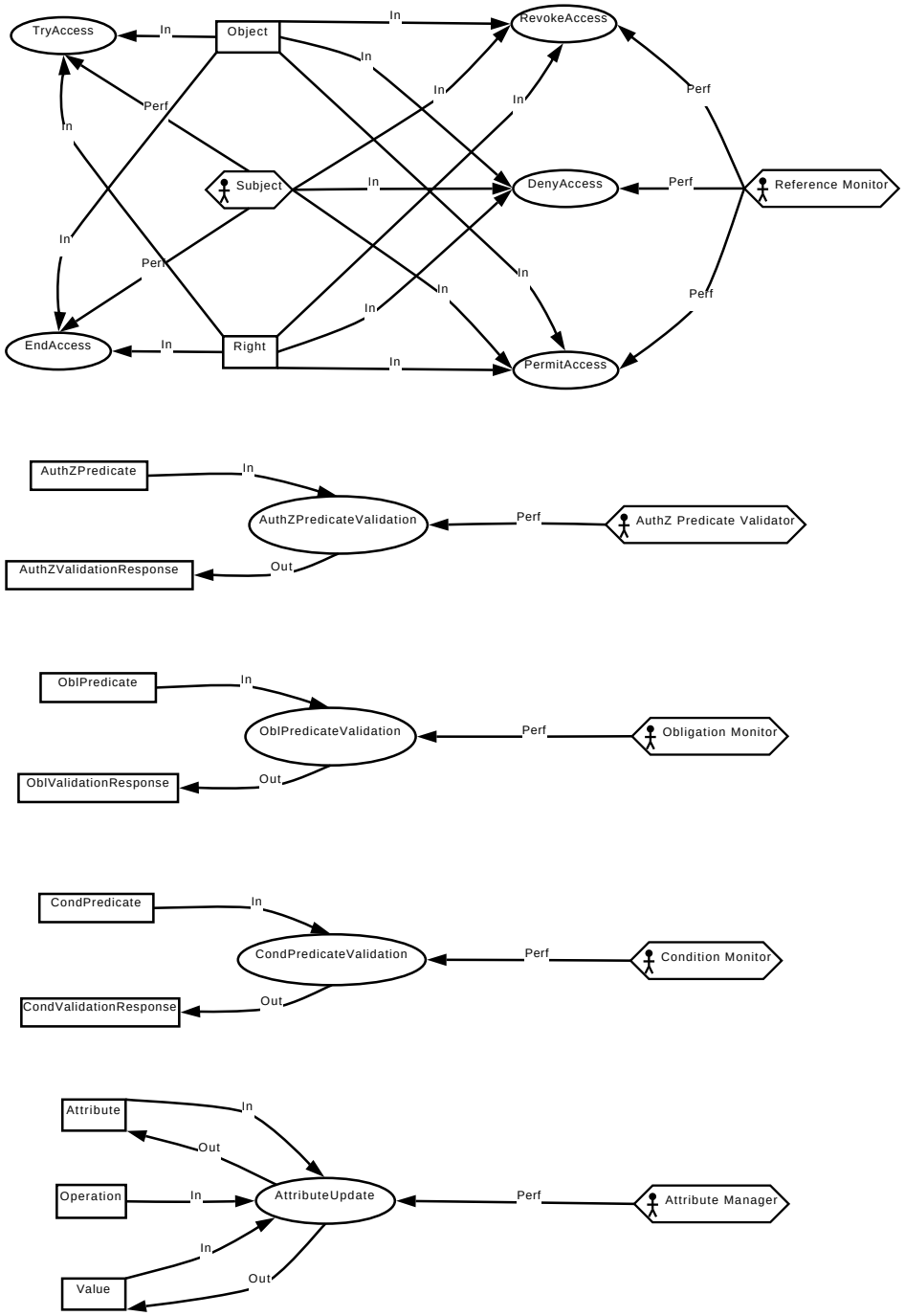

Figure 9.3: Pictorial abstract specification of an UCON$_{abc}$ enforcement mechanism, encoded using the KAOS agent model

and `EndAccess` are not specified since they are issued by the UCON subject (i.e. the *Access Requester*), which we consider as an agent in the environment and hence is not part of the enforcement mechanism. Each operation defines a state-transition in the application domain defined through *domain pre-* and *post-conditions*. Operations have input and output fields; for example, a *subject* (s), an *object* (o) and a *right* (r) are inputs for `PermitAccess`, `DenyAccess` and `RevokeAccess`.

Operation: PermitAccess
Performed By: Reference Monitor
Domain Pre-Condition: true
Domain Post-Condition: RM.permitaccess(s, o, r)
Input: subject, object, right

Operation: DenyAccess
Performed By: Reference Monitor
Domain Pre-Condition: true
Domain Post-Condition: RM.DenyAccess(s, o, r)
Input: subject, object, right
ReqPre-Condition: APV.validate($\neg ap_1 \vee \ldots \vee \neg ap_n$) $\vee$
OM.validate($\neg op_1 \vee \ldots \vee \neg op_n$) $\vee$
CM.validate($\neg cp_1 \vee \ldots \vee \neg cp_n$)

Operation: RevokeAccess
Performed By: Reference Monitor
Domain Pre-Condition: true
Domain Post-Condition: RM.RevokeAccess(s, o, r)
Input: subject, object, right
ReqPre- Condition:
ReqPre-Condition: APV.validate($\neg ap_1 \vee \ldots \vee \neg ap_n$) $\vee$
OM.validate($\neg op_1 \vee \ldots \vee \neg op_n$) $\vee$
CM.validate($\neg cp_1 \vee \ldots \vee \neg cp_n$)

Operation: AuthZPredicateValidation
Performed By: AuthZ Predicate Validator
Domain Pre-Condition: true
Domain Post-Condition: APV.validate($ap_1 \wedge \ldots \wedge ap_n$)
Input: AuthZPredicate
Output: AuthZValidationResponse

Operation: OblPredicateValidation
Performed By: Obligation Monitor
Domain Pre-Condition: true
Domain Post-Condition: OM.validate($\text{op}_1 \wedge \ldots \wedge \text{op}_n$)
Input: OblPredicate
Output: OblValidationResponse

Operation: CondPredicateValidation
Performed By: Condition Monitor
Domain Pre-Condition: true
Domain Post-Condition: CM.validate($\text{cp}_1 \wedge \ldots \wedge \text{cp}_n$)
Input: CondPredicate
Output: CondValidationResponse

Operation: AttributeUpdate
Performed By: Attribute Manager
Domain Pre-Condition: true
Domain Post-Condition: AM.update(s, o, r)
Input: Attribute, Operation, Value
Output: Attribute, Value

Objects of this operations model are *subject, object, right, AuthZPredicate, AuthZValidationResponse, OblPredicate, OblValidationResponse, CondPredicate, CondValidationResponse, Attribute, Operation* and *Value*. None of these objects have attributes.

In KAOS, an important distinction is made between (descriptive) *domain pre-/post-conditions* and (prescriptive) *required pre-*, *post-* and *trigger conditions*. The *required pre-condition* captures a permission to perform the operation only if the condition is true; by contrast, the *required post-condition* defines some additional conditions that any application of the operation must establish. The *required trigger condition* captures an obligation to perform the operation if the condition becomes true provided the *domain precondition* is true. More details on the formal semantics of KAOS operations is provided in [58].

Most of the operations presented above don't specify any *pre-*, *post-* or *trigger conditions*, since these are dependent on the order the single operations are invoked. Such order is encoded in the UCON_{abc} sub-models. For example, in the simplest case of a PreA_0 model, a `PermitAccess` operation can be issued by the RM when the output of the `AuthZPredicateValidation` operation – `AuthZValidationResponse` – is positive. Instead, in a PreA_1

model a `PermitAccess` operation can be issued only after the attributes are updated, i.e. after the AM performed the `AttributeUpdate` operation requested. As we show in Section 5.2.3, since a single policy can be a combination of multiple UCON_{abc} core models, control flow of operations can be even more complicated including an explicit sequentiality of the operations.

Every UCON model encodes a different sequentiality. Within the rest of this paper we will show the KAOS operational specification for all the UCON_{abc} sub-models. We will show that the derived operations always encode the same state-transitions as specified by those in this Section, but since the sequentiality of the single operations is different a model from each other, the *Required Pre-*, *Post-* and *Trigger* Conditions will be model-dependents. We can then be able, for each UCON model, to formally infer a *strategy* to encode the sequentiality of the operations just looking at the *Required Pre-, Post-* and *Trigger* Conditions specified within the operational specification of each UCON_{abc} sub-model. A similar approach was introduced in [9]. As it is said in the former Section, a possibility for the encoding of such *strategy* directly in the policy is the use of an operational policy language like POLPA, where the policy itself encode the strategy. When writing UCON policies using other policy languages, a possibility to encode the strategy is the use of an external scheduler.

In Section 10.2, we shall present our methodology for validating that the abstract operations described above can indeed operationalise, using prescriptive conditions, the policy requirements for all the UCON_{abc} sub-models.

9.3 Related Works

We are not aware of any other in-deep software engineering works on usage control for Grids and Data Grids. Even if there are prototypes following the UCON model, which we will list in Section 11.2, none of them have ever been applied to a Grid environment. That's why, within this Section, we only review some of the security tools used in today's Grids. Some of them had, or still have, a widespread use, other have been used for limited scope. This is by no means a comprehensive list.

9.3.1 The Gridmap File

One of the first attempt to provide authorization in Grid was in the form of the Globus Gridmap File. This file was simply a list of the authorized users, identified by a distinguished name, and the equivalent local user account name they are to be mapped into. This solution is infeasible for the next generation Grids, because the resource owner can not set a policy for who is allowed to do what, and maximize the workload of the resource admin-

istrator who must keep track of all the authorized users. The Gridmap file isn't scalable nor flexible.

9.3.2 The Community Authorization Service

The Globus team developed the Community Authorization Service (CAS) [31]. CAS allows for a separation of concerns between site policies and VO policies. It allows the resource owner to grant access to a portion of his/her resource to a VO. The CAS server acts as a trusted intermediary between VO users and resources: the users first contact the CAS asking for a permission to use a resource, the CAS server consults its policy, and grants or deny the access. The biggest of CAS's problems is that it completely removes control from site administrators, and that it requires a VO to know everything about the configuration of its farms.

9.3.3 G-PBox

One of the most interesting authorization framework is the Grid Policy Box (G-PBox)[14] which can be used for the representation and management of policies for Grid infrastructures. It is an approach for the management of policy repositories hierarchically distributed to independent, administrative-based layers, where each layer contains only policies regarding itself. It follows the classic PEP-PDP interaction, where PDPs/PAPs can be attached to VO, grid site and sub-site layers. G-PBox is composed by a server, which is the PDP, and an administration Graphical User Interface (GUI). The PDP is a XACML (specification version 2.0) engine and Java, C++ and C libraries for PEP communication are supplied. The GUI offers facilities for policy and distribution management, which means that it allows to create/remove/modify/move arbitrary XACML policies and policy sets and to send/receive policies to/from other PBoxes. G-PBox is a tool intended for authorization purposes and it relies on an external AA. In particular, it is already configured with a VOMS plug-in to retrieve informations from VOMS servers.

9.3.4 PERMIS

PERMIS [19, 20] dictates a hierarchical Role Based Access Control (RBAC, see Section 5) through the use of X.509 ACs [27], used to store users' roles and XML-based authorization policies. When a user makes an access request, the PEP authenticates him/her and asks the PDP for an authorization response, which makes a granted or denied decision. A credential validation service (CVS) validates the users' attributes that are to be used by a local PDP. PERMIS requires the Sources of Authority (SoA) to set the policies for every resource they own. A PERMIS policy says who the users are that are covered by the policy, what roles/attributes are supported, who

is allowed to allocate the roles to the users, what resources are covered by the policy, what actions are supported by the resources (e.g. read, write, delete), and what privileges (actions on resources) are granted to each role. PERMIS then checks the X.509 ACs that are possessed by the user, and sees if these conform to the policy, and if they are sufficient to grant the access being requested. Despite PERMIS is a fast-evolving and flexible tool, there are still some issues. The most important one regards the recognition of authority over multiple, hierarchical PDPs: PERMIS recognizes only two different administrative domains, the VO and the target domain, while we believe that Grids need more flexibility, recognizing VO level, site level, farm level and sub-farm level like in G-PBox. On the other hand, we consider the inner flexibility of multiple attribute sources as a risk, and a limit to the usability of the service, since we prefer the traditional grid-way of having just one source of attribute values (roles), which is the VO itself.

9.3.5 GOLD

GOLD, [76] modeled to satisfy the requirements of a chemical VO, implements a simple RBAC extension named task-based access control. Its purpose is to distinguish between roles and their instances using a context information, that's the task being run. Services can use the task to determine whether the requestor has enough permissions to perform operations, or access resources. Access rights can be dynamically activated and deactivated using contracts that define pre- and post-conditions. Evaluating GOLD, we're raising three questions: first of all, is the task information really useful for the security of an e-Science application? Secondly, what's the cost of the maintenance of the contract information? Third, aren't contracts part of the workflow management, rather than security? We think that task-based access control is a bit confusing, since contracts should be handled by a workflow management system, rather than being part of a security infrastructure. Moreover, since only some roles should be able to submit determinate tasks, checking that such task is in action at a resource is useless: simple RBAC can control which processes can run just using the role information. The only real access control happens when a user submits a workflow application claiming the usage of a non-context-related resource. Other problems lie in the non-hierarchical nature of the roles, the lack of the recognition of authority (same problem of PERMIS), and overall the cost of the maintenance of such a system.

9.3.6 The GT4 AuthZ framework

The **GT4 AuthZ framework** [54] is the last attempt by the Globus team to develop an authorization service. It is an implementation of the XACML [74] authorization model, although its master PDP is only responsible for

the coordination of external authorization sources, like PERMIS or old grid-map-files. External AAs like VOMS can provide attributes.

9.3.7 OASIS

OASIS [8] is a dynamic RBAC tool, where roles are completely decentralized and dynamically activated. We think that such a model, besides its great flexibility, is not feasible for a Grid since we pose serious concerns on its scalability.

9.3.8 GPlazma

gPLAZMA [25] is an authorization tool for dCache [35]. It queries **PRIMA** [59] for an authorization decision, while user attributes come from external AAs like VOMS. Its implementation is too much application-specific to be considered as a comparable service.

9.3.9 Local Centre Authorization Service (LCAS)

The **Local Centre Authorization Service** (LCAS) handles authorization requests to the gLite gatekeeper. The authorization decision of the LCAS is based upon the users' certificate and the job specification.

9.3.10 The gLite Java authorization framework (gJAF)

The **gLite Java authorization framework** (gJAF) is an agnostic authorization infrastructure, where hooks are provided for external authorization plug-in modules in order to route authorization requests to external policy systems.

10

Validating Policy Enforcement using a Goal-Oriented Approach

Within this Chapter, we present a formal software engineering methodology which is liberally inspired from KAOS. We then apply the methodology for a validation of the abstract UCON PDP specification we wrote in Section 9.2. The work demonstrates that the specification can enforce policies pertaing to all the UCON models.

10.1 A Methodology for Validating the Enforcement of UCON Policies

We published an initial version of what can be found in this Section and in the following Sections of this Chapter in [93].

Within this Section, we explain how we used some KAOS models for the validation of an abstract specification. This methodology is used for validating the enforcement mechanism of UCON_{abc} policies showed in Section 9.2. With this methodology, we used a goal-oriented approach not only used in the usual top-down manner where goals are refined to system requirements and the latter operationalised, but also in a bottom-up way. Some middleware is assumed to exist and it is abstracted to a KAOS operational specification and then used to validate that it indeed can operationalise system requirements in a correct manner.

We propose a validation methodology defined along the following steps:

- **Step 1:** In this step, we start from the bottom by abstracting a middleware architecture into the KAOS agent/operation models. In our case of the UCON enforcement architecture, this step results in the abstract specification of Section 9.2.

- **Step 2:** Next, we define a UCON family of policies as a top-level KAOS goal. van Lamsweerde proposed in [100] the elicitation of security-related goals using application-specific instantiations of generic specification patterns, like *Confidentiality, Integrity, Availability, etc.*

In other words, the top-level goal of the goal model would consist of the policy the system should enforce. In the case of the UCON_{abc} model, formal policy specifications for all the UCON policy models were defined in [111], which we will try to use as our top-level goals.

- **Step 3:** The top-level goal, i.e. the policy, is then decomposed through a series of refinements following the KAOS refinement approach. In this refinement process, certain completeness, consistency and minimality properties must be respected [56, Section 3.2.4.4]. This step ends when bottom-level system requirements are reached, which model the most low-level refinement of the top-level policy. Section 10.1.1 explains with details this step and the former one.

- **Step 4:** The last step is to validate that the operationalisation of Step 1 is a correct operationalisation. This implies that the operations must be complete, consistent and minimal with respect to the requirements [58]. Section 10.1.2 explains with details this last step.

Step 1 above is similar to classical abstraction methodologies, while Steps 2 and 3 above are inspired by the goal-driven policy refinement approach introduced by Bandara *et al* in [9], itself based on the ideas of [65]. Step 4 is our main contribution, which merges the bottom-up and the top-down approaches.

10.1.1 Goal-based Policy Refinement

Policy refinement concerns with transforming a high-level and abstract policy specification into a low-level and concrete one [65]. It includes:

- determining the resources that are needed to satisfy the requirements of a policy;

- translating the high-level policies into operational policies that can be enforced;

- verifying that the lower level policies actually meet the requirements specified by the high-level policy.

Here, we follow the goal-based approach to policy refinement introduced by Bandara *et al* in [9], which is based on KAOS goal-refinement. KAOS is appropriate for this task since it includes a rigurous notation for representing goals and strategies to refine a goal into a set of subgoals. These subgoals imply the parent goal and are more detailed. Goals are refined until they can be operationalised — i.e. effectively enforced — and are assigned to agents (see Section 6.4 and [99]). Goals can be formalised using linear temporal logic (LTL) [101], which is the formal language used to define the semantic of

UCON [111]. Verifications can then be made on goal refinements to ensure that the system meets the goals and that the goal model is well-formed.

A goal refinement is correct if it is complete, consistent, and minimal. A set of goals $\{G_1, G_2, \ldots, G_n\}$ refines a goal G in the domain D if the following conditions hold:

$$\begin{array}{ll} G_1, \ldots, G_n, D \Rightarrow G & \text{(completeness)} \\ G_1, \ldots, G_n, D \not\Rightarrow false & \text{(consistency)} \\ \bigwedge_{j \neq i} G_j, D \not\Rightarrow G \text{ for each } i \in [1..n] & \text{(minimality)} \end{array}$$

We start from a general refinement to justify the need for an enforcing mechanism; considering that:

In UCON$_{abc}$, a subject `s` is permitted to exercise the right `r` on the object `o` if `s` is allowed to have the right `r` on `o`. In LTL this is written:

∀ s:subject, o:object, r:right
□ permitaccess(s, o, r) → allowed(s, o, r)

A right `r` may be granted after the evaluation of an UCON policy, i.e. after the enforcing of an UCON Pre{ABC} policy from an UCON policy enforcing mechanism. Alternatively, a policy could be enforced after the right is granted, but before it is actually ended or revoked: this is the case for UCON On{ABC} policies. When attribute updates are required, a policy enforcement could happen before, during or after the right grant. In any case, when an access is permitted then a policy is somehow enforced:

∀ s:subject, o:object, r:right
□ permitaccess(s, o, r) → enforce(UCON_policy)

Next, we need to clearly define the meaning of the `enforce(UCON_policy)` predicate. Basically, each `UCON_policy` denotes a UCON model as defined by Sandhu. For each UCON$_{abc}$ sub-model, we define when to enforce the policy with respect to the `permitaccess(s,o,r)` action. For example, each policy pertaining to a PreA$_0$, PreB$_0$, PreC$_0$ models need to be enforced only before the access is actually granted. Instead, each policy pertaining to a PreA$_3$ or PreB$_3$ models need to be enforced not only before the access, but also after the end of it. Moreover, each On policy has to be partially enforced during the access period. The subsequent refinements will specify the sequentiality of the actions needed to enforce the policy model. Following this methodology we are capable to derive a precise abstract specification of the service, and to infer a *strategy* for the policy enforcement.

There are several assumptions made in the policy refinement. First, all predicates and actions are computable. Then, each UCON policy is referred

as a set of logical formulae for a single usage process `(s,o,r)`, and the interactions between concurrent usage processes are not captured. We also assume that before an access request is generated, the requesting subject and the target object exist in the system. Another assumption is that the time line is bounded during the life time of a single usage process, viz. the `tryaccess` is always the first action in a single usage process.

10.1.2 Goal Operationalisation

The Operationalization of goals is the last step remaining to apply the methodology of Section 10.1. This step consists in identifying operations relevant to the goals and deriving requirements on operations so that the goals are satisfied. There are two sub-steps. The first is about identifying operations, with their specific state transitions. Only elementary *domain pre-* and *post-conditions* are identified. Such *domain pre-* and *post- conditions* do not ensure the goal from which they are derived, and that's why we need to derive requirements on operations. The identified operations are strengthened with *required pre-, trigger*, and *post conditions*, so that the goals are satisfied. Formal techniques for deriving operations and requirements on operations from goals can be found in [56].

Once goals have been refined into subgoals that are realizable by single agents, the next step of the goal-oriented process consists in deriving the operations that are relevant to the goals, and the requirements on these operations so that the goals are satisfied. Within [56], we found many generic patterns which we re-used.

10.2 Validating the Enforcement Mechanism

Within this section, we use the methodology explained in the former Section to validate the abstract specification as shown in section 9.2 is capable to enforce policies pertaining to all the UCON_{abc} core models.

10.2.1 UCON PreA_0

In the UCON PreA_0 core model, a usage control decision is determined by authorizations before the usage, and there is no attribute update before, during, or after this usage. Discretionary access control (DAC) model with access control list (ACL) can be expressed with a preA_0 policy. A subject attribute is its identity, and an object attribute is an access control list `acl` of pairs `(id, r)`, where `id` is a subjects identity, and `r` is a right with which this subject can access this object. The predicate to be satified is `((s.id, r)` $\in$ `o.acl)`.

We require the policy to be enforced in the state before the access is permitted. The top goal follows. `enforce(UCON\_PreA0)` means that the

enforcement mechanism is capable to enforce a policy pertaining to the UCON PreA_0 family of core models. A similar consideration stands for all the following goal models that can be found in this Chapter.

Goal [PermitPreA0]
RefinedTo: [Permit], [CheckPredicates], [TryToAccess]
FormalDef: (∀ s:subject, o:object, r:right)
permitaccess(s, o, r) ⇒ • enforce(UCON_PreA0)

We then apply a first goal refinement as shown in Figure 10.1, while the formal sub-goals' definitions follow. We can use tools such as the FAUST toolkit [79] to demostrate that the refinement is correct.

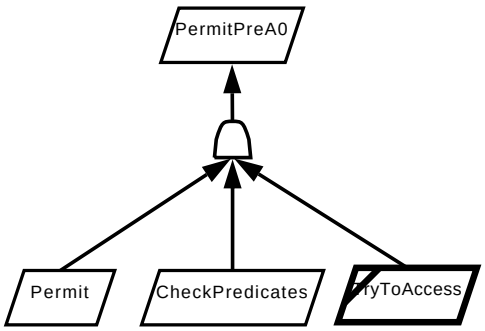

Figure 10.1: Initial goal refinement of an UCON PreA_0 core model

This first refinement, as well as many of those that will follow in the text, are examples of refinements following the milestone pattern.

Goal [Permit]
Refines: [PermitPreA0]
RefinedTo: [Permit Monitor/Control], [PermitToAccess]
FormalDef: (∀ s:subject, o:object, r:right)
permitaccess(s, o, r) ⇒ • ($\text{ap}_1 \wedge \ldots \wedge \text{ap}_n$)

Goal [CheckPredicates]
Refines: [PermitPreA0]
RefinedTo: [CP Monitor/Control], [PredicatesValidation]
FormalDef: (∀ s:subject, o:object, r:right)
($\text{ap}_1 \wedge \ldots \wedge \text{ap}_n$) ⇒ • tryaccess(s, o, r)

Goal [TryToAccess]
Refines: [PermitPreA0]

FormalDef: (∀ s:subject, o:object, r:right)
tryaccess(s, o, r) ⇒ • enforce(UCON_PreA0)

Even if [TryToAccess] is a final goal (an assumption of the system), neither [Permit] nor [CheckPredicates] are final goals, so they have to be refined further. In Figure 10.2 is shown the completion of the goal refinement, and the formal definitions of each of the shown sub-goal follows in the text. We apply accuracy and actuation goals to resolve the lack of monitorability and controllability as suggested in [57]. We identify two requirement goals, [PermitToAccess] and [PredicatesValidation], and assign two agents, the *Reference Monitor* and the *AuthZ Predicate Validator* to respectively take care to each of them.

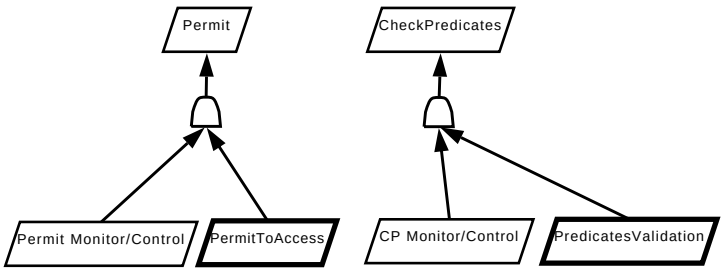

Figure 10.2: Goal model for an UCON PreA_0 core model

Goal [Permit Monitor/Control]
Refines: [Permit]
FormalDef: (∀ s:subject, o:object, r:right,
RM:Reference Monitor, APV:AuthZ Predicate Validator)
permitaccess(s, o, r) ⇔ RM.permitaccess(s, o, r)
$(\text{ap}_1 \wedge \ldots \wedge \text{ap}_n)$ ⇔ APV.validate($\text{ap}_1 \wedge \ldots \wedge \text{ap}_n$)

Goal [PermitToAccess]
Refines: [Permit]
FormalDef: (∀ s:subject, o:object, r:right,
RM:Reference Monitor, APV:Predicate Validator)
RM.permitaccess(s, o, r) ⇒ • APV.validate($\text{ap}_1 \wedge \ldots \wedge \text{ap}_n$)
Resp: Reference Monitor

Goal [CP Monitor/Control]
Refines: [CheckPredicates]
FormalDef: (∀ s:subject, o:object, r:right, APV:Predicate Validator)
$(\text{ap}_1 \wedge \ldots \wedge \text{ap}_n)$ ⇔ APV.validate($\text{ap}_1 \wedge \ldots \wedge \text{ap}_n$)

Goal [PredicatesValidation]
Refines: [CheckPredicates]
FormalDef: (∀ s:subject, o:object, r:right, APV:Predicate Validator)
APV.validate($ap_1 \wedge \ldots \wedge ap_n$) ⇒ • tryaccess(s, o, r)
Resp: AuthZ Predicate Validator

We are now capable to derive the KAOS agent and operation models. Figure 10.3 shows the KAOS operation model, together with the agent/responsibility model. As the reader can see, we identify two operations: `PermitAccess` and `AuthZPredicateValidation`.

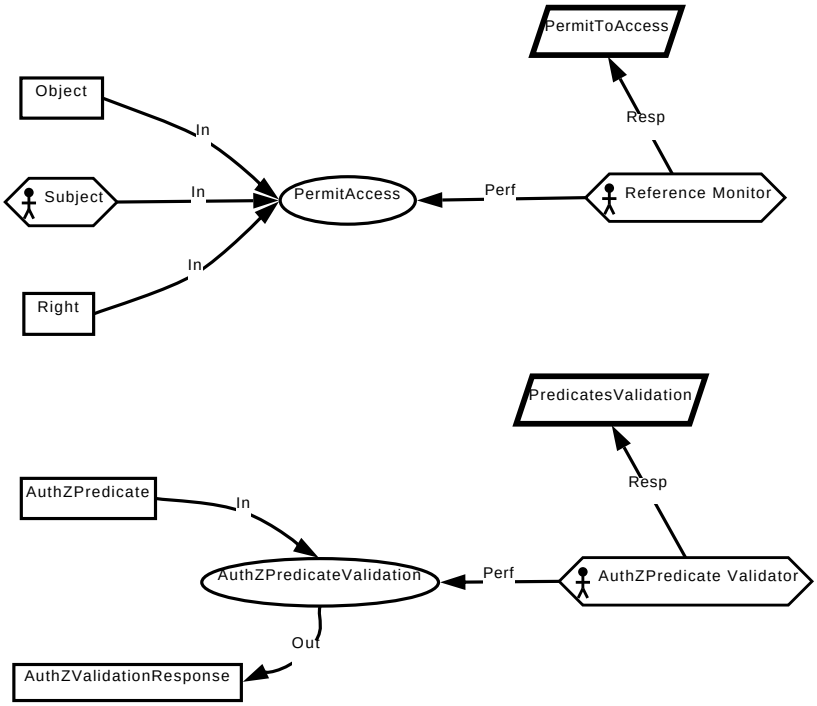

Figure 10.3: Excerpt of the operation model for an UCON $PreA_0$ enforcement mechanism

Next follows the KAOS operational specification for the UCON $PreA_0$ enforcement mechanism, derived using the KAOS operationalization patterns presented in [58]. The semantic of the KAOS operations defines a set of proof obligations verifying that realising an operation when the required *trigger*, *pre-* and *post-* conditions of a goal are true implies the goal. In this sense, a proof of the semantic of each operation in relation to the required conditions validates that enforcement operations implement (i.e. enforce) the corresponding policies.

Operation: PermitAccess

Performed By: Reference Monitor
Domain Pre-Condition: true
Domain Post-Condition: RM.permitaccess(s, o, r)
Input: subject, object, right
ReqPre for [PermitToAccess]: APV.validate($ap_1 \wedge \ldots \wedge ap_n$)

Operation: AuthZPredicateValidation
Performed By: AuthZ Predicate Validator
Domain Pre-Condition: true
Domain Post-Condition: APV.validate($ap_1 \wedge \ldots \wedge ap_n$)
Input: AuthZPredicate
Output: AuthZValidationResponse
ReqPre for [PredicatesValidation]: tryaccess(s, o, r)

The only difference between these operations and those shown in section 9.2 is in the specification of the *Required Pre-Condition* clause. This clause is required to ensure that the goals assigned to the individual agents are met. As we said at the end of Section 9.2, they are dependent from the order of the operations as specified by the model definition.

10.2.2 UCON $PreA_1$

In the UCON $PreA_1$ core model, a usage control decision is determined by authorizations before the usage, and one or more subject or object attributes are updated before this usage. As an example of policy, in a DRM pay-per-use application, a subject has a numerical valued attribute of `credit`, and an object has a numerical valued attribute of `value`. A `read` access can be approved when a subjects credit is more than an objects value. Before the access can start, an update to the subjects credit is performed by the system by subtracting the objects value. This attribute update is a *preUpdate*, and the predicate to be satisfied is, for example, `(Alice.credit` $\geq$ `ebook1.value)`.

What we're showing here is very similar to what is shown in section 10.2.1, and same can be said for all the next paragraph of section 10.2. Since the policy enforcing happens only before the access is permitted, the top goal is the following:

Goal [PermitPreA1]
RefinedTo: [Permit], [Update], [CheckPredicates], [TryToAccess]
FormalDef: ($\forall$ s:subject, o:object, r:right)
permitaccess(s, o, r) $\Rightarrow \bullet$ enforce(UCON_PreA1)

We then apply a first goal refinement as shown in Figure 10.4, while the formal sub-goals' definitions follow in the text.

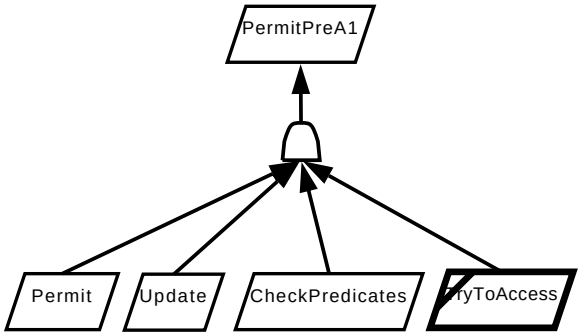

Figure 10.4: Initial goal refinement of an UCON PreA_1 core model

Goal [Permit]
Refines: [PermitPreA1]
RefinedTo: [Permit Monitor/Control], [PermitToAccess]
FormalDef: (∀ s:subject, o:object, r:right)
permitaccess(s, o, r) ⇒ • update(s,o,r)

Goal [Update]
Refines: [PermitPreA1]
RefinedTo: [Update Monitor/Control], [UpdateTheAttributes]
FormalDef: (∀ s:subject, o:object, r:right)
update(s, o, r) ⇒ • ($\text{ap}_1 \wedge \ldots \wedge \text{ap}_n$)

Goal [CheckPredicates]
Refines: [PermitPreA1]
RefinedTo: [CP Monitor/Control], [PredicatesValidation]
FormalDef: (∀ s:subject, o:object, r:right)
($\text{ap}_1 \wedge \ldots \wedge \text{ap}_n$) ⇒ • tryaccess(s, o, r)

Goal [TryToAccess]
Refines: [PermitPreA1]
FormalDef: (∀ s:subject, o:object, r:right)
tryaccess(s, o, r) ⇒ • enforce(UCON_PreA1)

The goals [Update], [Permit] and [CheckPredicates] are not final goals, so they have to be refined further on. In Figure 10.5 is shown the completion

of the goal refinement, and the formal definitions of each sub-goals follow in the text. We apply accuracy and actuation goals, and identify three requirement goals: [PermitToAccess], [UpdateTheAttributes] and [Predicates Validation]. We assign them three agents: the *Reference Monitor* the *Attribute Manager* and the *AuthZ Predicate Validator*.

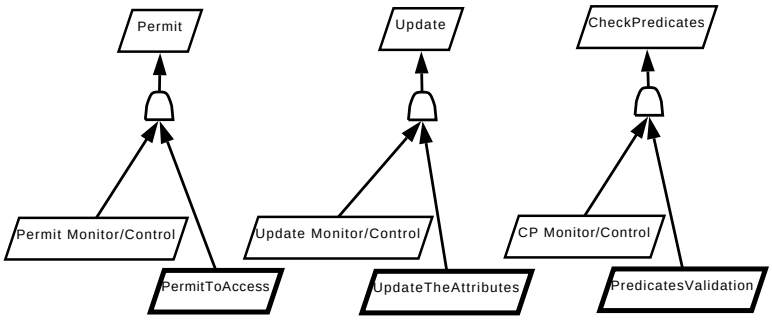

Figure 10.5: Goal model for an UCON PreA_1 core model

Goal [Permit Monitor/Control]
Refines: [Permit]
FormalDef: (∀ s:subject, o:object, r:right,
RM:Reference Monitor, AM:Attribute Manager)
permitaccess(s, o, r) ⇔ RM.permitaccess(s, o, r)
update(s, o, r) ⇔ AM.update(s, o, r)

Goal [PermitToAccess]
Refines: [Permit]
FormalDef: (∀ s:subject, o:object, r:right,
RM:Reference Monitor, AM:Attribute Manager)
RM.permitaccess(s, o, r) ⇒ • AM.update(s, o, r)
Resp: Reference Monitor

Goal [Update Monitor/Control]
Refines: [Update]
FormalDef: (∀ s:subject, o:object, r:right,
AM:Attribute Manager, APV:AuthZ Predicate Validator)
update(s, o, r) ⇔ AM.update(s, o, r)
$(\text{ap}_1 \wedge \ldots \wedge \text{ap}_n)$ ⇔ PV.validate($\text{ap}_1 \wedge \ldots \wedge \text{ap}_n$)

Goal [UpdateTheAttributes]
Refines: [Update]

FormalDef: (∀ s:subject, o:object, r:right,
AM:AttributeManager, APV:AuthZ Predicate Validator)
AM.update(s, o, r) ⇒ • APV.validate($ap_1 \wedge \ldots \wedge ap_n$)
Resp: Attribute Manager

Goal [CP Monitor/Control]
Refines: [CheckPredicates]
FormalDef: (∀ s:subject, o:object, r:right, APV:AuthZ Predicate Validator)
($ap_1 \wedge \ldots \wedge ap_n$) ⇔ APV.validate($ap_1 \wedge \ldots \wedge ap_n$)
tryaccess(s, o, r) ⇔ tryaccess(s, o, r)

Goal [Predicates Validation]
Refines: [CheckPredicates]
FormalDef: (∀ s:subject, o:object, r:right,
APV:AuthZ Predicate Validator)
APV.validate($ap_1 \wedge \ldots \wedge ap_n$)
⇒ • tryaccess(s, o, r)
Resp: AuthZ Predicate Validator

Figure 10.6 shows the KAOS operation model, together with the agent/responsibility model. We identify three operations.

Next follows the formal KAOS operational specification of the UCON $PreA_1$ enforcement mechanism.

Operation: PermitAccess
Performed By: Reference Monitor
Domain Pre-Condition: true
Domain Post-Condition: RM.permitaccess(s, o, r)
Input: subject, object, right
ReqPre for [PermitToAccess]: AM.update(s, o, r)

Operation: AttributeUpdate
Performed By: Attribute Manager
Domain Pre-Condition: true
Domain Post-Condition: AM.update(att)
Input: Attribute, Operation, Value
Output: Attribute, Value
ReqPre for [UpdateTheAttributes]: APV.validate($ap_1 \wedge \ldots \wedge ap_n$)

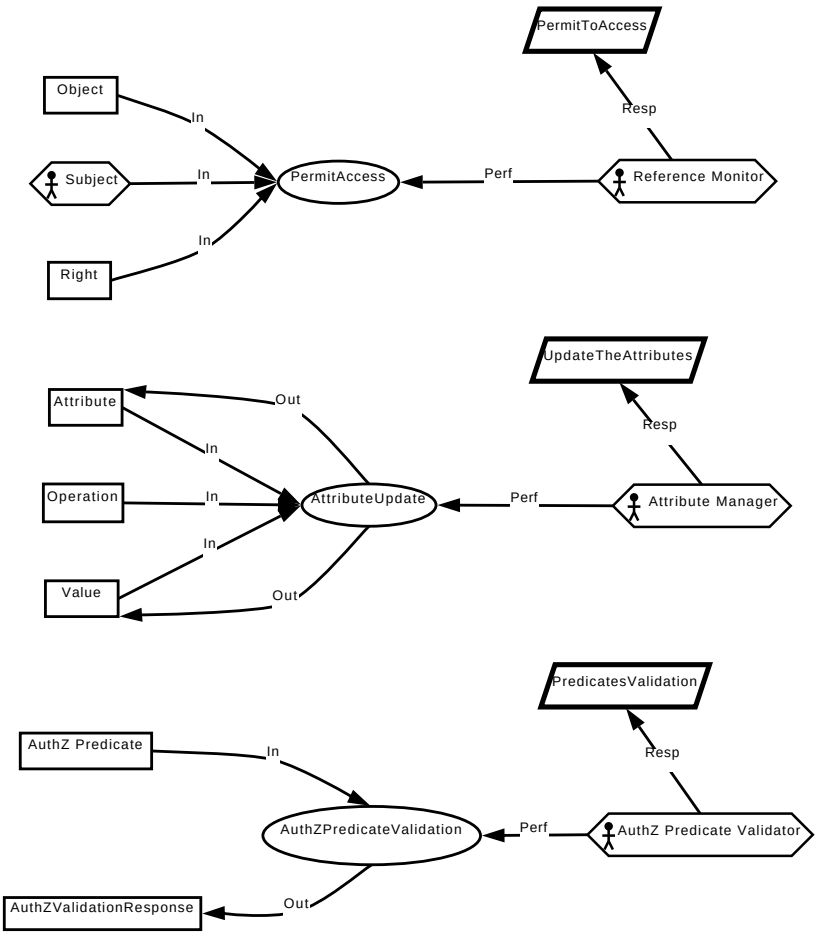

Figure 10.6: Excerpt of the operation model for an UCON PreA_1 enforcement mechanism

Operation: AuthZPredicateValidation
Performed By: AuthZ Predicate Validator
Domain Pre-Condition: true
Domain Post-Condition: APV.validate($p_1 \wedge \ldots \wedge p_n$)
Input: AuthZPredicate
Output: AuthZValidationResponse
ReqPre for [Predicates Validation]: tryaccess(s, o, r)

As in section 10.2.1, the only difference between these operations and those shown in section 9.2 is the specification of the *Required Pre-Condition* clause.

10.2.3 UCON PreA$_3$

In the UCON PreA$_3$ core model, a usage control decision is determined by authorizations before the usage, and one or more subject or object attributes are updated after this usage. An example is: in a DRM membership-based application, a subject `s` has attributes `expense` and `group`, and a file `o` has attributes `group` and `cost`. A subject can read any file in his/her own group. The predicate to be satisfied is `(s.group` = `o.group)`. The expense is updated by adding the `cost` of the file after the access: `s.expense`$'$ = `s.expense` + `o.cost`.

The policy enforcing happens before and after the access is permitted. The top-goal, [PermitPreA3], is easily refined in the goals [PermitPreA3-pre] and [PermitPreA3-post] as specified below.

Goal [PermitPreA3]
RefinedTo: [PermitPreA3-pre], [PermitPreA3-post]
FormalDef: (∀ s:subject, o:object, r:right)
permitaccess(s, o, r) ⇒ • enforce(UCON_PreA3) ∧ ∘ enforce(UCON_PreA3)

Goal [PermitPreA3-pre]
RefinedTo: [Permit], [CheckPredicates], [TryToAccess]
FormalDef: (∀ s:subject, o:object, r:right)
permitaccess(s, o, r) ⇒ • enforce(UCON_PreA3)

Goal [PermitPreA3-post]
RefinedTo: [End], [Update], [PreA3-completed]
FormalDef: (∀ s:subject, o:object, r:right)
permitaccess(s, o, r) ⇒ ∘ enforce(UCON_PreA3)

The first part of the goal refinement is shown in Figure 10.7. The formal sub-goals' definitions follow.

Goal [Permit]
Refines: [PermitPreA3-pre]
RefinedTo: [Permit Monitor/Control], [PermitToAccess]
FormalDef: (∀ s:subject, o:object, r:right)
permitaccess(s, o, r) ⇒ • ($ap_1 \wedge \ldots \wedge ap_n$)

Goal [CheckPredicates]
Refines: [PermitPreA3-pre]
RefinedTo: [CP Monitor/Control], [PredicatesValidation]

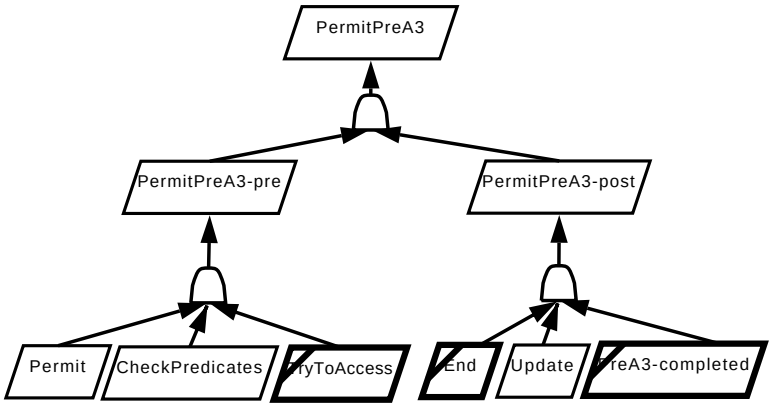

Figure 10.7: Initial goal refinement of an UCON $PreA_3$ core model

FormalDef: (∀ s:subject, o:object, r:right)
($p_1 \wedge \ldots \wedge p_n$) ⇒ • tryaccess(s, o, r)

Goal [TryToAccess]
Refines: [PermitPreA3-pre]
FormalDef: (∀ s:subject, o:object, r:right)
tryaccess(s, o, r) ⇒ • enforce(UCON_PreA3)

Goal [End]
Refines: [PermitPreA3-post]
FormalDef: (∀ s:subject, o:object, r:right)
permitaccess(s, o, r) ⇒ ∘ endaccess(s, o, r)

Goal [Update]
Refines: [PermitPreA3-post]
RefinedTo: [Update Monitor/Control], [UpdateAfterEnd]
FormalDef: (∀ s:subject, o:object, r:right)
endaccess(s, o, r) ⇒ ∘ update(s, o, r)

Goal [PreA3-completed]
Refines: [PermitPreA3-post]
FormalDef: (∀ s:subject, o:object, r:right)
update(s, o, r) ⇒ ∘ enforce(UCON_PreA3)

The goals [Permit], [CheckPredicates] and [Update] have to be refined further. In Figure 10.8 is shown the completion of the goal refinement,

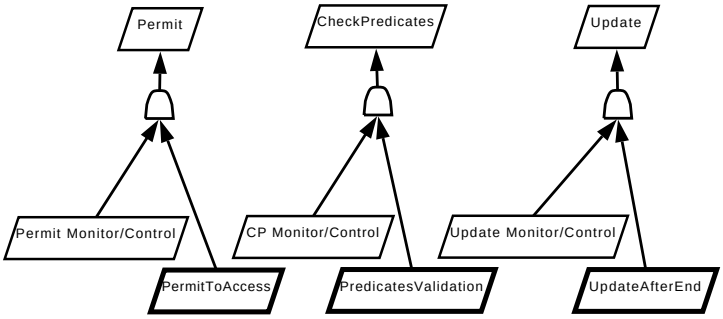

Figure 10.8: Goal model for an UCON PreA$_3$ core model

and the formal definitions of each sub-goals follow in the text. We identify three requirement goals, [PermitToAccess], [PredicatesValidation] and [UpdateAfterEnd], and assign the already known agents *Reference Monitor*, *AuthZ Predicate Validator* and *Attribute Manager* to respectively take care to each of them.

Goal [Permit Monitor/Control]
Refines: [Permit]
FormalDef: (∀ s:subject, o:object, r:right,
RM:Reference Monitor, APV:AuthZ Predicate Validator)
permitaccess(s, o, r) ⇔ RM.permitaccess(s, o, r)
$(ap_1 \wedge \ldots \wedge ap_n) \Leftrightarrow$ APV.validate$(ap_1 \wedge \ldots \wedge ap_n)$

Goal [PermitToAccess]
Refines: [Permit]
FormalDef: (∀ s:subject, o:object, r:right,
RM:Reference Monitor, APV:AuthZ Predicate Validator)
RM.permitaccess(s, o, r) ⇒ • APV.validate$(ap_1 \wedge \ldots \wedge ap_n)$
Resp: Reference Monitor

Goal [CP Monitor/Control]
Refines: [CheckPredicates]
FormalDef: (∀ s:subject, o:object, r:right, APV:AuthZ Predicate Validator)
$(ap_1 \wedge \ldots \wedge ap_n) \Leftrightarrow$ APV.validate$(ap_1 \wedge \ldots \wedge ap_n)$
tryaccess(s, o, r) ⇔ tryaccess(s, o, r)

Goal [PredicatesValidation]
Refines: [CheckPredicates]
FormalDef: (∀ s:subject, o:object, r:right, APV:AuthZ Predicate Validator)
APV.validate$(ap_1 \wedge \ldots \wedge ap_n) \Rightarrow$ • tryaccess(s, o, r)

Resp: AuthZ Predicate Validator

Goal [Update Monitor/Control]
Refines: [Update]
FormalDef: (∀ s:subject, o:object, r:right, AM:Attribute Manager)
update(s, o, r) ⇔ AM.update(s, o, r)

Goal [UpdateAfterEnd]
Refines: [Update]
FormalDef: (∀ s:subject, o:object, r:right, AM:AttributeManager)
endaccess(s, o, r) ⇒ ○ AM.update(s, o, r)
Resp: Attribute Manager

The Figure of the KAOS operation model for this UCON sub-model is not shown, since it's pretty much the same as of Figure 10.6. The formal specifications of the three operations are shown next:

Operation: PermitAccess
Performed By: Reference Monitor
Domain Pre-Condition: true
Domain Post-Condition: RM.permitaccess(s, o, r)
Input: subject, object, right
ReqPre for [PermitToAccess]: APV.validate($ap_1 \wedge \ldots \wedge ap_n$)

Operation: AuthZPredicateValidation
Performed By: AuthZ Predicate Validator
Domain Pre-Condition: true
Domain Post-Condition: APV.validate($ap_1 \wedge \ldots \wedge ap_n$)
Input: AuthZPredicate
Output: AuthZValidationResponse
ReqPre for [PredicatesValidation]: tryaccess(s, o, r)

Operation: AttributeUpdate
Performed By: Attribute Manager
Domain Pre-Condition: true
Domain Post-Condition: AM.update(att)
Input: Attribute, Operation, Value
Output: Attribute, Value
ReqPre for [UpdateAfterEnd]: endaccess(s, o, r)

As previous sections, the only difference between these operations and those shown in section 9.2 is the specification of the *Required Pre-Condition* clause.

10.2.4 UCON OnA_0

In the UCON OnA_0 core model, a usage control decision is determined by authorizations during the usage, and there is no attribute update before, during, or after this usage. The policy enforcing happens after the access is permitted, and before it is ended by the user. The access can be revoked when the predicates are not satisfied. An example of a policy pertaining to a OnA_0 core model is the following: in an VO, a user Bob (with role `employee`) has a temporary position to conduct a short-term project with a certificate of `temp_cert`. While Bob is accessing some sensitive information, his digital certificate (`temp_cert`) for this project is being checked repeatedly. If his certificate number is in the `Certification Revocation List (CRL)` of the VO, his temporary role membership is revoked and he cannot access the information any more. There are no attribute updates, and the predicate to be satisfied is simply `temp_cert` $\in$ `CRL`.

The top goals is following.

Goal [PermitOnA0]
FormalDef: ($\forall$ s:subject, o:object, r:right)
permitaccess(s, o, r) $\Rightarrow$ ($ap_1 \wedge \ldots \wedge ap_n$)
$\wedge \Diamond_{\leq d}$ (revokeaccess(s, o, r) $\vee$ endaccess(s, o, r))

Where d is a finite time slice, that is necessary since unbounded achieve goals are not realizable. The goal refinement is shown in Figure 10.9.

We now apply accuracy and actuation goals (not shown for brevity) to resolve the lack of monitorability and controllability, identifying three requirement goals, [PermitToAccess], [Revoke], and [end] and assign two agents, the *Reference Monitor* and the *AuthZ Predicate Validator* to respectively take care to each of them.

Goal [OnPermit]
FormalDef: ($\forall$ s:subject, o:object, r:right)
RM.permitaccess(s, o, r) $\Rightarrow$ APV.validate($ap_1 \wedge \ldots \wedge ap_n$)
Resp: Reference Monitor

Goal [EventualRevokeOrEnd]
FormalDef: ($\forall$ s:subject, o:object, r:right)
permitaccess(s, o, r) $\Rightarrow \Diamond_{\leq d}$ (revokeaccess(s, o, r) $\vee$ endaccess(s, o, r)
Resp: Reference Monitor

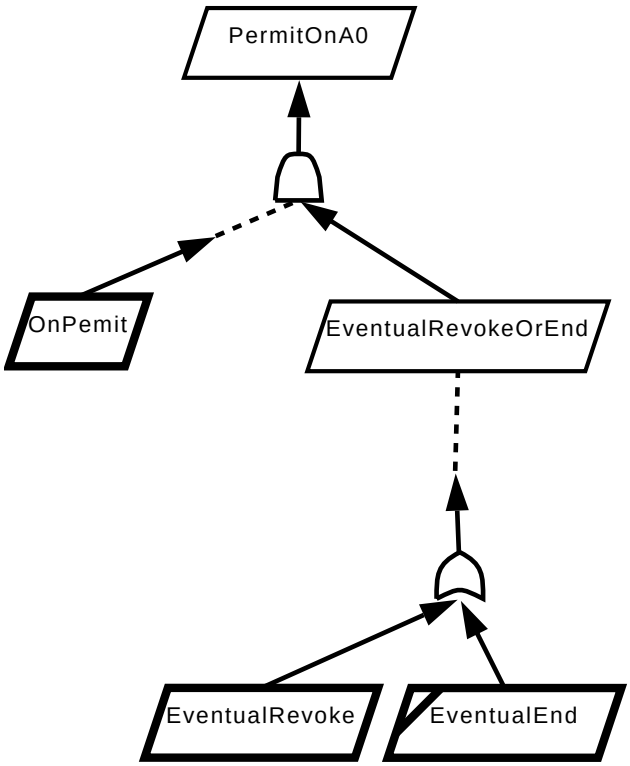

Figure 10.9: Goal refinement of an UCON OnA_0 core model

Goal [EventualRevoke]
FormalDef: (∀ s:subject, o:object, r:right)
RM.permitaccess(s, o, r) ⇒ $\Diamond_{\leq d}$ RM.revokeaccess(s, o, r)
Resp: Reference Monitor

Goal [EventualEnd]
FormalDef: (∀ s:subject, o:object, r:right)
RM.permitaccess(s, o, r) ⇒ $\Diamond_{\leq d}$ RM.endaccess(s, o, r)

The KAOS operational specifications for the UCON OnA_0 enforcement mechanism follow in the text. The only difference between these operations and their equivalent shown in section 9.2 is the specification of the *Required Pre* and *Post-Condition* clauses. The `RevokeAccess` operation will be shown in Section 10.2.12

Operation: PermitAccess
Performed By: Reference Monitor
Domain Pre-Condition: true

Domain Post-Condition: RM.permitaccess(s, o, r)
Input: subject, object, right
ReqPost for [OnPermit]: APV.validate($\text{ap}_1 \wedge \ldots \wedge \text{ap}_n$)

10.2.5 UCON OnA_1

In the UCON OnA_1 core model, a usage control decision is determined by authorizations during the usage, and there is one or more attribute updates before this usage. An example of policy could be similar to the one of section 10.2.4, with the further constraint that Bob can not access more than `MAX_files` at the same time, with the number of current accessed file stored in the `accessed_files` attribute. The predicate to be satisfied is the following: `(accessed_files` $\leq$ `MAX_files)`, with the *preUpdate* `s.accessed_files`$'$ = `s.accessed_files` +1.

The top goals is following.

Goal [PermitOnA1]
FormalDef: ($\forall$ s:subject, o:object, r:right)
permitaccess(s, o, r) $\Rightarrow \bullet$ enforce(UCON_OnA1)
$\wedge$ ($\text{ap}_1 \wedge \ldots \wedge \text{ap}_n$)
$\wedge \Diamond_{\leq d}$ (revokeaccess(s, o, r) $\vee$ endaccess(s, o, r))

Where d is a finite time slice, that is necessary since unbounded achieve goals are not realizable. The goal refinement is shown in Figure 10.10.

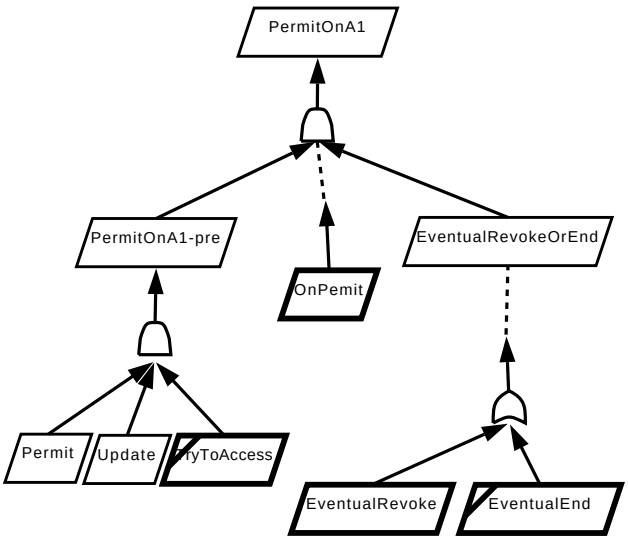

Figure 10.10: Goal refinement of an UCON OnA_1 core model

We also apply accuracy and actuation goals, which we don't show here, to resolve the lack of monitorability and controllability, identifying three

requirement goals, [onPermit], [EventualRevoke], and [UpdateBeforePermit] and assign two agents, the *Reference Monitor* and the *Attribute Manager* to take care of them.

Goal [PermitOnA1-pre]
RefinedTo: [Permit], [Update], [TryToAccess]
FormalDef: (∀ s:subject, o:object, r:right)
permitaccess(s, o, r) ⇒ • enforce(UCON_OnA1)

Goal [OnPermit]
FormalDef: (∀ s:subject, o:object, r:right)
RM.permitaccess(s, o, r) ⇒ APV.validate($ap_1 \wedge \ldots \wedge ap_n$)
Resp: Reference Monitor

Goal [EventualRevokeOrEnd]
FormalDef: (∀ s:subject, o:object, r:right)
permitaccess(s, o, r) ⇒ $\Diamond_{\leq d}$ (revokeaccess(s, o, r) ∨ endaccess(s, o, r)
Resp: Reference Monitor

Goal [EventualRevoke]
FormalDef: (∀ s:subject, o:object, r:right)
RM.permitaccess(s, o, r) ⇒ $\Diamond_{\leq d}$ RM.revokeaccess(s, o, r)
Resp: Reference Monitor

Goal [EventualEnd]
FormalDef: (∀ s:subject, o:object, r:right)
RM.permitaccess(s, o, r) ⇒ $\Diamond_{\leq d}$ RM.endaccess(s, o, r)

The second part of the goal refinement can be found in 10.11, while the formal sub-goals' definitions follow.

Goal [Permit]
Refines: [PermitOnA1-pre]
RefinedTo: [Permit Monitor/Control], [PermitToAccess]
FormalDef: (∀ s:subject, o:object, r:right)
permitaccess(s, o, r) ⇒ • update(s, o, r)

Goal [Update]
Refines: [PermitOnA1-pre]

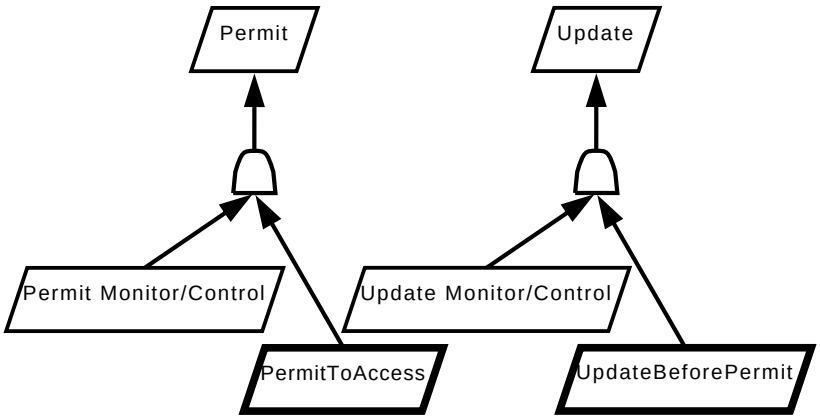

Figure 10.11: Completion goal refinement of an UCON OnA_1 core model

RefinedTo: [Update Monitor/Control], [UpdateBeforePermit]
FormalDef: (∀ s:subject, o:object, r:right)
update(s, o, r) ⇒ • tryaccess(s, o, r)

Goal [TryToAccess]
Refines: [PermitOnA1-pre]
FormalDef: (∀ s:subject, o:object, r:right)
tryaccess(s, o, r) ⇒ • enforce(UCON_OnA1)

Goal [PermitToAccess]
Refines: [Permit]
FormalDef: (∀ s:subject, o:object, r:right,
RM:Reference Monitor, AM:Attribute Manager)
RM.permitaccess(s, o, r) ⇒ • AM.update(s, o, r)
Resp: Reference Monitor

Goal [UpdateBeforePermit]
Refines: [Update]
FormalDef: (∀ s:subject, o:object, r:right, AM:Attribute Manager)
AM.update(s, o, r) ⇒ • tryaccess(s, o, r)
Resp: Attribute Manager

The KAOS operational specifications for the UCON OnA_1 enforcement mechanism follow in the text. The only difference between these operations and their equivalent shown in section 9.2 is the specification of the *Required Pre* and *Post-Condition* clauses. The `RevokeAccess` operation will be shown in Section 10.2.12

Operation: PermitAccess
Performed By: Reference Monitor
Domain Pre-Condition: true
Domain Post-Condition: RM.permitaccess(s, o, r)
Input: subject, object, right
ReqPre for [PermitToAccess]: AM.update(s, o, r)
ReqPost for [OnPermit]: APV.validate($ap_1 \wedge \ldots \wedge ap_n$)

Operation: AttributeUpdate
Performed By: Attribute Manager
Domain Pre-Condition: $\neg$ AM.update(s, o, r)
Domain Post-Condition: AM.update(s, o, r)
Input: subject, object, right
ReqPre for [UpdateBeforePermit]: tryaccess(s, o, r)

10.2.6 UCON OnA_2

In the UCON OnA_2 core model, a usage control decision is determined by authorizations during the usage, and there is one or more attribute updates during this usage. The policy enforcing happens before, during and after the access is permitted. The top goal is:

The top goals is following.

Goal [PermitOnA2]
FormalDef: ($\forall$ s:subject, o:object, r:right)
permitaccess(s, o, r) $\Rightarrow$ ($ap_1 \wedge \ldots \wedge ap_n$)
$\wedge \Diamond_{\leq c}$ update(s, o, r)
$\wedge \Diamond_{\leq d}$ (revokeaccess(s, o, r) $\vee$ endaccess(s, o, r))

Where d and e are finite time slices, that are necessary since unbounded achieve goals are not realizable, with $c < d$. The goal refinement is shown in Figure 10.13.

We also apply accuracy and actuation goals, which we don't show here, to resolve the lack of monitorability and controllability, identifying three requirement goals, [OnPermit], [EventualRevoke] and [OnUpdate] and assign two agents, the *Reference Monitor* and the *Attribute Manager* to take care of them.

Goal [OnPermit]
FormalDef: ($\forall$ s:subject, o:object, r:right)
RM.permitaccess(s, o, r) $\Rightarrow$ APV.validate($ap_1 \wedge \ldots \wedge ap_n$)
Resp: Reference Monitor

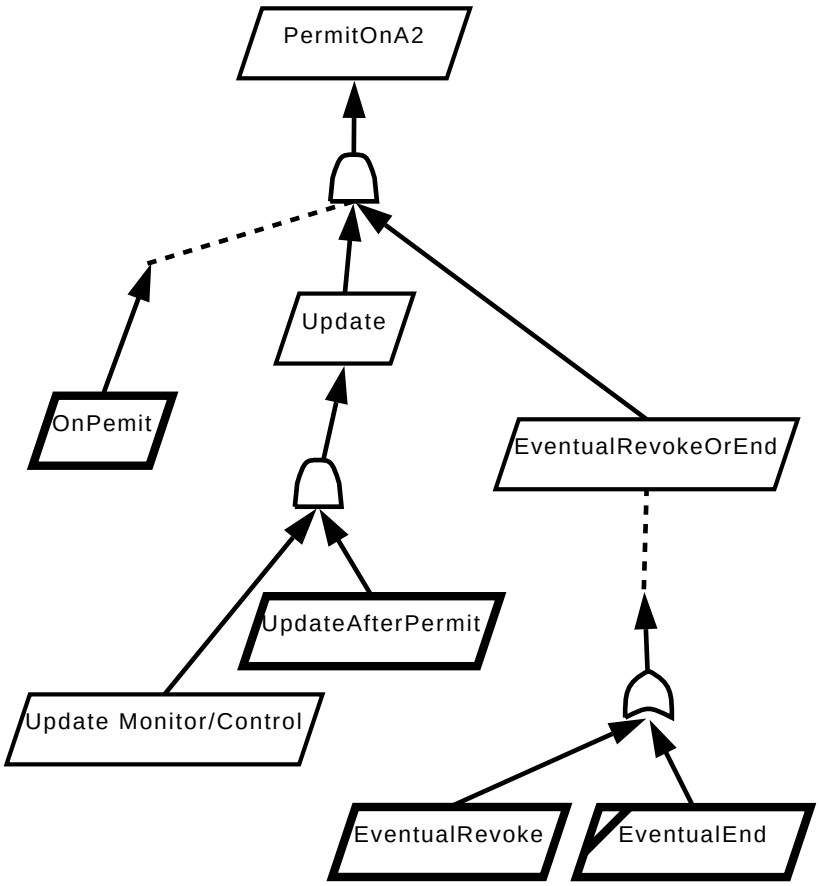

Figure 10.12: Goal refinement of an UCON OnA_2 core model

Goal [Update]
Refines: [PermitOnA2]
RefinedTo: [Update Monitor/Control], [UpdateAfterPermit]
FormalDef: (∀ s:subject, o:object, r:right)
permitaccess(s, o, r) ⇒ ∘ update(s, o, r)

Goal [EventualRevokeOrEnd]
FormalDef: (∀ s:subject, o:object, r:right)
permitaccess(s, o, r) ⇒ $\Diamond_{\leq d}$ (revokeaccess(s, o, r) ∨ endaccess(s, o, r)
Resp: Reference Monitor

Goal [PermitOnA3-post]
RefinedTo: [EndOrRevoke], [Update], [OnUpdateCompleted]
FormalDef: (∀ s:subject, o:object, r:right)
permitaccess(s, o, r) ⇒ ◊ enforce(UCON_OnA3)

Goal [EventualRevoke]
FormalDef: (∀ s:subject, o:object, r:right)
RM.permitaccess(s, o, r) ⇒ $\Diamond_{\leq d}$ RM.revokeaccess(s, o, r)
Resp: Reference Monitor

Goal [EventualEnd]
FormalDef: (∀ s:subject, o:object, r:right)
RM.permitaccess(s, o, r) ⇒ $\Diamond_{\leq d}$ RM.endaccess(s, o, r)

Goal [Update Monitor/Control]
Refines: [Update]
FormalDef: (∀ s:subject, o:object, r:right,
RM:Reference Monitor, AM:Attribute Manager)
permitaccess(s, o, r) ⇔ RM.permitaccess(s, o, r)
update(s, o, r) ⇔ AM.update(s, o, r)

Goal [UpdateAfterPermit]
Refines: [Update]
FormalDef: (∀ s:subject, o:object, r:right,
RM:Reference Monitor, AM:Attribute Manager)
RM.permitaccess(s, o, r) ⇒ ∘ AM.update(s, o, r)
Resp: Attribute Manager

The KAOS operational specifications for the UCON OnA_0 enforcement mechanism follow in the text. The only difference between these operations and their equivalent shown in section 9.2 is the specification of the *Required Pre* and *Post-Condition* clauses. The `RevokeAccess` operation will be shown in Section 10.2.12

Operation: PermitAccess
Performed By: Reference Monitor
Domain Pre-Condition: true
Domain Post-Condition: RM.permitaccess(s, o, r)
Input: subject, object, right
ReqPost for [OnPermit]: APV.validate($ap_1 \wedge \ldots \wedge ap_n$)

Operation: AttributeUpdate
Performed By: Attribute Manager

Domain Pre-Condition: true
Domain Post-Condition: AM.update(s, o, r)
Input: subject, object, right
ReqPre for [UpdateAfterPermit]: RM.permitaccess(s, o, r)

10.2.7 UCON OnA_3

In the UCON OnA_3 core model, a usage control decision is determined by authorizations during the usage, and there is one or more attribute updates after this usage. The example policy at the beginning of section 10.2.5 can be completed with the following *postUpdate*: `s.accessed_files`$' =$ `s.accessed_files` -1.

The top goals is following.

Goal [PermitOnA3]
FormalDef: (∀ s:subject, o:object, r:right)
permitaccess(s, o, r) ⇒ ($ap_1 \wedge \ldots \wedge ap_n$)
$\wedge \Diamond_{\leq d}$ (revokeaccess(s, o, r) ∨ endaccess(s, o, r))
$\wedge \Diamond_{\leq e}$ enforce(UCON_OnA3)

Where d and e are finite time slices, that are necessary since unbounded achieve goals are not realizable, with $e > d$. The goal refinement is shown in Figure 10.13.

We also apply accuracy and actuation goals, which we don't show here, to resolve the lack of monitorability and controllability, identifying three requirement goals, [OnPermit], [EventualRevoke] and [UpdateAfterEnd] and assign two agents, the *Reference Monitor* and the *Attribute Manager* to take care of them.

Goal [OnPermit]
FormalDef: (∀ s:subject, o:object, r:right)
RM.permitaccess(s, o, r) ⇒ APV.validate($ap_1 \wedge \ldots \wedge ap_n$)
Resp: Reference Monitor

Goal [EventualRevokeOrEnd]
FormalDef: (∀ s:subject, o:object, r:right)
permitaccess(s, o, r) ⇒ $\Diamond_{\leq d}$ (revokeaccess(s, o, r) ∨ endaccess(s, o, r)
Resp: Reference Monitor

Goal [PermitOnA3-post]
RefinedTo: [EndOrRevoke], [Update], [OnUpdateCompleted]
FormalDef: (∀ s:subject, o:object, r:right)

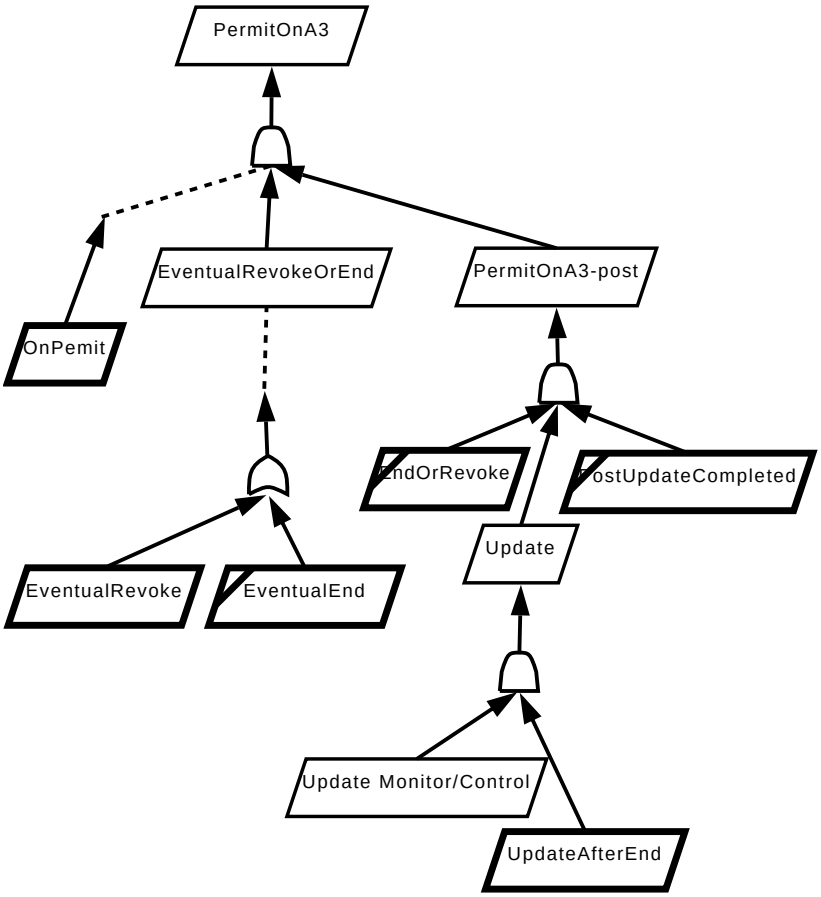

Figure 10.13: Goal refinement of an UCON OnA_3 core model

permitaccess(s, o, r) $\Rightarrow \Diamond$ enforce(UCON_OnA3)

Goal [EventualRevoke]
FormalDef: ($\forall$ s:subject, o:object, r:right)
RM.permitaccess(s, o, r) $\Rightarrow \Diamond_{\leq d}$ RM.revokeaccess(s, o, r)
Resp: Reference Monitor

Goal [EventualEnd]
FormalDef: ($\forall$ s:subject, o:object, r:right)
RM.permitaccess(s, o, r) $\Rightarrow \Diamond_{\leq d}$ RM.endaccess(s, o, r)

Goal [Update]
Refines: [PermitOnA3-post]

RefinedTo: [Update Monitor/Control], [UpdateAfterEnd]
FormalDef: (∀ s:subject, o:object, r:right)
endaccess(s, o, r) ⇒ ◊ update(s, o, r)

Goal [PostUpdateCompleted]
Refines: [PermitOnA3-post]
FormalDef: (∀ s:subject, o:object, r:right)
update(s, o, r) ⇒ ◊ enforce(UCON_OnA3)

Goal [Update Monitor/Control]
Refines: [Update]
FormalDef: (∀ s:subject, o:object, r:right, AM:Attribute Manager)
update(s, o, r) ⇔ AM.update(s, o, r)

Goal [UpdateAfterEnd]
Refines: [Update]
FormalDef: (∀ s:subject, o:object, r:right, AM:Attribute Manager)
endaccess(s, o, r) ⇒ ◊ AM.update(s, o, r)
Resp: Attribute Manager

The KAOS operational specifications for the UCON OnA_3 enforcement mechanism follow in the text. The only difference between these operations and their equivalent shown in section 9.2 is the specification of the *Required Pre* and *Post-Condition* clauses. The `RevokeAccess` operation will be shown in Section 10.2.12

Operation: PermitAccess
Performed By: Reference Monitor
Domain Pre-Condition: true
Domain Post-Condition: RM.permitaccess(s, o, r)
Input: subject, object, right
ReqPost for [OnPermit]: APV.validate($ap_1 \wedge \ldots \wedge ap_n$)

Operation: AttributeUpdate
Performed By: Attribute Manager
Domain Pre-Condition: true
Domain Post-Condition: AM.update(s, o, r)
Input: subject, object, right
ReqPre for [UpdateAfterEnd]: endaccess(s, o, r)

10.2.8 UCON $PreB_0$

In the UCON $PreB_0$ core model, a usage control decision is determined by obligations before the usage, and there is no attribute update before, during, or after this usage. We require the policy to be enforced in the state before the access is permitted. The top goal is then the following:

Goal [PermitPreB0]
RefinedTo: [Permit], [CheckPredicates], [TryToAccess]
FormalDef: (∀ s:subject, o:object, r:right)
permitaccess(s, o, r) ⇒ • enforce(UCON_PreB0)

We then apply a first goal refinement as shown in Figure 10.14, while the formal sub-goals' definitions follow.

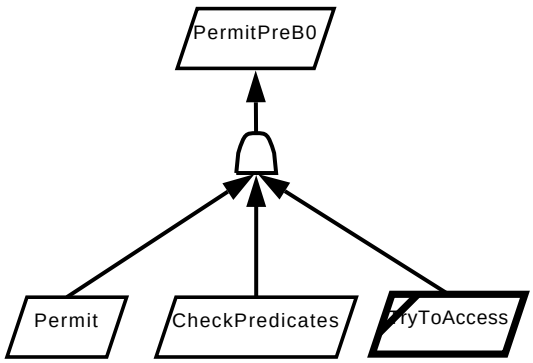

Figure 10.14: Initial goal refinement of an UCON $PreB_0$ core model

Goal [Permit]
Refines: [PermitPreB0]
RefinedTo: [Permit Monitor/Control], [PermitToAccess]
FormalDef: (∀ s:subject, o:object, r:right)
permitaccess(s, o, r) ⇒ • ($op_1 \wedge \ldots \wedge op_n$)

Goal [CheckPredicates]
Refines: [PermitPreB0]
RefinedTo: [CP Monitor/Control], [PredicatesValidation]
FormalDef: (∀ s:subject, o:object, r:right)
($op_1 \wedge \ldots \wedge op_n$) ⇒ • tryaccess(s, o, r)

Goal [TryToAccess]
Refines: [PermitPreB0]

FormalDef: (∀ s:subject, o:object, r:right)
tryaccess(s, o, r) ⇒ • enforce(UCON_PreB0)

Neither [Permit] nor [CheckPredicates] are final goals, so they have to be refined further. In Figure 10.15 is shown the completion of the goal refinement, and the formal definitions of each of the shown sub-goal follows in the text. In a very similar way to what we have done in Section 10.2.1, we apply accuracy and actuation goals, and identify two requirement goals, [PermitToAccess] and [PredicatesValidation], and assign two agents, the *Reference Monitor* and the *Obligation Validator* to respectively take care to each of them.

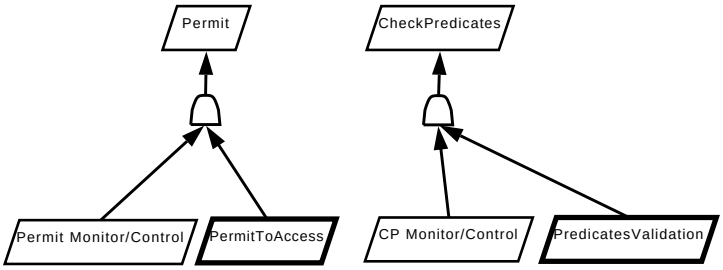

Figure 10.15: Goal model for an UCON PreB_0 core model

Goal [Permit Monitor/Control]
Refines: [Permit]
FormalDef: (∀ s:subject, o:object, r:right,
RM:Reference Monitor, OM:Obligation Monitor)
permitaccess(s, o, r) ⇔ RM.permitaccess(s, o, r)
$(\text{op}_1 \wedge \ldots \wedge \text{op}_n)$ ⇔ OM.validate($\text{op}_1 \wedge \ldots \wedge \text{op}_n$)

Goal [PermitToAccess]
Refines: [Permit]
FormalDef: (∀ s:subject, o:object, r:right,
RM:Reference Monitor, OM:Obligation Monitor)
RM.permitaccess(s, o, r) ⇒ • OM.validate($\text{op}_1 \wedge \ldots \wedge \text{op}_n$)
Resp: Reference Monitor

Goal [CP Monitor/Control]
Refines: [CheckPredicates]
FormalDef: (∀ s:subject, o:object, r:right, OM:Obligation Monitor)
$(\text{op}_1 \wedge \ldots \wedge \text{op}_n)$ ⇔ OM.validate($\text{op}_1 \wedge \ldots \wedge \text{op}_n$)

Goal [PredicatesValidation]
Refines: [CheckPredicates]
FormalDef: (∀ s:subject, o:object, r:right, OM:Obligation Monitor)
OM.validate($op_1 \wedge \ldots \wedge op_n$) ⇒ • tryaccess(s, o, r)
Resp: Obligation Monitor

We are now capable to derive the KAOS agent and operation models. Figure 10.16 shows the KAOS operation model, together with the agent/responsibility model. As the reader can see, we identify two operations: `PermitAccess` and `OblPredicateValidation`.

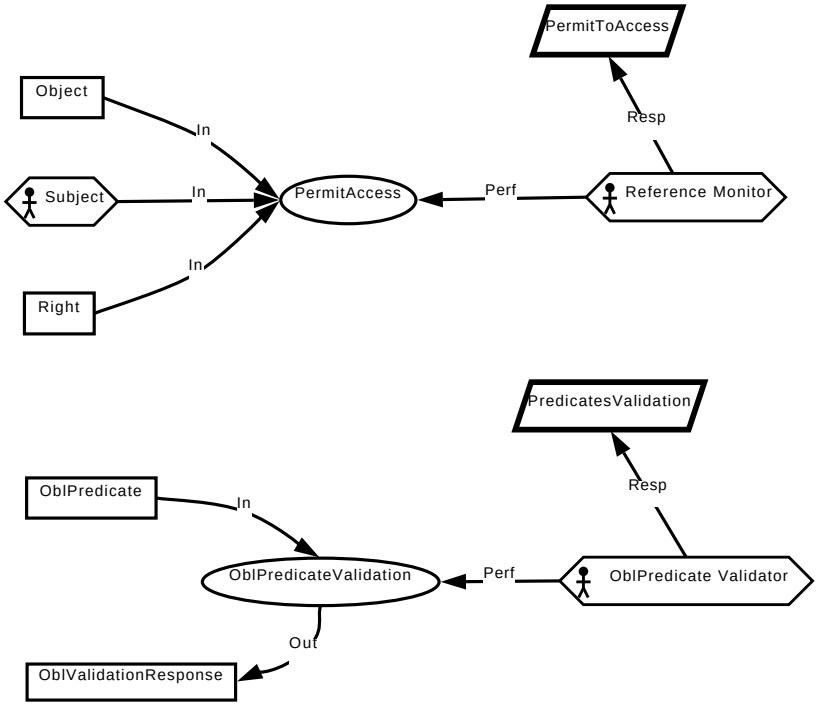

Figure 10.16: Excerpt of the operation model for an UCON $PreB_0$ enforcement mechanism

Next follows the KAOS operational specification for the UCON $PreB_0$ enforcement mechanism, derived using the KAOS operationalization patterns presented in [58].

Operation: PermitAccess
Performed By: Reference Monitor
Domain Pre-Condition: ¬ RM.permitaccess(s, o, r)
Domain Post-Condition: RM.permitaccess(s, o, r)
Input: subject, object, right

ReqPre for [PermitToAccess]: OM.validate($op_1 \wedge \ldots \wedge op_n$)

Operation: OblPredicateValidation
Performed By: Obligation Monitor
Domain Pre-Condition: $\neg$ OM.validate($op_1 \wedge \ldots \wedge op_n$)
Domain Post-Condition: OM.validate($op_1 \wedge \ldots \wedge op_n$)
Input: OblPredicate
Output: OblValidationResponse
ReqPre for [PredicatesValidation]: tryaccess(s, o, r)

As usual, the only difference between these operations and those shown in section 9.2 is in the specification of the *Required Pre-Condition* clause.

10.2.9 UCON $PreB_1$, UCON $PreB_3$, UCON OnB_0, UCON OnB_1, UCON OnB_2, UCON OnB_3

As the reader has already noted, most of the Sections in this Chapter are similar one to another. Especially, the last Section 10.2.8 is very similar to Section 10.2.1: the only differences between the two are in the fact that Section 10.2.1 treats *Authorizations*, while 10.2.8 is all about *Obligations*. A same reasoning can be made about all the other Sections about $UCON_B$: a Section on UCON $PreB_1$ would be similar to Section 10.2.2, the one on $PreB_3$ would be similar to Section 10.2.3, one on OnB_0 to 10.2.4, and so on. That's the reason we don't expand these Sections.

10.2.10 UCON $PreC_0$

In the UCON $PreC_0$ core model, a usage control decision is determined by conditions before the usage, and, as is the case for usage control based on conditions, there is no attribute update before, during, or after this usage. This Section is very similar to Section 10.2.1.

We require the policy to be enforced in the state before the access is permitted. The top goal is then the following:

Goal [PermitPreC0]
RefinedTo: [Permit], [CheckPredicates], [TryToAccess]
FormalDef: ($\forall$ s:subject, o:object, r:right)
permitaccess(s, o, r) $\Rightarrow \bullet$ enforce(UCON_PreC0)

We then apply a first goal refinement as shown in Figure 10.17, while the formal sub-goals' definitions follow.

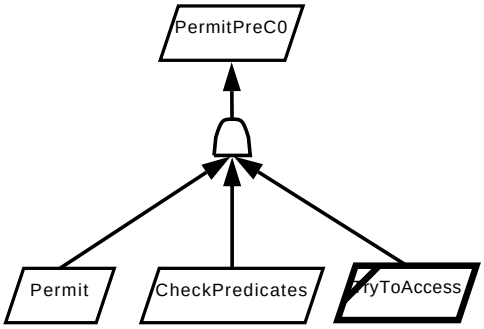

Figure 10.17: Initial goal refinement of an UCON PreC_0 core model

Goal [Permit]
Refines: [PermitPreC0]
RefinedTo: [Permit Monitor/Control], [PermitToAccess]
FormalDef: (∀ s:subject, o:object, r:right)
permitaccess(s, o, r) ⇒ • ($\text{cp}_1 \wedge \ldots \wedge \text{cp}_n$)

Goal [CheckPredicates]
Refines: [PermitPreC0]
RefinedTo: [CP Monitor/Control], [PredicatesValidation]
FormalDef: (∀ s:subject, o:object, r:right)
($\text{cp}_1 \wedge \ldots \wedge \text{cp}_n$) ⇒ • tryaccess(s, o, r)

Goal [TryToAccess]
Refines: [PermitPreA0]
FormalDef: (∀ s:subject, o:object, r:right)
tryaccess(s, o, r) ⇒ • enforce(UCON_PreC0)

[Permit] and [CheckPredicates] are not final goals, so they have to be refined further. In Figure 10.18 is shown the completion of the goal refinement, and the formal definitions of each of the shown sub-goal follows in the text. We identify two requirement goals, [PermitToAccess] and [PredicatesValidation], and assign two agents, the *Reference Monitor* and the *Condition Monitor* to respectively take care to each of them.

Goal [Permit Monitor/Control]
Refines: [Permit]
FormalDef: (∀ s:subject, o:object, r:right,
RM:Reference Monitor, CM:Condition Monitor)
permitaccess(s, o, r) ⇔ RM.permitaccess(s, o, r)

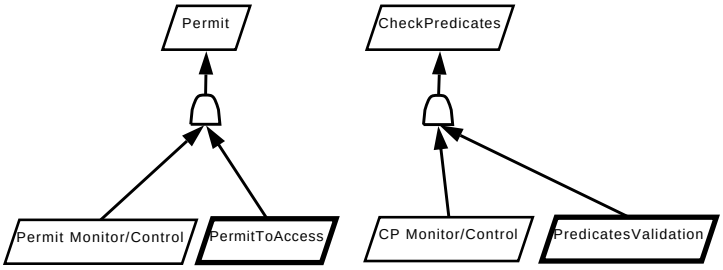

Figure 10.18: Goal model for an UCON PreC_0 core model

$(\text{cp}_1 \wedge \ldots \wedge \text{cp}_n) \Leftrightarrow \text{CM.validate}(\text{cp}_1 \wedge \ldots \wedge \text{cp}_n)$

Goal [PermitToAccess]
Refines: [Permit]
FormalDef: (∀ s:subject, o:object, r:right,
RM:Reference Monitor, CM:Condition Monitor)
$\text{RM.permitaccess(s, o, r)} \Rightarrow \bullet\ \text{CM.validate}(\text{cp}_1 \wedge \ldots \wedge \text{cp}_n)$
Resp: Reference Monitor

Goal [CP Monitor/Control]
Refines: [CheckPredicates]
FormalDef: (∀ s:subject, o:object, r:right, CM:Condition Monitor)
$(\text{cp}_1 \wedge \ldots \wedge \text{cp}_n) \Leftrightarrow \text{CM.validate}(\text{cp}_1 \wedge \ldots \wedge \text{cp}_n)$

Goal [PredicatesValidation]
Refines: [CheckPredicates]
FormalDef: (∀ s:subject, o:object, r:right, CM:Condition Monitor)
$\text{APV.validate}(\text{cp}_1 \wedge \ldots \wedge \text{cp}_n) \Rightarrow \bullet\ \text{tryaccess(s, o, r)}$
Resp: Condition Monitor

Figure 10.19 shows the KAOS operation model, together with the agent/responsibility model. We identify a couple of operations: `PermitAccess` and `CondPredicateValidation`:

Next follows the KAOS operational specification for the UCON PreC_0 enforcement mechanism:

Operation: PermitAccess
Performed By: Reference Monitor
Domain Pre-Condition: true
Domain Post-Condition: RM.permitaccess(s, o, r)

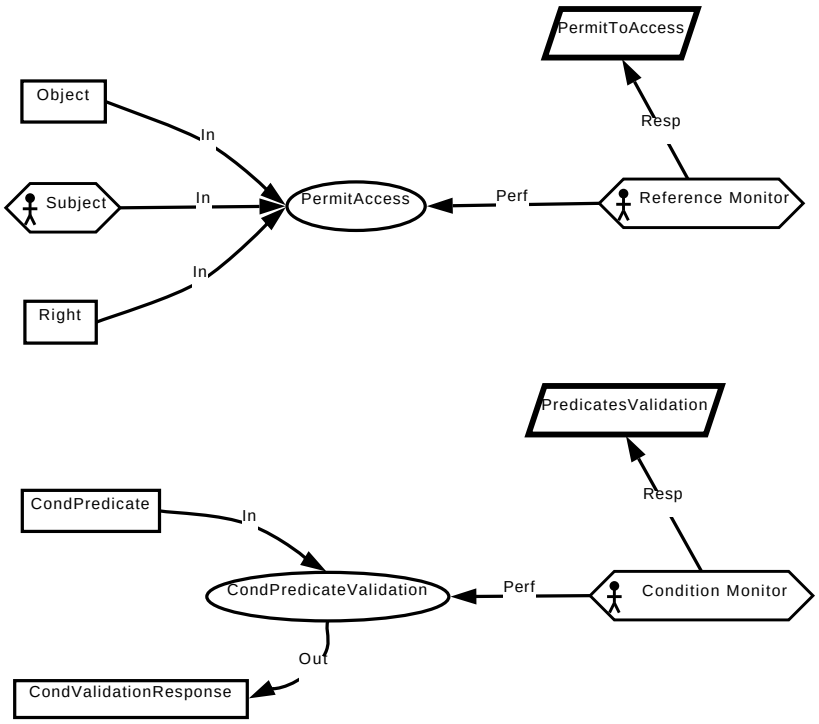

Figure 10.19: Excerpt of the operation model for an UCON PreC_0 enforcement mechanism

Input: subject, object, right
ReqPre for [PermitToAccess]: CM.validate($\text{cp}_1 \wedge \ldots \wedge \text{cp}_n$)

Operation: CondPredicateValidation
Performed By: Condition Monitor
Domain Pre-Condition: true
Domain Post-Condition: CM.validate($\text{cp}_1 \wedge \ldots \wedge \text{cp}_n$)
Input: CondPredicate
Output: CondValidationResponse
ReqPre for [PredicatesValidation]: tryaccess(s, o, r)

The only difference between these operations and those shown in Section 9.2 is in the specification of the *Required Pre-Condition* clause.

10.2.11 UCON OnC_0

In the UCON OnC_0 core model, a usage control decision is determined by conditions during the usage. The policy enforcing happens after the access

is permitted, and before it is ended by the user. The access can be revoked when the predicates are not satisfied. The top goals follow:

The top goals is following.

Goal [PermitOnC0]
FormalDef: (∀ s:subject, o:object, r:right)
permitaccess(s, o, r) ⇒ ($ap_1 \wedge \ldots \wedge ap_n$)
$\wedge \Diamond_{\leq d}$ (revokeaccess(s, o, r) ∨ endaccess(s, o, r))

Where d is a finite time slice, that is necessary since unbounded achieve goals are not realizable. The goal refinement is shown in Figure 10.20.

Figure 10.20: Goal refinement of an UCON OnC_0 core model

We now apply accuracy and actuation goals (not shown for brevity) to resolve the lack of monitorability and controllability, identifying two requirement goals, [PermitToAccess] and [EventualRevoke], and assign an agent, the *Reference Monitor* to take care of them.

Goal [OnPermit]
FormalDef: (∀ s:subject, o:object, r:right)
RM.permitaccess(s, o, r) ⇒ CM.validate($cp_1 \wedge \ldots \wedge cp_n$)
Resp: Reference Monitor

Goal [EventualRevokeOrEnd]
FormalDef: (∀ s:subject, o:object, r:right)
permitaccess(s, o, r) ⇒ $\Diamond_{\leq d}$ (revokeaccess(s, o, r) ∨ endaccess(s, o, r)
Resp: Reference Monitor

Goal [EventualRevoke]
FormalDef: (∀ s:subject, o:object, r:right)
RM.permitaccess(s, o, r) ⇒ $\Diamond_{\leq d}$ RM.revokeaccess(s, o, r)
Resp: Reference Monitor

Goal [EventualEnd]
FormalDef: (∀ s:subject, o:object, r:right)
RM.permitaccess(s, o, r) ⇒ $\Diamond_{\leq d}$ RM.endaccess(s, o, r)

The KAOS operational specifications for the UCON OnA_0 enforcement mechanism follow in the text. The only difference between these operations and their equivalent shown in section 9.2 is the specification of the *Required Pre* and *Post-Condition* clauses. The `RevokeAccess` operation will be shown in Section 10.2.12

Operation: PermitAccess
Performed By: Reference Monitor
Domain Pre-Condition: true
Domain Post-Condition: RM.permitaccess(s, o, r)
Input: subject, object, right
ReqPost for [OnPermit]: CM.validate($\text{cp}_1 \wedge \ldots \wedge \text{cp}_n$)

10.2.12 Denying and Revoking the access

A careful reader may have noted that in the previos sections we didn't model neither the `DenyAccess` nor the `RevokeAccess` operations. The reason lies in the fact that we refined only positive permissions. Within this Section we show the refinements of `DenyAccess` and `RevokeAccess`.

Denying the access

In UCON, an access is denied when, after a `tryaccess(s,o,r)`, the predicates are not satisfied. A `DenyAccess` operation can be issued only when evaluating *Pre*{*ABC*} policies. The refinement shown here is valid for all the UCON *Pre*{*ABC*} models.

The top goal is the following:

Goal [AccessDenied]
RefinedTo: [Deny], [CheckPredicates], [TryToAccess]
FormalDef: (∀ s:subject, o:object, r:right)
denyaccess(s, o, r) ⇒ • policyNotSatisfied(s, o, r)

This goal can be easily refined in the formal sub-goals' definitions as follow. The first part of the refinement is shown in Figure 10.21.

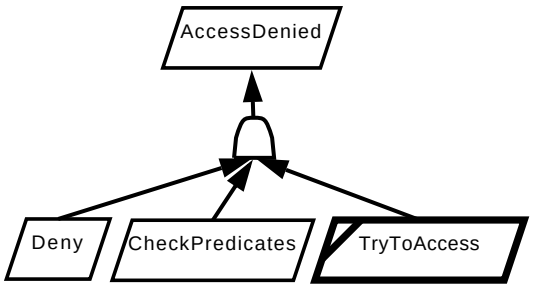

Figure 10.21: Initial goal refinement for the deny access goal

Goal [Deny]
Refines: [AccessDenied]
FormalDef: (∀ s:subject, o:object, r:right)
denyaccess(s, o, r) ⇒ • (($\neg ap_1 \vee \ldots \vee \neg ap_n$) ∨
($\neg op_1 \vee \ldots \vee \neg op_n$) ∨
($\neg cp_1 \vee \ldots \vee \neg cp_n$))

Goal [CheckPredicates]
Refines: [AccessDenied]
RefinedTo: [CP Monitor/Control], [PredicatesInValidation]
FormalDef: (∀ s:subject, o:object, r:right)
(($\neg ap_1 \vee \ldots \vee \neg ap_n$) ∨
($\neg op_1 \vee \ldots \vee \neg op_n$) ∨
($\neg cp_1 \vee \ldots \vee \neg cp_n$)) ⇒ • tryaccess(s, o, r)

Goal [TryToAccess]
Refines: [AccessDenied]
FormalDef: (∀ s:subject, o:object, r:right)
tryaccess(s, o, r) ⇒ • policyNotSatisfied(s, o, r)

Even if [TryToAccess] is a final goal (an assumption of the system), neither [Deny] nor [CheckPredicates] are finals, so they have to be refined

further. In Figure 10.22 is shown the completion of the goal refinement, and the formal definitions of each sub-goals follow in the text. We apply accuracy and actuation goals to resolve the lack of monitorability and controllability. We identify two requirement goals, [DenyTheAccess] and [PredicatesInValidation], and assign two agents, the *Reference Monitor* and *Predicate Validator* to respectively take care to each of them.

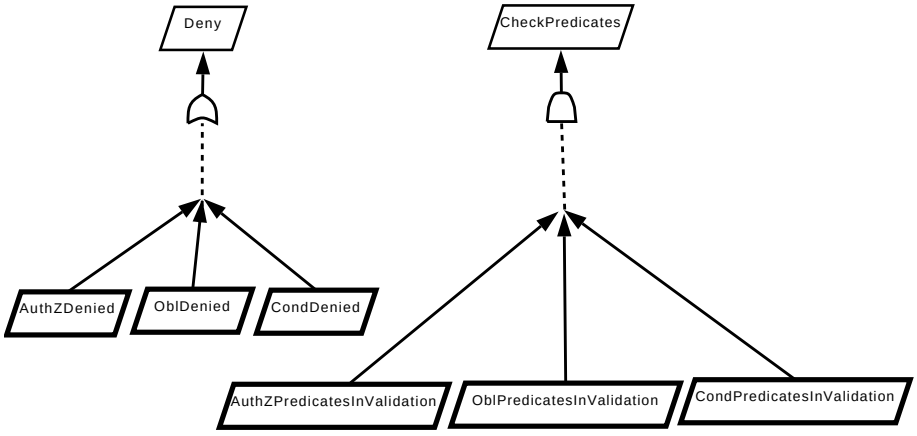

Figure 10.22: Completion of the goal refinement for the deny access

The goal [Deny] is OR-refined in three goals, which we apply actuation goals in a very similar manner to what we did in the previous chapter. They result in the following three requirements: [AuthZDenied], [OblDenied] and [CondDenied]:

Goal [AuthZDenied]
FormalDef: (∀ s:subject, o:object, r:right,
RM:Reference Monitor, APV:AuthZ Predicate Validator)
RM:denyaccess(s, o, r) ⇒ • APV.validate(($\neg ap_1 \vee \ldots \vee \neg ap_n$))
Resp: Reference Monitor

Goal [OblDenied]
FormalDef: (∀ s:subject, o:object, r:right,
RM:Reference Monitor, OM:Obligation Monitor)
RM.denyaccess(s, o, r) ⇒ • OM.validate(($\neg op_1 \vee \ldots \vee \neg op_n$))
Resp: Reference Monitor

Goal [CondDenied]
FormalDef: (∀ s:subject, o:object, r:right,
RM:Reference Monitor, CM:Condition Monitor)
RM.denyaccess(s, o, r) ⇒ • ($\neg cp_1 \vee \ldots \vee \neg cp_n$)

Resp: Reference Monitor

The goal [CheckPredicates] is AND-refined in three goals, which we apply actuation goals. The result consists in the following three requirements: [AuthZPredsInvalidation], [OblPredsInvalidation] and [CondPredsInvalidation]:

Goal [AuthZPredicatesInValidation]
FormalDef: (∀ s:subject, o:object, r:right, APV:AuthZ Predicate Validator)
APV.validate($\neg$ ap_1 $\vee$ ... $\vee$ $\neg$ ap_n)
$\Rightarrow$ • tryaccess(s, o, r)
Resp: AuthZ Predicate Validator

Goal [OblPredicatesInValidation]
FormalDef: (∀ s:subject, o:object, r:right, OM:Obligation Monitor)
OM.validate($\neg$ op_1 $\vee$ ... $\vee$ $\neg$ op_n)
$\Rightarrow$ • tryaccess(s, o, r)
Resp: Obligation Monitor

Goal [CondPredicatesInValidation]
FormalDef: (∀ s:subject, o:object, r:right, CM:Condition Monitor)
CM.validate($\neg$ cp_1 $\vee$ ... $\vee$ $\neg$ cp_n))
$\Rightarrow$ • tryaccess(s, o, r)
Resp: Condition Monitor

We are now capable to derive the KAOS agent and operation models. Figure 10.23 the operation model, together with the agent/responsibility model. We identify four operations. The first operation, `DenyAccess`, comes from the operationalization of the requirements [AuthZDenied], [OblDenied] and [CondDenied]. The second (`AuthZPredicateValidation`), third (`OblPredicateValidation`) and fourth (`CondPredicateValidation`) ones are similar to those already seen in the previous sections.

Next follows the formal specification of the operations.

Operation: DenyAccess
Performed By: Reference Monitor
Domain Pre-Condition: $\neg$ RM.denyaccess(s, o, r)
Domain Post-Condition: RM.denyaccess(s, o, r)
Input: subject, object, right
ReqPre for [AuthZDenied], [OblDenied] and **[CondDenied]**:
APV.validate($\neg$ ap_1 $\vee$ ... $\vee$ $\neg$ ap_n) $\vee$

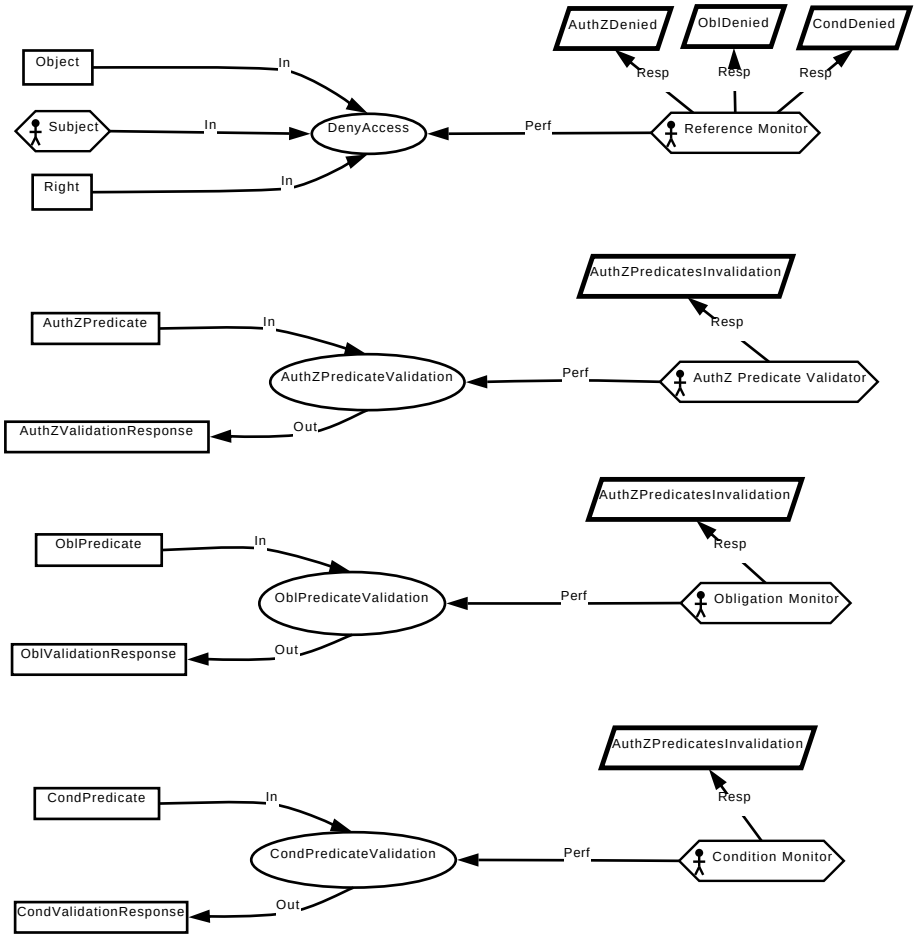

Figure 10.23: Operation model for an enforcing mechanism to deny an access

$$\text{OM.validate}(\neg\ \text{op}_1 \vee \ldots \vee \neg\ \text{op}_n\ ) \vee$$
$$\text{CM.validate}(\neg\ \text{cp}_1 \vee \ldots \vee \neg\ \text{cp}_n\ )$$

Operation: AuthZPredicateValidation
Performed By: Predicate Validator
Domain Pre-Condition: $\neg$ APV.validate($\neg\ \text{ap}_1 \vee \ldots \vee \neg\ \text{ap}_n\ $)
Domain Post-Condition: APV.validate($\neg\ \text{ap}_1 \vee \ldots \vee \neg\ \text{ap}_n\ $)
Input: AuthZPredicate
Output: AuthZValidationResponse
ReqPre for [AuthZPredicatesInValidation]: tryaccess(s, o, r)

Operation: OblPredicateValidation
Performed By: Obligation Monitor

Domain Pre-Condition: $\neg$ OM.validate($\neg$ op$_1$ $\vee \ldots \vee \neg$ op$_n$)
Domain Post-Condition: OM.validate($\neg$ op$_1$ $\vee \ldots \vee \neg$ op$_n$)
Input: OblPredicate
Output: OblValidationResponse
ReqPre for [OblPredicatesInValidation]: tryaccess(s, o, r)

Operation: CondPredicateValidation
Performed By: Condition Monitor
Domain Pre-Condition: $\neg$ CM.validate($\neg$ cp$_1$ $\vee \ldots \vee \neg$ cp$_n$)
Domain Post-Condition: CM.validate($\neg$ cp$_1$ $\vee \ldots \vee \neg$ cp$_n$)
Input: CondPredicate
Output: CondValidationResponse
ReqPre for [CondPredicatesInValidation]: tryaccess(s, o, r)

If we don't consider the predicates to be (in)validated, there is no difference between the operations as specified here and those shown in section 9.2.

Revoking the access

In UCON, an access is revoked when, during an ongoing access, the predicates are not (more) satisfied. A `RevokeAccess` operation can be issued only when evaluating a $On\{ABC\}$ policies. The refinement shown here is valid for all the $On\{ABC\}$ models.

The top goal is the following:

Goal [AccessRevoked]
RefinedTo: [Revoke], [CheckPredicates], [PolicyOnceSatisfied]
FormalDef: ($\forall$ s:subject, o:object, r:right)
revokeaccess(s, o, r) $\Rightarrow \bullet$ policyNotMoreSatisfied(s, o, r)

This goal can be easily refined in the formal sub-goals' definitions as follow. The complete refinement is shown in Figure 10.24.

Goal [Revoke]
Refines: [AccessRevoked]
FormalDef: ($\forall$ s:subject, o:object, r:right)
revokeaccess(s, o, r) $\Rightarrow$ ($\bullet$ ($\neg$ ap$_1$ $\vee \ldots \vee \neg$ ap$_n$) $\vee$
($\neg$op$_1$ $\vee \ldots \vee \neg$op$_n$) $\vee$
($\neg$cp$_1$ $\vee \ldots \vee \neg$cp$_n$))

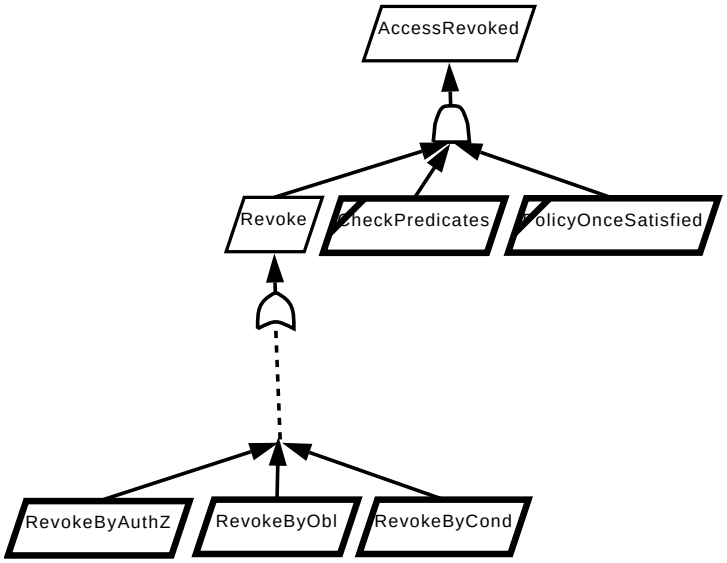

Figure 10.24: Goal refinement for the revoke access goal

Goal [CheckPredicates]
Refines: [AccessRevoked]
FormalDef: (∀ s:subject, o:object, r:right)
(($\neg ap_1 \vee \ldots \vee \neg ap_n$) ∨
($\neg op_1 \vee \ldots \vee \neg op_n$) ∨
($\neg cp_1 \vee \ldots \vee \neg cp_n$)) ⇒ • PolicyChecked(s, o, r)

Goal [PolicyOnceSatisfied]
Refines: [AccessDenied]
FormalDef: (∀ s:subject, o:object, r:right)
PolicyChecked(s, o, r) ⇒ • policyNotMoreSatisfied(s, o, r)

The `PolicyChecked(s, o, r)` predicate used in the goals [CheckPredicates] and [PolicyOnceSatisfied] is a place-holder dependent from the type of UCON On{ABC} model being evaluated. The goals [CheckPredicates] and [PolicyOnceSatisfied] are final goals (they are assumptions of the system), while [revoke] has to be refined further. The formal definitions of each subgoal follows in the text. As usual, we apply accuracy and actuation goals to resolve the lack of monitorability and controllability, but as is the former section, they are not shown here. We identify the requirement final goals [RevokeByAuthZ], [RevokeByObl] and [RevokeByCond], and assign it the *Reference Monitor* agent.

Goal [RevokeByAuthZ]

FormalDef: (∀ s:subject, o:object, r:right,
RM:Reference Monitor, APV:AuthZ Predicate Validator)
RM:revokeaccess(s, o, r) ⇒ • APV.validate(($\neg ap_1 \vee \ldots \vee \neg ap_n$))
Resp: Reference Monitor

Goal [RevokeByObl]
FormalDef: (∀ s:subject, o:object, r:right,
RM:Reference Monitor, OM:Obligation Monitor)
RM.revokeaccess(s, o, r) ⇒ • OM.validate(($\neg op_1 \vee \ldots \vee \neg op_n$))
Resp: Reference Monitor

Goal [RevokeByCond]
FormalDef: (∀ s:subject, o:object, r:right,
RM:Reference Monitor, CM:Condition Monitor)
RM.revokeaccess(s, o, r) ⇒ • ($\neg cp_1 \vee \ldots \vee \neg cp_n$)
Resp: Reference Monitor

We are now capable to derive the KAOS agent and operation models. Figure 10.25 shows them, together with the agent/responsibility model. The formal specification of the operation identified follows.

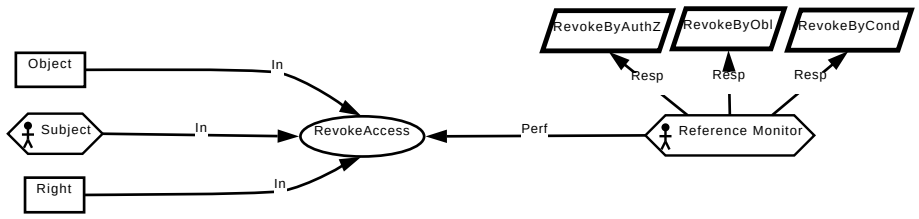

Figure 10.25: Operation model for an enforcing mechanism to revoke an access

Operation: RevokeAccess
Performed By: Reference Monitor
Domain Pre-Condition: ¬ RM.revokeaccess(s, o, r)
Domain Post-Condition: RM.revokeaccess(s, o, r)
Input: subject, object, right
ReqPre for [RevokeByAuthZ], **[RevokeByObl]** and **[RevokeByCond]**:
APV.validate($\neg ap_1 \vee \ldots \vee \neg ap_n$) ∨
OM.validate($\neg op_1 \vee \ldots \vee \neg op_n$) ∨
CM.validate($\neg cp_1 \vee \ldots \vee \neg cp_n$)

As expected, If we don't consider the predicates to be (in)validated, there is no difference between this operation as specified here and the one shown in section 9.2.

10.2.13 Correcteness of the Operationalisation

The correctness of the operationalisation of the above policy requirements is defined in the following property, which captures the completeness, minimality and consistency properties as stated in [58].

Correctness of Operationalisation. The operationalisation of the requirements [PermitToAccess], [PredicatesValidation], [UpdateTheAttributes], [UpdateAfterPermit], [OnValidation] and [UpdateAfterEnd] by the operations `PermitAccess`, `AuthZPredicateValidation`, `OblPredicateValidation`, `CondPredicateValidation` and `AttributeUpdate` is correct. This is shown by proving the following:

$$
\begin{aligned}
\forall Op \in \{&\texttt{PermitAccess},\\
&\texttt{AttributeUpdate},\\
&\texttt{AuthZPredicateValidation},\\
&\texttt{OblPredicateValidation},\\
&\texttt{CondPredicateValidation}\\
&\texttt{AuthZDenied},\\
&\texttt{OblDenied},\\
&\texttt{CondDenied},\\
&\texttt{AuthZPredicateinValidation},\\
&\texttt{OblPredicateinValidation},\\
&\texttt{CondPredicateinValidation},\\
&\texttt{RevokeByAuthZ},\\
&\texttt{RevokeByObl},\\
&\texttt{RevokeByCond}\}\\
R \in \{&\mathbf{ReqPre}, \mathbf{ReqPost}\},\\
G \in \{&[PredicatesValidation],\\
&[UpdateTheAttributes],\\
&[UpdateAfterPermit],\\
&[OnValidation],\\
&[UpdateAfterEnd],\\
&[DenyAccess],\\
&[RevokeAccess]\}:
\end{aligned}
$$

$$[\![Op.R]\!] \vDash G \quad \wedge \quad G \vDash [\![Op.R]\!] \quad \wedge \quad [\![Op.R]\!] \nvDash \mathbf{False}$$

with respect to the definitions of $[\![\ ]\!]$ and $\vDash$ as given in [58]. The $[\![\ ]\!]$ relation gives the meaning of the pre/post/trigger conditions of operations

and $\vDash$ represents the usual derivation operator in logic given premises on its left side. □ □

10.2.14 Encoding a Strategy

It is worth noting here that the sequentiality of operations determines the specific policy they can operationalise. When deriving the KAOS operational specification for all the UCON$_{abc}$ sub-models, we note that the derived operations always encode the same state-transitions as specified by those in Section 9.2, but since the sequentiality of the operations is different in each model, the *Required Pre-*, *Post-* and *Trigger Conditions* become model-dependent. We are able, for each UCON sub-model, to formally infer a *strategy* to encode the sequentiality of the operations by just looking at the *Required Pre-, Post-* and *Trigger Conditions* specified within the formal operational specification of each UCON$_{abc}$ sub-model. It should be obvious that, in order for the abstract specification presented in Section 9.2 to be an enforcement mechanism for all policies, that a way to encode the policy *strategy* is needed. A possibility lies in the use of an operational policy language like POLPA [60], where the policy specification itself encodes the strategy. Otherwise, an external scheduler can be used for the particular UCON$_{abc}$ sub-model to which the policy pertains.

10.3 Related Works

The work reported in this Chapter is associated to two strands of related works: policy refinement and derivation of enforcement mechanisms. The use of goal-refinement for refining policies as presented here was introduced by Bandara *et al* in [9]. However, their emphasis is on applying abduction techniques in order to determine the sequence of events needed to achieve a goal given a system architecture that already include enforcing components. Close to Bandera's work is the work of [85], which also refines policies by applying requirement engineering and model checking techniques based on a temporal logic formalisation similar to the one used in this paper. His approach allows one to find system executions aimed at fulfilling low-level goals that logically entail high-level strategic guidelines. From system executions, policy information is abstracted and eventually encoded into a set of refined policies specified in Ponder. Above approaches have been applied to the networking management domain.

An alternative approach to policy refinement is presented by Chadwick *et al* in [94], based on the existence of a resource hierarchy. Their work exploits Semantic-Web technology to automate the refinement process. We consider the representation of a resource hierachy as an interesting idea and plan to study as future work the inclusion of resource hierarchy in goal-based approaches to policy refinement.

In relation to the derivation of enforcement mechanims, Janicke et al present in [48] a framework for the derivation of enforcement mechanisms that guarantee compliance with the policies. Their work is based on formalising the policies in Interval Temporal Logic (ITL) and concentrates only on history-based access control policies. Our work is more operational and we consider it could be linked better to current efforts to implementing usage control for Grids, such as [110] and [60]

11 Conclusion and Future Work

This thesis is about usage control for Data Grids. We analysed usage control models, techniques and systems, and we applied them to a Data Grid abstraction. This work is a step toward a continuous monitoring and control of the data access and usage in a Data Grid. Regardless, much work has still to be done.

11.1 Summary and Results

In Chapter 3 we explained some general concepts found in the computer security world, and Section 3.3 is central when summarizing our idea on usage control, which we consider as the new, and improved, access control. Then, we used Chapter 4 to explain some basic ideas about controlling the usage of Grid data: the notions we put in that Chapter remained valid throughout the whole thesis. Basically, we consider Grid security as a process, which is composed by two black boxes: an (extended) Grid Authentication box, which is followed by a Grid Authorization box. Later, throughout the thesis, we presented solutions for both. Chapter 4 is also used to explain the possible application models for controlling the usage of Grid data, the policy framework models developed in the past, the OGF recommendations, and also review some security tools used by today's Data Grids.

Figure 11.1 is used to explain the status of the whole Grid usage control process as was found at the beginning of the whole work of the thesis. The developement of each black box has been analysed as a software engineering process, that goes from requirements analysis to maintainance, passing from architecture, design and implementation phases. A further step, *standard compliant*, is added, since the use of *standards* is becoming very important for the Grid community.

As can be seen by looking at Figure 11.1, the extended Grid authentication was covered by at least one (good) software tool. Instead, authorization decisions about the usage of Grid data still lacked even a model suitable to accomodate the security requirements of a Data Grid. Some of the tools we reviewed in Section 9.3 are used in today's Data Grids to accomodate their authorization needs, but none of them can solve all the

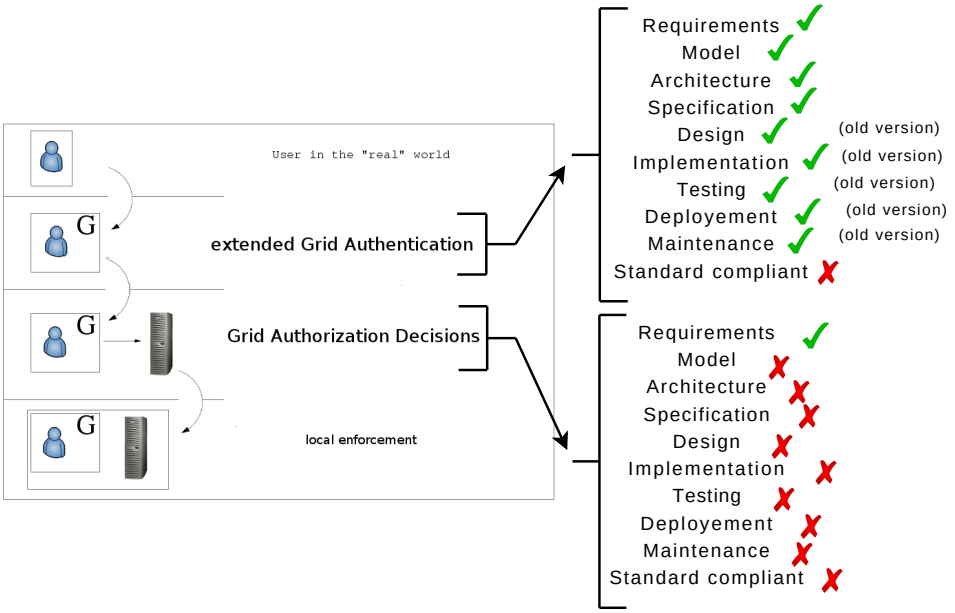

Figure 11.1: Status of Grid authentication and authorization before this thesis

security requirements we explained in Section 7.1. That's why, in Chapter 7, we showed the foundations for developing a software for usage control for Data Grids, including a distributed usage control model. Such model is suitable for multi-authoritative distributed systems, and can make use of policies pertaining to the UCON$_{abc}$ models.

Chapter 8 is about the re-engineering of a software, the VOMS tool, which provides an extended Grid authentication tool which is *standard compliant*. Later, in Chapter 9, we showed two architectures for Data Grid usage control, one concrete and one abstract, focussing on the authorization part. The second of these architecture is an abstract specification, which is verified with a methodology derived from the KAOS formal requirement engineering process. Such methodology is a brand-new, bottom up methodology, which has been developed for this thesis to show that the abstract architecture specification, showed in Chapter 9, is correct.

In summary, this thesis has presented a distributed usage control model suitable for Data Grids, a usage-based Data-Grid authorization architecture with strong reference to the OGSA work on Grid authorization architecture, a usage control architecture for Semantic Data-Grids, and a rigorous approach to the design of an enforcement mechanism for UCON$_{abc}$ usage control policies. We concentrated on the UCON model proposed by Park and Sandhu and studied its application for the case of Data Grid Management Systems (DGMS). Our approach consists in applying the KAOS requirements-engineering methodology to the design of the enforcement mech-

anism. The starting point is the definition of an abstract specification of the enforcement mechanism. Then, we applied KAOS to each of the UCON$_{abc}$ sub-models to prove that the specification is correct. The UCON$_{abc}$ policies can be refined into concrete ones — which could be enforced by the resulting system — by applying KAOS goal refinement. We also presented some ideas on the use of Semantic Grid techologies for controlling the granularity of usage control, and thus making UCON appropriate for its usage by different Grid applications running on the same middleware.

The results of this work are summarized with Figure 11.2, which is a companion to Figure 11.1. The Grid authentication step has been analysed, and a re-engineering work has been shown. The Grid authorization black box has been analysed in deep. Models, architectures and specifications has been proposed. In the following Section we will list some prototypes that can be considered when implementing the "Grid Authorization box", together with their pros and cons.

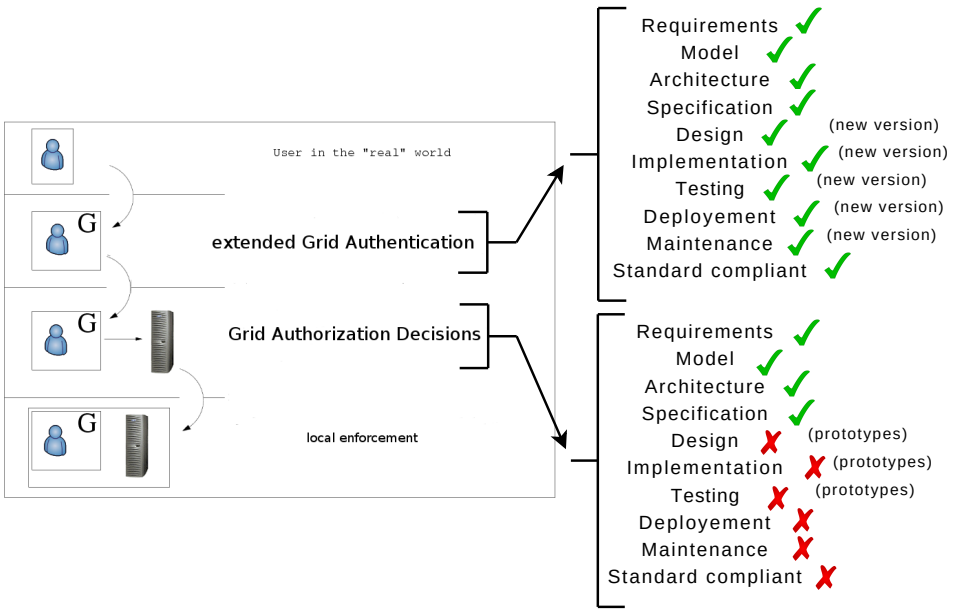

Figure 11.2: Status of Grid authentication and authorization after this thesis

We consider this thesis as a step toward an integrated usage control framework for Data Grids. We also believe that many of the ideas presented here can be adapted for the case of computational Grids and distributed systems alike.

11.2 Future Works

Future works may follow at least the following two strands: the first is about completing the engineering process, thus designing, implementing, testing and deploying the architectures and specifications that can be found in this thesis. The second strand takes in consideration the management of policies in a large environment, by using semantic Grid technologies.

11.2.1 On Completing the Software Engineering Process

Going from a theoretical analysis of Usage Control enforcement, as presented in this work, to its concrete application, on Data Grids or on other collaborative, pervasive computing systems, poses a number of questions. We are not aware of specific usage control frameworks for Data Grids, although there are already some running implementations for other scenarios.

In [60], Martinelli and Mori provide a model for usage control for computational Grids for the Globus Toolkit, following Sandhu's UCON model. The prototype implements the standard PEP-PDP architecture, and the PoLPA policy language is used to encode UCON policies. The PEP has been integrated within the application execution environment to monitor the accesses to the local resources performed by the applications executed on behalf of remote GRID users. The PDP gets the security policy from a repository, and builds its internal data structures for the policy representation. The PDP is invoked by the PEP every time the subject attempts to access a resource. It exploits its representation and determine whether the access should be allowed or not, returning to the PEP `permit` and `deny` invocations. The PDP continuously evaluates a set of given authorizations, conditions and obligations while an access is in progress, and it could invoke the PEP to terminate it through a `revoke` action. The architecture comprises the managers for attributes, conditions and obligations. The *Condition Manager* is invoked by the PDP every time the security policy requires the evaluation of a condition. The *Attribute Manager* is in charge of retrieving and updating the value of attributes. The *Obligation Manager* monitors the execution of obligations.

The authors claim that one of the strengths of this prototype is represented by its policy language, which is PoLPA. Anyway, PoLPA may represent a weakness when considering a possible deployement of this software on a Grid infrastructure, since it's not a standard, and this contrasts with requirement number 5 of those we listed in Section 7.1.2. Other deployements problems come from the fact that Martinelli and Mori focussed on single GRID computational services. We argue that the adaptation of UCON to Data Grid poses a greater number of issues to be solved. This thesis highlighted a number of them. Anyway, we believe that, with some adaptations, this prototype could be part of a larger Distributed Usage Control imple-

mentation for Data Grids.

In [110], Zhang *et al* propose a UCON prototype implementation. The security architecture leverages a centralized attribute repository in each VO and a usage monitor in each Resource Provider (RP) for attribute management. Both PDP and PEP are located on the RP side. For an access, the PDP collects the subject, object and system attributes, and makes the usage control decision, which is enforced by the PEP. The immutable subject attributes are pushed to the PDP by the requesting subject.

The policies are specified with XACML, which, as recognized by the same authors, seems suffers of several limitations to exactly encode UCON policies. The fact that even a standard language like XACML does not provide sufficient expressive capability leads to considerations that we leave as future works. As is the case for the former one, this prototype has not been applied to an actual (Data) Grid security architecture, like the OGSA one.

Another prototype that has to be considered for the implementation of the *super-PEP* architectural element (see Section 9.1) is envisaged in [7], which is a work from the same authors of [60], and again uses PoLPA, this time not as a policy language, but as an orchestrator language. An alternative, standard orchestrator language, is represented by the Business Process Execution Language (BPEL) [50].

Even if these prototypes should be considered when implementing a Data Grid usage control architecture, none of them consider the inner multi-authoritative nature of Data Grids and their specific issues.

11.2.2 On the Use of Semantic Grid Technologies

We published some of the ideas for future work that we report in this Section in [36].

In the near future, data on the order of hundreds of petabytes will be spread in multiple storage systems worldwide dispersed in, potentially, billions of replicated data items. The creation, definition and enforcement of usage control policies may represent an issue in terms of management, scalability, governability and consistency. For example, in current hierarchical file systems, access control is made specifying the authorizations on every one of billions of files. If usage and access control techniques are to be really useful in a large pervasive environment, they should be able to solve the scalability and governability problems presented by the more traditional access control models, such as Identity Based Access Control (IBAC) — normally implemented using Access Control Lists (ACLs) — or even the more flexible Role Based Access Control (RBAC) [28]. In the implementations of traditional access control models, when an authorization policy changes for a specific user or role, the security manager must implement the adjustment in every entry involved, potentially all. Moreover, frequent authorization

mutations and a big number of users or roles make worse the possibility of the authorization system being managed in an effective way. These factor may generate a policy explosion phenomenon. What's needed is a mechanism for keeping under control the policy granularity. A simple solution lies in the semantic binding assertions regarding Grid users and resources, as exposed in a Semantic Grid.

UCON subjects and objects may be semantic concepts extracted from those VO ontologies or scientific model ontologies used in the Semantic Grid, like those we explained in Section 8.4. Before going any further, we make a clear distinction between semantic attributes and UCON attributes. Semantic attributes can globally describe users, data and resources properties, but are not meant to be security attributes. Instead, the UCON attributes define only subjects' and objects' security properties, and for many of them there is no need to be known outside the usage control service. For example, consider the following UCON PreA_{1+3} policy, viz. a pre-authorization policy with pre and post attribute updates (written in POLPA, where . represents sequence of actions):

```
TryAccess(John_Doe, file_xyz, view).
PredicateValidation(
    [John_Doe.openedFiles <
     John_Doe.MAX_openedFiles]).
AttributeUpdate(
    John_Doe.openedFiles, add, 1).
PermitAccess(John_Doe, file_xyz, view).
EndAccess(John_Doe, file_xyz, view).
AttributeUpdate(
    John_Doe.openedFiles, reduce, 1).
```

The subject is the user `John_Doe`, the object is the abstract name `file_xyz`, and the UCON (Grid) right requested is simply `view`. This policy makes use of two John_Doe's attributes in the predicate at lines 2-4, `openedFiles` and `MAX_openedFiles`, and updates `openedFiles` at lines 5-6 and 9-10. The attribute `openedFiles` represents the number of files accessed at the same time by `John_Doe`, while `MAX_openedFiles` represents the maximum number of files that can be accessed at the same time. These attributes don't need to be known outside the usage control service, because they are used to store security properties and don't describe semantic characteristics of the user. We argue that no UCON attribute, neither mutable or persistent, could be considered as a semantic one.

Semantic Grid technologies come in play for the definition of the UCON subjects and objects. A semantic-aware UCON PDP is depicted in Figure 11.3, and is obviously much similar to the one presented in Section 9.1 with Figures 9.1 and 9.2. In a Semantic Grid, the client service (i.e. the Grid User) and the data to be accessed (e.g. the abstract name managed by the DGMS) are represented by a Knowledge Entities (KE) (see Section 8.4).

For what concerns the DGMS, the metadata repository can be used to store the KE of the abstract names. Even if in Semantic Grids specific Grid Users will keep asking to access specific Grid Data, a semantic-aware PDP would search for applicable policies using the multiple fields of the KEs of both the Grid user and the resource to be accessed. In this way, two or more policies could be applicable for a single access request, thus generating more than a single policy control for a single access request.

When no policy is applicable, the access is denied. When multiple UCON Pre$\{ABC\}$ policies are to be evaluated, even if just one is satisfied, then the access will be permitted. When multiple UCON On$\{AB\}$ policies are to be evaluated, even if just one is no more satisfied, then the access will be revoked.

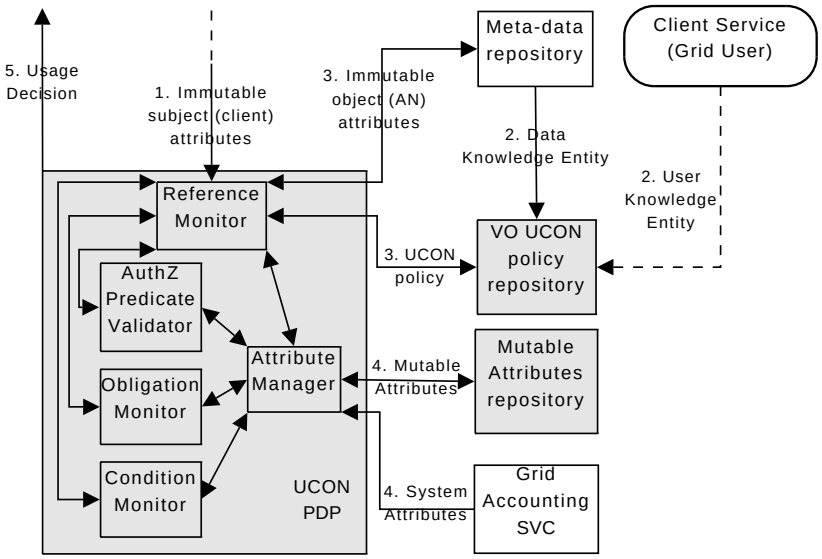

Figure 11.3: A semantic-aware UCON PDP

A security administrator can control the policy granularity using the semantic fields shown in Figure 8.3 for the definition of collective policies, like the following simple PreA_0 policy (written in POLPA):

```
TryAccess(Institution:STFC, Study:ISIS, read).
PredicateValidation([]).
PermitAccess(Institution:STFC, Study:ISIS, read).
EndAccess(Institution:STFC, Study:ISIS, read).
```

This policy states that each User associated with the Institution `STFC` can read those GD pertaining to the `ISIS` study. UCON attributes can be associated to these UCON subjects and objects.

The possibility to control the policy granularity, and thus to avoid the policy explosion is of particular interest for those VOs that consider the

specification of a per-user, per-role or per-data policies a useless effort. High Energy Physics VOs usually fall in this category.

Bibliography

[1] J.R. Abrial, M.K.O. Lee, D. Neilson, PN Scharbach, and I. Sørensen. The B-method. In *Proceedings of the 4th International Symposium of VDM Europe on Formal Software Development*, volume 2, pages 398–405. Springer.

[2] J. Ainsworth, S. Newhouse, and J. MacLaren. Resource usage service (rus) based on ws-i basic profile 1.00. www.ogf.org/Public_Comment_Docs/Documents/Apr-2007/ogsa-bes-v33.pdf, 2006.

[3] Roberto Alfieri, Roberto Cecchini, Vincenzo Ciaschini, Luca dell'Agnello, Ákos Frohner, Károly Lörentey, and Fabio Spataro. From gridmap-file to voms: managing authorization in a grid environment. *Future Generation Comp. Syst.*, 21(4):549–558, 2005.

[4] S. Andreozzi. *On the Quality-Based Evaluation and Selection of Grid Services*. PhD thesis, University of Bologna, Department of Computer Science, Apr 2006.

[5] M. Antonioletti, M. Atkinson, A. Krause, S. Laws, S. Malaika, N. W. Paton, D. Pearson, and G. Riccardi. Web services data access and integration - the core (ws-dai) specification, version 1.0, 2006.

[6] M. Antonioletti, D. Berry, A. Chervenak, P.r Kunszt, A. Luniewski, S. Laws, and M. Morgan. Ogsa data architecture v0.6.6. Technical report, Open Grid Forum, 2007.

[7] Benjamin Aziz, Alvaro Arenas, Fabio Martinelli, Ilaria Matteucci, and Paolo Mori. Controlling usage in business process workflows through fine-grained security policies. In Springer, editor, *5th International Conference on Trust, Privacy& Security in Digital Business*, 2008.

[8] J. Bacon, K. Moody, and W. Yao. A model of OASIS role-based access control and its support for active security. *ACM Transactions on Information and System Security (TISSEC)*, 5(4):492–540, November 2002.

[9] Arosha K. Bandara, Emil C. Lupu, Jonathan Moffett, and Alessandra Russo. A Goal-based Approach to Policy Refinement. In *5th IEEE Workshop on Policies for Distributed Systems and Networks*. IEEE Computer Society, 2004.

[10] T. Barton, J. Basney, T. Freeman, T. Scavo, F. Siebenlist, V. Welch, R. Ananthakrishnan, B. Baker, M. Goode, and K. Keahey. Identity federation and attribute-based authorization through the globus toolkit, shibboleth, gridshib, and myproxy, Apr 2006.

[11] Tim Berners-Lee. Semantic web road map. http://www.w3.org/DesignIssues/Semantic.html, 1998.

[12] E. Bertino and E. Ferrari. A logical framework for reasoning about access control models. *ACM Transactions on Information and System Security (TISSEC)*, 6(1):71–127, 2003.

[13] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Or-chard. Web services architecture, 2004. `http://www.w3.org/TR/ws-arch/`.

[14] A. Caltroni, V. Ciaschini, A. Ferraro, A. Ghiselli, G. Rubini, and R. Zappi. G-pbox: A policy framework for grid environments. In *Proceedings of the International CHEP 2004*, Interlaken, Switzerland, 2004.

[15] S. Cantor, F. Hirsch, J. Kemp, R. Philpott, and E. Maler. Bindings for the oasis security assertion markup language (saml) v2.0. http://docs.oasis-open.org/security/saml/v2.0/saml-bindings-2.0-os.pdf, 2005.

[16] S. Cantor, J. Kemp, R. Philpott, and E. Maler. Oasis security assertion markup language (saml) tc. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security, 2005.

[17] D. Chadwick. An x.509 role-base privilege management infrastructure. Technical report, 2002.

[18] D. Chadwick. Functional components of grid service provider authorisation service middleware. Technical report, Open Grid Forum, 2008.

[19] David Chadwick, Gansen Zhao, Sassa Otenko, Romain Laborde, Linying Su, and Tuan-Anh Nguyen. Building a modular authorisation infrastructure. In *The UK e-Science All Hands Meeting, Nottingham*, September 2006.

[20] DW Chadwick and A Otenko. Implementing Role Based Access Controls using X.509 Privilege Management - the PERMIS Authorisation Infrastructure. In Borka Jerman-Blazic, Wolfgang Schneider, and Tomaz Klobucar, editors, *Security and Privacy in Advanced Networking Technologies*, NATO Science Series, pages 26–39. IOS Press, unknown 2004. Proceedings of the NATO Advanced Networking Workshop on Advanced Security Technologies in Networking, Bled, Slovenia, 15-18 September 2003.

[21] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets, 1999.

[22] Madhu Chetty and Rajkumar Buyya. Weaving computational grids: How analogous are they with electrical grids? *Computing in Science and Engg.*, 4(4):61–71, 2002.

[23] V. Ciaschini, V. Venturi, and A. Ceccanti. The voms attribute certificate format. Technical report, OGSA Authorization working group, Jan 26, 2005.

[24] Óscar Corcho, Pinar Alper, Ioannis Kotsiopoulos, Paolo Missier, Sean Bechhofer, and Carole A. Goble. An overview of s-ogsa: A reference semantic grid architecture. *J. Web Sem.*, 4(2):102–115, 2006.

[25] dCACHE Group. grid-aware pluggable authorization management.

[26] Sandrine Duflos, Gladys Diaz, Valérie Gay, and Eric Horlait. A comparative study of policy specification languages for secure distributed applications. 2506:157–??, 2002.

[27] S. Farrell and R. Housley. An Internet Attribute Certificate Profile for Authorization. RFC 3281 (Proposed Standard), April 2002.

[28] D. Ferraiolo, R. Sandhu, S. Gavrila, D.R. Kuhn, and R. Chandramouli. Proposed nist standard for role-based access control. *ACM Transactions on Information and System Security (TISSEC)*, (3):224–274, 2001.

[29] M.D.F. Ferraiolo, M.D.M. Gilbert, and M.N. Lynch. AN EXAMINATION OF FEDERAL AND COMMERCIAL ACCESS CONTROL POLICY NEEDS. *National Computer Security Conference, 1993 (16th) Proceedings: Information Systems Security: User Choices*, 1995.

[30] I. Foster, A. Grimshaw, P. Lane, W. Lee, S. Newhouse, S. Pickles, D. Pulsipher, C. Smith, and M. Theimer. Ogsa basic execution service version 1.0, 2007.

[31] I. Foster, C. Kesselman, L. Pearlman, S. Tuecke, and V. Welch. The community authorization service: Status and future. In *In Proceedings of Computing in High Energy Physics 03 (CHEP '03)*, 2003.

[32] Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. *Grid Computing: Making the Global Infrastructure a Reality*, chapter The Physiology of the Grid, pages 217–249. Wiley, 2003.

[33] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Jounral of Supercomputer Applications*, 15(3), 2001.

[34] A Frohner, Peter Z Kunszt, R Brito da Rocha, and E Laure. Security of distributed data management. Technical Report EGEE-TR-2006-003. EGEE-TR-2006-DATASEC, 2006.

[35] Patrick Fuhrmann and Volker Gülzow. dcache, storage system for the future. In *Euro-Par*, pages 1106–1113, 2006.

[36] A. Ghiselli, L. Magnoni, F. Stagni, and R. Zappi. Enriched namespace to support content-aware authorization policies. *Journal of Physics: Conference Series*, 2007.

[37] Antonia Ghiselli, Federico Stagni, and Riccardo Zappi. Review of security models applied to distributed data access. In *Euro-Par Workshops*, pages 34–48, 2006.

[38] Francois Grey, Matti Heikkurinen, Rosy Mondardini, and Robindra Prabhu. How does grid computing work?, May 2008. `http://www.gridcafe.org/version1/gridatwork/architecture.html`.

[39] A. S. Grimshaw and W. A. Wulf. Legion-a view from 50,000 feet. In *HPDC '96: Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*, page 89, Washington, DC, USA, 1996. IEEE Computer Society.

[40] Andrew Grimshaw, Adam Ferrari, and Katherine Holcomb. Metacomputing– what's in it for me? greg lindahl.

[41] Andrew S. Grimshaw, William A. Wulf, James C. French, Alfred C. Weaver, and Jr. Paul F. Reynolds. Legion: The next logical step toward a nationwide virtual computer. Technical report, Charlottesville, VA, USA, 1994.

[42] The Storage Resource Management Working Group. An internet attribute certificate profile for authorization. http://sdm.lbl.gov/srm-wg/doc/SRM.v2.2.pdf, may 2008.

[43] P. Gutmann. Why xml security is broken. http://www.cs.auckland.ac.nz/ pgut001/pubs/xmlsec.txt, 2004.

[44] HealthGrid. Healthgrid white paper. Technical Report HealthGrid-White_Paper-Draft_v.1.1-5, HealthGrid, 2004.

[45] R. Housley, W. Polk, W. Ford, and D. Solo. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 3280 (Proposed Standard), April 2002. Updated by RFCs 4325, 4630.

[46] Internet2/MACE. The shibboleth project.

[47] ITU-T. Information technology open systems interconnection security frameworks for open systems: Access control framework. Technical report, ITU-T, 1995.

[48] Helge Janicke, Antonio Cau, Francois Siewe, and Hussein Zedan. Deriving Enforcement Mechanisms from Policies. In *Eighth IEEE International Workshop on Policies for Distributed Systems and Networks*. IEEE Computer Society, 2007.

[49] C.S. Jordan. *Guide to Understanding Discretionary Access Control in Trusted Systems*. DIANE Publishing, 1987.

[50] D. Jordan and J. Evdemon. Web services business process execution language version 2.0. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf, 2007.

[51] Carl Kesselman and Ian Foster. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, November 1998.

[52] Klaus Krauter, Rajkumar Buyya, and Muthucumaru Maheswaran. A taxonomy and survey of grid resource management systems for distributed computing. *Software: Practice and Experience*, 32:135–164, 2002.

[53] L. Lamport. *Specifying systems*. Addison-Wesley, 2003.

[54] B. Lang, I. Foster, F. Siebenlist, R. Ananthakrishnan, and T. Freeman. A multipolicy authorization framework for grid security. In *Proc. Fifth IEEE Symposium on Network Computing and Application*, July 2006.

[55] P. Lebrun. The Large Hadron Collider, a megascience project. Prepared for INFN Eloisatron Project: 38th Workshop, Erice, Italy, 3-11 Aug 1999.

[56] E. Letier. *Reasoning about Agents in Goal-Oriented Requirements Engineering*. PhD in informatics, Universit Catholique de Louvain, Universit Catholique de Louvain, Dpt. Ingnierie Informatique, Belgium, 2001.

[57] E. Letier and A. van Lamsweerde. Agent-based tactics for goal-oriented requirements elaboration. 2001.

[58] E. Letier and A. van Lamsweerde. Deriving Operational Software Specifications from System Goals. In *FSE'10: 10th ACM S1GSOFT Symp. on the Foundations of Software Engineering*, 2002.

[59] M. Lorch, D. B. Adams, D. Kafura, M. S. R. Koneni, A. Rathi, and S. Shah. The prima system for privilege management, authorization

and enforcement in grid environments. In *GRID '03: Proceedings of the Fourth International Workshop on Grid Computing*, page 109, Washington, DC, USA, 2003. IEEE Computer Society.

[60] Fabio Martinelli and Paolo Mori. A Model for Usage Control in GRID systems. In *Grid-STP 2007, International Conference on Security, Trust and Privacy in Grid Systems*. IEEE Computer Society, 2007.

[61] Fabio Martinelli, Paolo Mori, and Anna Vaccarelli. Towards continuous usage control on grid computational services. In *ICAS/ICNS*, page 82, 2005.

[62] Steve McConnell. *Code complete: a practical handbook of software construction*. Microsoft Press, Redmond, WA, USA, 1993.

[63] J. McLean and NAVAL RESEARCH LAB WASHINGTON DC CENTER FOR HIGH ASSURANCE COMPUTING SYSTEMS (CHACS. Security Models and Information Flow, 1990.

[64] A. Mige. *Definition of a formal framework for specifying security policies - The Or-BAC model*. PhD in informatics, Ecole Nationale Suprieure des Tlcommunications, Ecole Nationale Suprieure des Tlcommunications, 2005.

[65] Jonathan D. Moffett and Morris S. Sloman. Policy Hierarchies for Distributed System Management. *IEEE JSAC Special Issue on Network Management*, 11(9), 11 1993.

[66] R. Moore, A. Jagatheesan, A. Rajasekar, M. Wan, and W. Schroeder. Data Grid Management Systems. In *Proceedings of the 21stIEEE/-NASA Conference on Mass Storage Systems and Technologies*, Marylaand, USA, 2004.

[67] C. Munro and B. Koblitz. Performance comparison of the lcg2 and glite file catalogues. In *Nucl. Instrum. Methods Phys. Res., 10th International Workshop on Advanced Computing and Analysis Techniques in Physics Research*, pages 48–52, 2006.

[68] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[69] A. Nadalin, C. Kaler, R. Monzillo, and P. Hallam-Baker. Web service security: Saml token profile 1.1. http://docs.oasis-open.org/wss/v1.1/wss-v1.1-spec-pr-SAMLTokenProfile-01.pdf, 2006.

[70] A. Nadalin, C. Kaler, R. Monzillo, and P. Hallam-Baker. Web service security: Soap message security 1.1 (ws-security 2004). http://docs.oasis-open.org/wss/v1.1/wss-v1.1-spec-pr-SOAPMessageSecurity-01.pdf, 2006.

[71] Syed Naqvi, Philippe Massonet, and Alvaro Arenas. A study of languages for the specification of grid security policies. Technical Report TR-0037, Institute on Knowledge and Data Management, CoreGRID - Network of Excellence, April 2006.

[72] Z. Nemeth and V. Sunderam. Characterizing grids: Attributes, definitions, and formalisms. *Journal of Grid Computing*, 1(1):9–23, 2003.

[73] Zsolt Nmeth and Vaidy Sunderam. Virtualization in grids: A semantical approach. In Jos C. Cunha and Omer F. Rana, editors, *Grid Computing: Software Environments and Tools*. Springer, 2006.

[74] OASIS. Oasis extensible access control markup language (xacml) tc. http://www.oasis-open.org/committees/xacml, 2005.

[75] A. Frohner P. Kunszt. glite data management security model disussion, 2005.

[76] S. Parastatidis P. Periorellis. Task based access control for virtual organizations. In *4th International workshop on scientific engineering of distributed java applications*, pages 38–47, November 2004.

[77] J. Park and R.S. Sandhu. The UCON_{abc} Usage Control Model. *ACM Transactions on Information and System Security*, 7(1):128–174, February 2004.

[78] Donn Parker. *Toward a New Framework for Information Security*, chapter 5. Robert Gezelter, 2002.

[79] C. Ponsard, P. Massonet, J. F. Molderez, A. Rifaut, A. van Lamsweerde, and Tran Van Hung. Early Verification and Validation of Mission Critical Systems. *Journal of Formal Methods in System Design*, 30(3), 2007.

[80] Alexander Pretschner, Manuel Hilty, and David Basin. Distributed usage control. *Communications of the ACM*, September 2006.

[81] Alexander Pretschner, Fabio Massacci, and Manuel Hilty. Usage control in service-oriented architectures. In *TrustBus*, pages 83–93, 2007.

[82] R. Randall and R. Philpott. Saml attribute sharing profile for x.509 authentication-based systems. www.oasis-open.org/committees/download.php/18058/, 2006.

[83] M. Richters and M. Gogolla. On Formalizing the UML Object Constraint Language OCL. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 449–464, 1998.

[84] Nick Rozanski and Eóin Woods. *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley Professional, April 2005.

[85] J. Rubio-Loyola, J. Serrat, M. Charalambides, P. Flegkas, G. Pavlou, and A. Lafuente. Using Linear Temporal Model Checking for Goal-Oriented Policy Refinement Frameworks. In *Sixth IEEE International Workshop on Policies for Distributed Systems and Networks*. IEEE, 2005.

[86] R. Sandhu, V. Bhamidipati, and Q. Munawer. The ARBAC97 model for role-based administration of roles. *ACM Transactions on Information and System Security (TISSEC)*, 2(1):105–135, 1999.

[87] Ravi S. Sandhu and Jaehong Park. Usage Control: A Vision for Next Generation Access Control. In *MMM-ACNS*, pages 17–31, 2003.

[88] R.S. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman. Role-Based Access Control Models. *COMPUTER*, pages 38–47, 1996.

[89] Bruce Schneier. *Secrets and Lies : Digital Security in a Networked World*. Wiley, January 2004.

[90] The Globus security team. Gt 4.0 security. http://www.globus.org/toolkit/docs/4.0/security/, 2005.

[91] J.M. Spivey. *The Z notation*. Prentice Hall New York, 1989.

[92] F. Stagni, A. Arenas, B. Aziz, and F. Martinelli. On usage control in data grids. In *Proceedings of The 3rd IFIP International Conference on Trust Management (TM'09)*. Springer, 2009.

[93] Federico Stagni, Alvaro E. Arenas, and Benjamin Aziz. On usage control in data grids. Technical Report TR-0154, Institute on Knowledge and Data Management, CoreGRID - Network of Excellence, June 2008.

[94] Linying Su, David W. Chadwick, Andrew Basden, and James A. Cunningham. Automated Decomposition of Access Control Policies. In *Sixth IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 3–13. IEEE Computer Society, 2005.

[95] S. Sufi and B. M. Matthews. The cclrc scientific metadata model: a metadata model for the exploitation of scientific studies and associated data. In *Knowledge and Data Management in Grids*, 2005.

[96] EGEE JRA2 team. Egee global security architecture for web and legacy services. deliverable EGEE-JRA3-TEC-487004-DJRA3.1-v1-1, EGEE JRA3, 2004.

[97] W. Trmper. Summary about posix.1e. http://wt.xpilot.org/publications/posix.1e, 1999.

[98] S. Tuecke, V. Welch, D. Engert, L. Pearlman, and M. Thompson. Internet x.509 public key infrastructure (pki) proxy certificate profile. RFC 3820 (Proposed Standard), jun 2004.

[99] A. van Lamsweerde. Requirements Engineering in the Year 00: A Research Perspective. In *International Conference on Software Engineering*, pages 5–19, 2000.

[100] Axel van Lamsweerde. Elaborating security requirements by construction of intentional anti-models. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 148–157, Washington, DC, USA, 2004. IEEE Computer Society.

[101] Moshe Y. Vardi. Branching vs. linear time: Final showdown. In Tiziana Margaria and Wang Yi, editors, *Proceedings of the 7th International Conference On Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, volume 2031 of *Lecture Notes in Computer Science*, pages 1–22, Genova, Italy, April 2001. Springer.

[102] G. Venekamp. Activity user requirements. deliverable EGEE-JRA3-TEC-485295-UserReq-v1-0, EGEE JRA3, 2004.

[103] V. Venturi, M. Riedel, A.S. Memon, M.S. Memon, F. Stagni, B. Sch uller, D. Mallmann, B. Tweddell, A. Gianoli, V. Ciaschini, S. van de Berghe, D. Snelling, and A. Streit. Using saml-based voms for authorization within web services-based unicore grids. LNCS. in conjunction with EuroPar 2007, SPRINGER, August 2007.

[104] Valerio Venturi, Federico Stagni, Alberto Gianoli, Andrea Ceccanti, and Vincenzo Ciaschini. Virtual organization management across middleware boundaries. In *E-SCIENCE '07: Proceedings of the Third IEEE International Conference on e-Science and Grid Computing*, pages 545–552, Washington, DC, USA, 2007. IEEE Computer Society.

[105] Srikumar Venugopal, Rajkumar Buyya, and Kotagiri Ramamohanarao. A Taxonomy of Data Grids for Distributed Data Sharing, Management, and Processing. *ACM Comput. Surv.*, 38(1):3, 2006.

[106] J. Viega and G. Mcgraw. *Building secure software: how to avoid security problems the right way.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[107] J. Vollbrecht, P. Calhoun, S. Farrell, L. Gommans, G. Gross, B. de Bruijn, C. de Laat, M. Holdrege, and D. Spence. AAA Authorization Framework. RFC 2904 (Informational), August 2000.

[108] P. Walmsley, D. C. Fallside, H. S. Thompson, D. Beech, M. Maloney, N. Mendelsohn, and P. V. Biron A. Malhotra. Xml schema. http://www.w3.org/TR, 2004.

[109] A. Westerinen, J. Schnizlein, J. Strassner, M. Scherling, B. Quinn, S. Herzog, A. Huynh, M. Carlson, J. Perry, and S. Waldbusser. Terminology for Policy-Based Management. RFC 3198 (Informational), November 2001.

[110] Xinwen Zhang, Masayuki Nakae, Michael J. Covington, and Ravi Sandhu. Toward a usage-based security framework for collaborative computing systems. *ACM Trans. Inf. Syst. Secur.*, 11(1):1–36, 2008.

[111] Xinwen Zhang, Francesco Parisi-Presicce, Ravi Sandhu, and Jaehong Park. Formal Model and Policy Specification of Usage Control. *ACM Transactions on Information and System Security*, 8(4):351–387, 2005.

List of Figures

List of Tables