European Research Network on Foundations, Software Infrastructures and Applications for large scale distributed, GRID and Peer-to-Peer Technologies
A Network of Excellence funded by the European Commission

On Usage Control in Data Grids

Federico Stagni {federico.stagni}@fe.infn.it
Istituto Nazionale di Fisica Nucleare sez. di Ferrara,
via Saragat 1 - 44100 Ferrara, Italy

Alvaro E. Arenas, Benjamin Aziz
{A.E.Arenas, B.Aziz}@rl.ac.uk
e-Science centre,
STFC Rutherford Appleton Laboratory,
Oxfordshire, UK

CoreGRID Technical Report
Number TR-0154
June 16, 2008

Institute on Knowledge and Data Management

CoreGRID - Network of Excellence
URL: http://www.coregrid.net

CoreGRID is a Network of Excellence funded by the European Commission under the Sixth Framework Programme

Project no. FP6-004265

On Usage Control in Data Grids

Federico Stagni {federico.stagni}@fe.infn.it
Istituto Nazionale di Fisica Nucleare sez. di Ferrara,
via Saragat 1 - 44100 Ferrara, Italy

Alvaro E. Arenas, Benjamin Aziz
{A.E.Arenas, B.Aziz}@rl.ac.uk
e-Science centre,
STFC Rutherford Appleton Laboratory,
Oxfordshire, UK

CoreGRID TR-0154

June 16, 2008

Abstract

This paper reasons on usage control in Data Grids. First, we present a usage-based Grid authorization architecture using the functional components of the currents Grids, and consider the advantages of using Semantic Grid techologies for the specification of UCON subjects and objects. Then, we analyse the formal requirements for an enforcing mechanism of UCON policies, using the KAOS requirements engineering methodology with a bottom-up approach. To do it, we provide an abstract specification of an enforcement mechanism. Then, we prove that this specification is sound and complete showing formally that it can enforce all the policies pertaining to the Sandhu's UCON authorization sub-models. Using the rigorous requirement engineering methodology of KAOS, we derive for each sub-model the operational requirements, showing that each one can be enforced by the specification previously provided.

1 Introduction

Data Grids [30] 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.

A Data Grid Management System (DGMS) [17] is a software system used to manage Data Grids through the use of multiple abstraction mechanisms providing logical namespaces that hide the complexity of distributed data and heterogeneous resources. DGMS systems allow for data to be shared over several administrative domains in a seamless manner. However, as in any resource sharing environment, robust and rigorous treatment of data security in a DGMS 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 paper studies usage control enforcement, and take DGMS as an application case-study. Usage control techniques extend traditional access control by controlling data access as well as usage [19, 21]. Recently there has been an fresh interest in applying usage control to Grid systems [15, 31]. Here we have adopted the usage control model proposed by Park and Sandhu in [19] as UCON_{abc}.

This research work is carried out under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

After a review of the existing UCON implementations, we present a usage-based Grid authorization architecture using the functional components of the currents Grids, as presented by the Open Grid Forum (OGF)[1] group on Grid authorization. We consider the advantages of using Semantic Grid techologies for the specification of UCON subjects and objects, and how this can improve the usage control granularity. Next, we concentrate on two important aspects in the design of policy-based management systems: policy refinement and specification of enforcement mechanisms.

We have followed the KAOS requirements-engineering methodology [27] for formalising the requirements of an enforcing mechanism for UCON policies. When using KAOS, our strategy is to use the methodology with a bottom-up approach. We first show an abstract specification of an UCON_a enforcement mechanism, i.e. the UCON family of models dealing with authorizations. We encode the specification using the requirement specification language provided by KAOS. We then apply KAOS to all the UCON_a sub-models [32], and derive the KAOS agent and operational models for each of them. This way, we formally show that the enforcement mechanism is sound and complete, and capable to enforce all the policies following in the UCON_a family of core models.

The rest of the paper is structured as follows. In section 2, we give some background on gloabal namespaces as managed by a DGMS, on the UCON model and on the KAOS requirements-engineering methodology. Section 3 reviews existing UCON implementations and shows architectures for usage control for traditional and semantic grids. Section 4 shows an abstract specification of an UCON_a enforcement mechanism. In Section 5, we apply the KAOS goal model to prove that the abstract specification as shown in section 4 is capable to enforce all the UCON_a core models. Finally, section 6 discusses related work, and section 7 concludes the paper and highlights directions for future work.

2 Background

In this section, we review the global namespace concept the DGMS deals with, the UCON model and the KAOS requirement engineering methodology.

2.1 Global Namespace

In a Grid environment, the applications and the users should be able to access dispersed Grid Data without knowing their location. A DGMS provides a *naming* capability allowing users to refer to specific data resources in a physical storage system using a high level logical identifier.

The OGF provides implementation guidelines and standards to implement location independence in the grid. Data resources has to be recognized by name without any location information. The Open Grid Services Architecture (OGSA) work on data architecture [2] 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 contacted to extract the data. Usually, users are not 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. The figure 1 describes the relationship on terms.

In a Data Grid, there is a small number of authoritative points, viz. sources of Data authorizations. A DGMS, together with the local storage services, is an authoritative point, where usage control policies should be enforced,

[1] web address: `http://www.ogf.org`

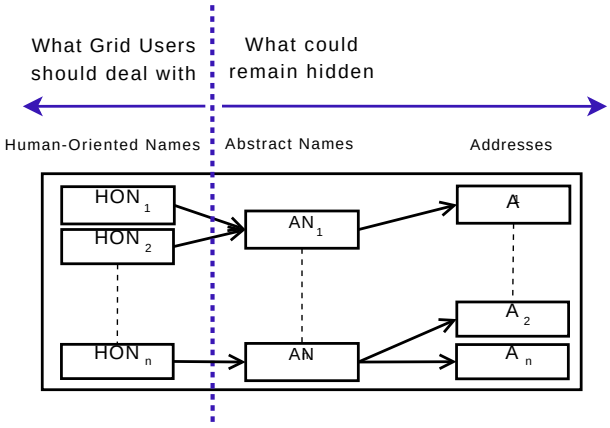


Figure 1: Data naming in Grid.

regardless of the local data services the data are actually stored. In a DGMS, the object of the usage control policies should be the *Abstract Names* requested by the Grid Users. The policies may be written by VO administrators, or, in very limited way, by the Grid users themselves.

2.2 The UCON Model

The **UCON**$_{abc}$ **usage control** model is a recent framework defined by Park and Sandhu [19, 23] 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 Data Grids, especially considering the plethora of various security needs coming from the different Data Grid applications. Next, we describe the elements of the UCON model.

- *Subjects and Objects*: The subject is the entity that exercises rights, i.e. that executes access operations on objects. An object, instead, is an entity that is accessed by subjects through access operations. When applying usage control on DGMS systems, the subjects are the users of the system and objects are the abstract names.

- *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. In DGMS, rights are permissions given to users to read from or write to abstract names.

- *Attributes*: Both subjects and objects have attributes. These 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.

- *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. Authorisation predicates express a set rules that determine whether to grant the requested right or not. An example from DGMS systems is the permission of a user to read an abstract name. The authorization predicate could exploit both attributes of the subject and of the object. The evaluation of the authorization predicates can be performed before or during the execution of an action. Obligations are UCON decision factors that are used to verify whether the subject has satisfied some mandatory requirements before performing an action or whether the subject continuously satisfies these requirements while performing the action. Obligations usually refer to future requirements that must be obeyed. 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 or during an action. In the rest of the paper, we only consider authorization predicates, also known as the the UCON$_a$ family of core models [32].

As a matter of fact, UCON$_{abc}$ is actually a family of models with several parameters. The presence of Authorizations (A), oBligations (B) and Conditions (C), pre- and on-going decisions, as well as the mutability of attributes (immutable (0), preUpdate (1), onUpdate (2), postUpdate (3)) are the factors to be considered. The UCON$_a$ submodels we consider are PreA$_0$, PreA$_1$, PreA$_3$, OnA$_0$, OnA$_1$, OnA$_2$ and OnA$_3$. A PreA$_0$ policy is an pre-authorization policy with no attributes update, a PreA$_1$ is a pre-authorization with a preUpdate of on or more attributes, and so on. The PreA$_2$ policy model isn't considered because it's not a case likely to be useful in practice.

Figure 2 illustrates the different actions that subjects and systems can perform in the UCON model [32]. These actions relate to 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. These actions relate to the different phases of an object's usage.

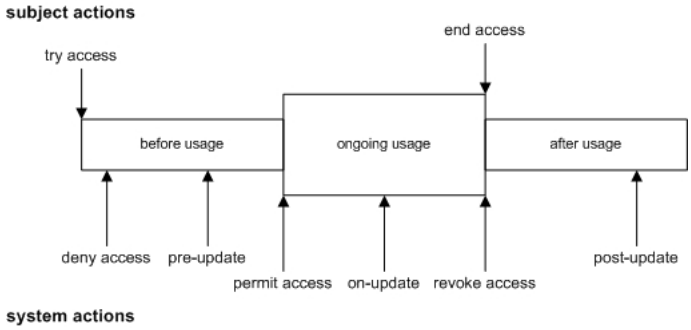


Figure 2: The UCON actions model [32]

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

- `tryaccess(s,o,r)`: performed by 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)`.
- `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)`.

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.

2.3 The KAOS Methodology

Knowledge acquisition in automated specification (KAOS) is a generic methodology based on capturing, structuring and precise formulation of system goals [27]. 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.

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) [29] 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 Q will be true until a time when P will be true.

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

Picture 3 shows an overview of the KAOS models and their inter-relations. This example has been widely taken from [12], and shows excerpt of the goal, object, agent and operation models for a meeting scheduling problem. Within the goal model of picture 3, 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. Picture 3 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. Picture 3 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.

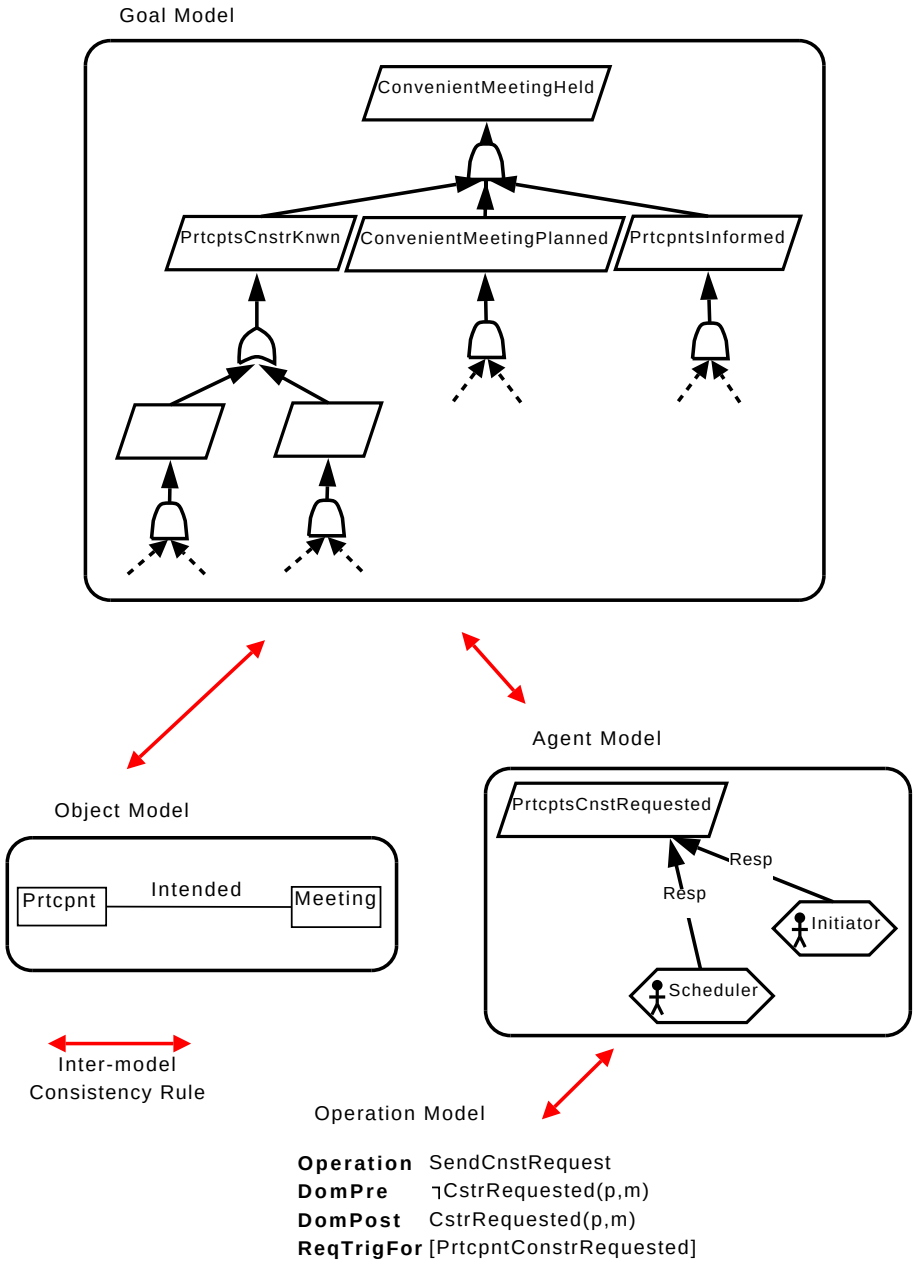


Figure 3: Overview of the KAOS models

3 Usage Control on Grids

In this section, we first review some reference implementations of usage control for Grids, then we introduce the target Grid security architecture for our UCON-policy enforcing mechanism, and finally we make some considerations on the application of usage control in Semantic Grids.

3.1 Reference Implementations of Usage Control for Grids

At our knowledge, the only implementations of usage control for Grids are [15] and [31].

In [15], Martinelli and Mori provide a model on usage control for computational Grids, following the Sandhu's UCON model. One of the most interesting peculiarities of their work is the use of a *POlicy Language based on Process Algebra* (POLPA) as policy specification language, which is especially suitable to model the usage policy models of the original UCON model. The security policy describes the order in which the security-relevant actions can be performed.

The prototype implements an architecture where the main components are a Policy Enforcement Point (PEP) and a Policy Decision Point (PDP), such as most of the common authorization systems. The PEP is integrated in the GRID environment middleware, e.g. into the Globus architecture[2], and implements the `tryaccess(s, o, r)` and the `endaccess(s, o, r)` actions. The prototyped PEP has been integrated within the application execution environment to monitor the accesses to the local resources (e.g. files or sockets) performed by the applications executed on behalf of remote GRID users. The **PDP** is the component of the architecture that performs the usage decision process. The PDP gets the security policy from a repository, and it builds its internal data structures for the policy representation. The PDP is invoked by the PEP every time that the subject attempts to access a resource. It exploits its internal representation of the policy to determine whether the access should be allowed or not and, consequently, it returns `permitaccess(s, o, r)` or `denyaccess(s, o, r)` to the PEP, that enforces it. 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 the `revokeaccess(s, o, r)` 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.

In [31], Zhang *et al* propose an UCON prototype implementation for Grids and collaborative applications, by following a layered approach with policy, enforcement, and implementation models, called the *policy-enforcement-implementation* (PEI) framework. The security architecture leverages a centralized attribute repository in each Virtual Organization (VO) and a usage monitor in each Resource Provider (RP) for attribute management.

The policies are specified with the eXtensible Access Control Markup Language (XACML) [18]. As recognized by the same authors, even if XACML is an open-standard format to specify access control policies, it suffers from the impossibility to exactly encode an abstract UCON policy.

Within the architecture, both PDP and PEP are located on the resource provider side. For an access, the PDP collects the subject and object attributes, as well as system attributes provided by supporting services in the VO, and makes the control decision, which is enforced by the PEP. The immutable subject attributes are pushed to the PDP by the requesting subject. Mutable subject attributes are pulled by the PDP from the VOs centralized attribute repository, and mutable object attributes are pulled by the PDP from the local RPs usage monitor. The updates of mutable subject attributes are performed by the PDP, and the updates of mutable object attributes are captured by the local usage monitor. Any update of subject or object attributes and any change of system conditions triggers the re-evaluation of the policy by the PDP according to the ongoing usage session and may result in revocation of the ongoing usage or update of attributes if necessary.

They integrated the enforcement architecture in the context of the Grid Security Infrastructure (GSI) [10]. The architecture includes three main components within a VO: user platforms, individual resource providers (RPs), and an attribute repository (AR). AR is a centralized service to store and push mutable subject and system attributes in a VO. Object attributes are stored in a usage monitor (UM) on each RP side.

Within their work, both Martinelli and Mori and Zhang *et al.* focused on GRID computational services. We argue that the adaptation of UCON to Data Grid poses a greater number of issues to be solved. He hope this paper will highlight a good number of them. Moreover, none of the previous prototype is specifically applied to an actual (Data)

[2]web address: `http://www.globus.org`

Grid architecture. We think that, in order for usage control to be applied on production Grids, the OGSA work on Grid authorization should be better evaluatued and considered. The next section closes this gap.

3.2 A Usage-Based Grid Authorization Architecture

The OGF's OGSA authorization working group [3] provides an information document reviewing the functional components of Grid service provider authorisation service middleware [6]. 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 (IdP) or an Attribute Authority (AA) (e.g. the Virtual Organization Membership Service (VOMS) [1]). The credential can be embedded in an Attribute Certificate extension [8], and/or in a proxy certificate [26], or using a SAML [5] token. Credentials can then be validated by a Credential Validation Service (CVS), that return the valid attributes of the subject. Others fuctional components comprise: the Policy Decision Point (PDP), viz. the functional component responsible for returning an authorisation decision given the users access request and the users valid attributes; the Policy Enforcement Point (PEP), which enforces the results returned from a policy engine (normally a PDP); and the Context Handler (CH), responsible for handling the communications between PEPs, CVSs and PDPs. The interactions between these functional components can be constructed in four different ways, according as the credentials and the authorization decisions are pulled or pushed. For example, picture 4 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.

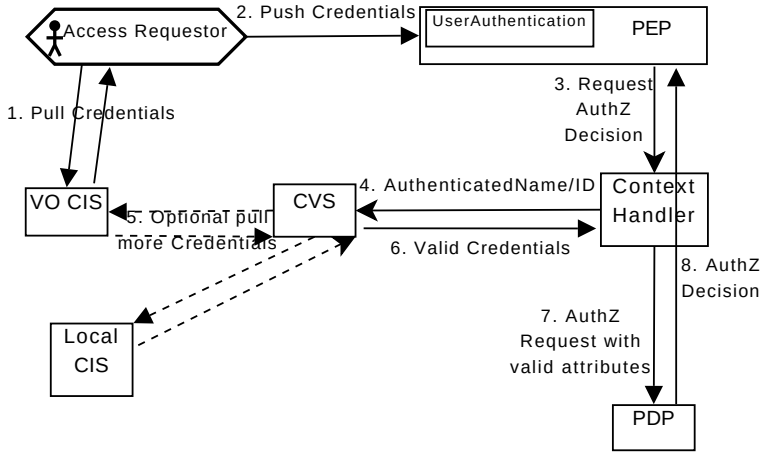


Figure 4: A Grid authorization architecture where Credentials are pushed

Passing from a Grid authorization architecture to a usage-based Grid authorization architecture doesn't require changes the way the fuctional components interact each others. Picture 5 shows a usage-based Grid authorization architecture with the same functional component interactions of picture 4. There are the following differences:

1. from an UCON point of view, valid attributes released by a CVS are examples of *immutable* (persistent) attributes (e.g., the VOMS' role and group membership);
2. A complex UCON PDP should be able to evaluate policies where the predicates are statements about the subjects' and objects' attributes. Three sub-components, namely the *Reference Monitor*, the *Predicate Validator* and the *Attribute Manager*, make up the UCON PDP. They are explained with details in section 4;
3. External components are needed to supply the UCON PDP with the needed information:
   - An *VO 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;
   - An *VO's attributes repository* stores the mutable attributes of the subjects;

[3] web address: `http://forge.gridforum.org/sf/projects/ogsa-authz`

- An *RP's attributes repository* stores the mutable attributes of the objects.

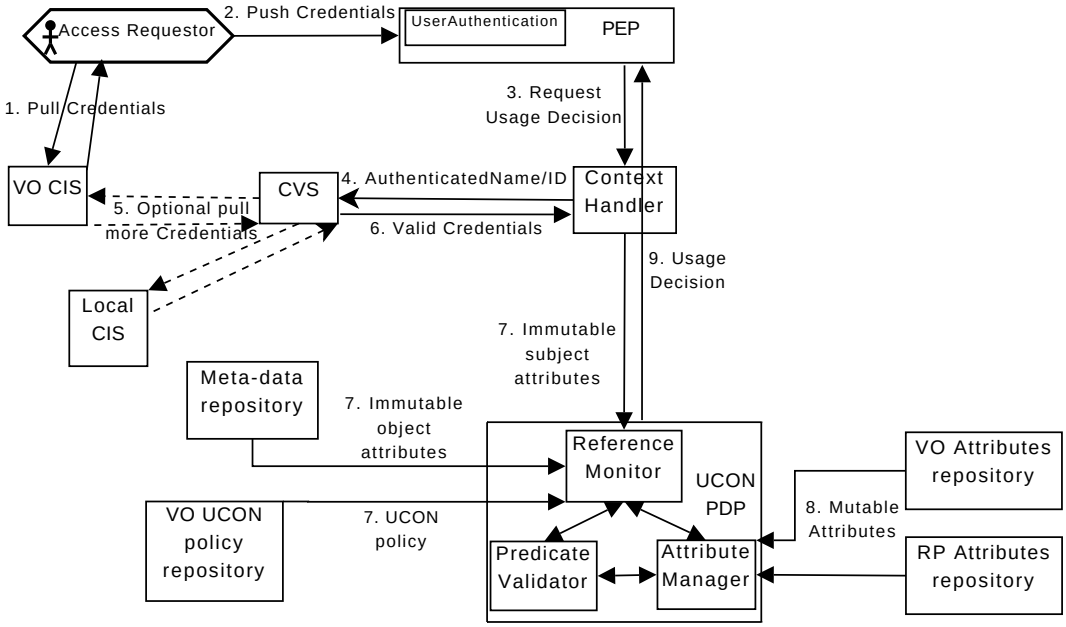


Figure 5: A usage-based Grid authorization architecture where Credentials are pushed

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 are pulled by the PDP from the VO's centralized attribute repository, and mutable object attributes are pulled by the PDP from the local RPs usage monitor, which records the temporal and dynamic properties of the object. The updates of mutable subjects' and objects' attributes are performed by the Attribute Manager sub-component.

When a UCON PDP is used to control the usage permissions of the resources managed by a DGMS, the following restrictions must be applied:

- 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. We remind that the abstract name is a unique, virtual, data identifier;
- 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*;
- An **immutable object attribute** is a persistent security description of the abstract name. An example can be represented by the *privacy* level;
- An **RP's attributes repository** stores mutable security attributes of the abstract names.

We remind that, within this paper, we only deal with the UCON_a family of core models, so that the usage-based Grid authorization architecture presented here doesn't take in consideration *obligations* and *conditions*.

3.3 Usage Control in Semantic Grids

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. In the scenario of a DGMS managing a global namespace with billions of entries, the creation, definition and enforcement of usage control policies may represent an issue in

terms of management, scalability, and consistency. For example, in current hierarchical file systems, access control is made specifying the authorizations on everyone of billions of files.

If usage control techniques want to be really useful in a large, pervasive, environment, it should be able to solve those 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) [9]. In the implementations of these access control models, when an authorization policy change 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 user or roles make worse the possibility of the authorization system being managed in an effective way. What's needed is a mechanism for regulating the policy granularity. We think that semantic binding assertions regarding Grid users and resources, as exposed in a Semantic Grid environment [7], could be used to regulate the usage control granularity. UCON subjects and objects should be semantic concepts extracted from, for example, those VO ontologies or scientific model ontologies used in the Semantic Grid.

Since the preliminary Tim Berners-Lee's vision of the web evolution [4], the Semantic Web is a field that received 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[4] proved the interest of the Grid community in Semantic Web techologies. 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 [7], 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.

Within this paper, we are not interested in the techologies, specifications, data interchange formats or notations used in the Semantic Grid context. Instead, we want to give a preliminary highlight on the advantages deriving from a semantic-aware usage control service.

In UCON, each subject and object is associated with attributes: subjects' and objects' attributes are properties or capabilities that can be used for the usage decision process. Park and Sandhu state in [19] that subjects and objects are defined and represented by their respective attributes. This sentence could be source of misunderstanding: 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 policy (written in POLPA):

```
1 TryAccess(John_Doe, file_xyz, read).
2 PredicateValidation([John_Doe.openedFiles < John_Doe.MAX_openedFiles]).
3 AttributeUpdate(John_Doe.openedFiles, add, 1).
4 PermitAccess(John_Doe, file_xyz, read).
5 EndAccess(John_Doe, file_xyz, read).
```

Within this policy, the UCON subject is the (Grid) user `John_Doe`, the UCON object is the (Grid) file `file_xyz`, and the UCON (Grid) right requested is simply `read`. This policy makes use of a couple of John_Doe's attributes in the predicate at line 2, `openedFiles` and `MAX_openedFiles`, and updates `openedFiles` at line 3. 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 by John_Doe. 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.

Instead, Semantic Grid technologies can come in play for the definition of the UCON subjects and objects. In a Semantic Grid, following the terminology introduced in [7], each *Grid Entity* is associated to a *Knowledge Entity* through a *Semantic Binding*. Knowledge Entities are special types of Grid Entities that represent or could operate with some form of knowledge. Examples of Knowledge Entities 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 Knowledge Entities.

A semantic-aware usage control service is depicted in figure 6. This service is similar to the one presented in section 3.2 in picture 5. In a Semantic Grid, the access requestor (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 Entity. For what concern the DGMS, the meta-data repository can be used to store the Knowledge Entity of the abstract name. Specific Grid Users keep asking

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

to access specific Grid Data, but in a semantic-aware usage control service the reaserch for the applicable policies is done on the multiple fields of the Knowledge Entities of both the Grid user and the resource to be accessed. This way, two or more policies could be applicable for a single access request, thus generating more than a single usage control process for what is, instead, a single access request. This can generate some confusion, especially when two or more UCON OnA policies happens to be in conflict each others. A way to keep track of all the policies involved is then needed, as well as a policy conflict analysis tool. Instead, since in UCON the closed system assumption is in force, if no policy is applicable, the access is denied.

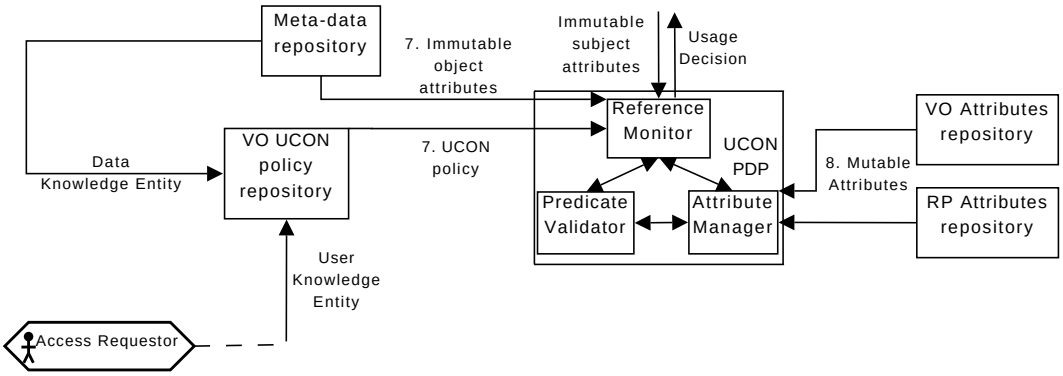
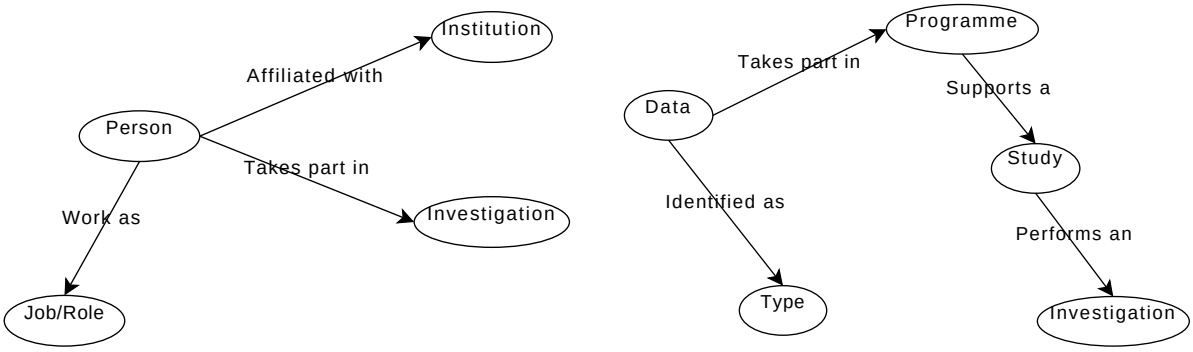


Figure 6: A semantic-aware usage control service

An example of Knowledge Entity representing the Grid Entity *Grid User* is shown in picture 7(a), while picture 7(b) shows an example of a Knowledge Entity for a *Grid Data* stored in Data Grid. A semantic-aware DGMS could associate a data Knowledge Entity like this one to each of the managed *abstract names*. The graph of picture 7(a) is liberally inspired from [7], while the graph of picture 7(b) has been liberally derived from the CCLRC scientific metadata model [25]. These examples are not meant to be complete. We also note that a *Grid Data* description normally makes use of application-dependent metadata, thus in a real system a Knowledge Entity of a Grid Data 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 Grid Data is described not only by the *Type* (e.g. file, or stream), but also by the *Programme* of work, the supported *Study*, and by an *Investigation*. The interested reader should refer to [25] 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".



(a) An example for a Grid user Knowledge Entity

(b) An example for a data Knowledge Entity

Figure 7: Examples of Knowledge Entities for users and data

A security administrator can control the usage control granularity using the semantic fields shown in pictures 7(a) and 7(b) for the definition of collective policies, like the following simple PreA_0 policy:

```
1 TryAccess(Institution:STFC, Study:ISIS, read).
```

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

This policy simply states that each User associated with STFC can read those Data pertaining to the ISIS study. The administrator could also associate UCON attributes to the UCON subject `Institution:STFC` and to the UCON object `Study:ISIS` and ask for the validation of predicates using those attributes.

The possibility to control the policy granularity 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.

4 An Abstract Specification of Enforcement Mechanism for Usage Control

Within this section, we show an abstract specification of an enforcement mechanism for UCON$_a$ policies, using the requirement specification language provided by KAOS. Within this paper we use KAOS with a bottom-up approach: rather than deriving a specification using the KAOS standard methodology as presented in section 2.3, we first present a complete specification and then, in the next sections, we will apply the KAOS requirement engineering methodology to each of the UCON$_a$ sub-models to prove that what is presented here is sound and complete. The specification has been partially abstracted from the usage-based authorization architecture of section 3.2, while the operations are inferred from the UCON formal representation presented in [32] and in section 2.2. Picture 8 shows a graphical representation of the UCON PDP components (as KAOS agents) and the operations those components can perform.

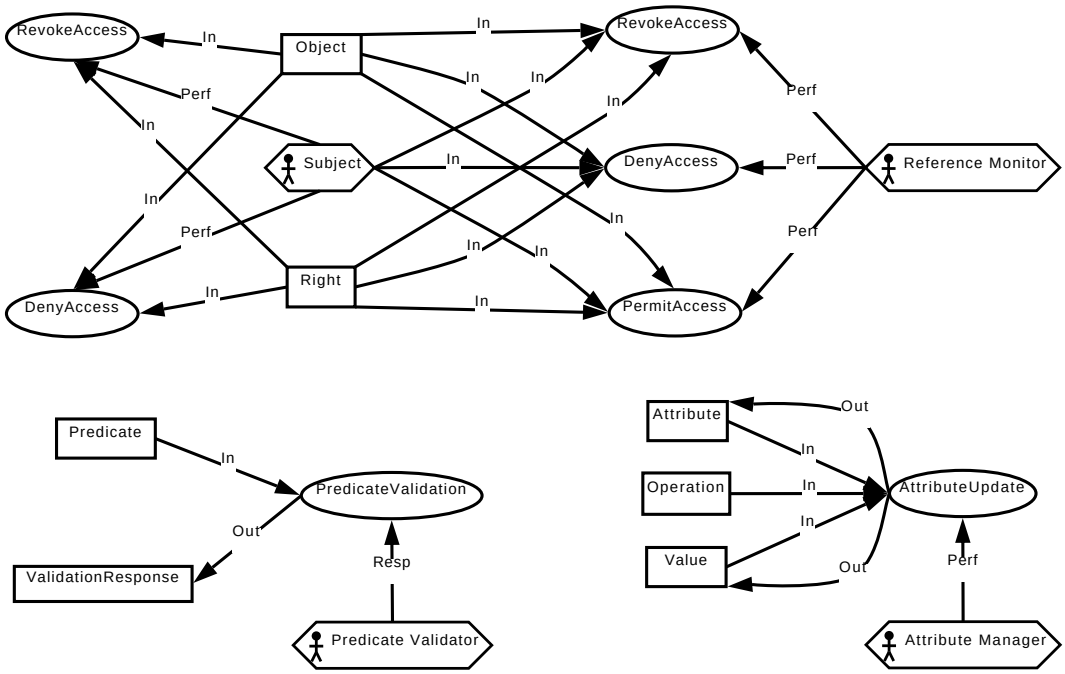


Figure 8: Abstract specification of an UCON$_a$ enforcement mechanism

As you can see from picture 8, we identify three agents:

- the **Attribute Manager** (AM) update the attributes and return their values;
- the **Predicate Validator** (PV) takes care of validating the policy predicates;
- the **Reference Monitor** (RM) is a gateway for all the authorization decisions, of the DGMS, or whatever is the client.

The RM can receive `TryAccess` and `EndAccess` invocations, and is responsible to issue the `PermitAccess`, `DenyAccess` or `RevokeAccess` operations. The PV can be invoked for the validation of the predicates, viz. performing the `PredicateValidation` operation. The AM can be invoked for the update of the UCON attributes with the `AttributeUpdate` operation.

It should be noted that simple Grid users should not be able to deal directly with the enforcement mechanism: rather, as shown in section 3.2, Grid users should contact just the DGMS, asking for a data represented by a Human Oriented Name. Then the DGMS shall ask for an authorization response from the enforcement mechanism regarding the user, as it is recognized by the DGMS, and the abstract name the user has requested to access.

We now provide a written operational software specification of most of the operations shown in picture 8, using the KAOS operation model. `TryAccess` and `EndAccess` are not specified here since they are issued by the users, which we consider as an agent in the environment and thus not part of the enforcement mechanism. Each operation defines a state-transition in the application domain, defined through *domain pre-* and *post-conditions*. The operations can have input and output fields; for example, a *subject*, an *object* and a *right* are input to the operations `PermitAccess`, `DenyAccess` and `RevokeAccess`.

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

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- Condition:
PV.validate($\neg$ $p_1 \vee \ldots \vee \neg$ p_n)

Operation: RevokeAccess
Performed By: Reference Monitor
Domain Pre-Condition: $\neg$ RM.RevokeAccess(s, o, r)
Domain Post-Condition: RM.RevokeAccess(s, o, r)
Input: subject, object, right
ReqPre- Condition:
PV.validate($\neg$ $p_1 \vee \ldots \vee \neg$ p_n)

Operation: PredicateValidation
Performed By: Predicate Validator
Domain Pre-Condition:
$\neg$ PV.validate($p_1 \wedge \ldots \wedge p_n$)
Domain Post-Condition:
PV.validate($p_1 \wedge \ldots \wedge p_n$)
Input: Predicate
Output: ValidationResponse

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

When specifying an operation in KAOS, an important distinction is made between (descriptive) domain *pre-/post-conditions* and (prescriptive) *pre-*, *post-* and *trigger conditions* required for achieving some goal(s). The *required pre-condition* for some goal 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 in order to achieve the corresponding goal. The *required trigger condition* for some goal captures an obligation to perform the operation if the condition becomes true provided the domain precondition is true.

Most of the operations presented above don't specify any *pre-*, *post-* or *trigger conditions*, since these are dependent from the order the single operations are invoked. Such order is encoded in the UCON sub-models. For example, in the simplest case of a UCON PreA_0 model, a *PermitAccess* operation can be issued by the RM when the output of the *PredicateValidation* operation – *ValidationResponse* – is positive. Instead, in a PreA_1 model a *PermitAccess* operation can be issued only after the attributes are updated, viz. after the AM performed the *AttributeUpdate* operation requested. Moreover, in both of this policies, the PV should contact the AM for updated attribute information. Since a single policy can be a combination of multiple UCON core models, the sequentiality can be even more complicated.

In the next sections we will demonstrate how the same operations described here, when using prescriptive conditions, are capable to enforce all the UCON_a sub-models.

5 Using KAOS for a Formal Specification Proof

Within this section, we'll use KAOS to refine UCON policies, deriving an enforcement mechanism for each UCON$_a$ sub-model. Policy refinement concerns with transforming a high-level and abstract policy specification into a low-level and concrete one [16]. It includes (1) determining the resources that are needed to satisfy the requirements of a policy; (2) translating the high-level policies into operational policies that can be enforced; and (3) 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 [3], 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. enforced — and are assigned to agents. Goals can be formalised using linear temporal logic (LTL) [29], which is the formal language used to define the semantic of UCON [32]. 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:

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

Within this section, we apply the KAOS goal model to prove that the abstract specification as shown in section 4 is capable to enforce all the UCON$_a$ core models. We start from a general refinement to justify the need for an enforcing mechanism; considering that:

∀ s:subject, o:object, r:right
  permitaccess(s, o, r) ⇒ requireToAccess(s, o, r)

Applying a *milestone pattern* [27] we refine the previous goal in the following two goals:

∀ s:subject, o:object, r:right
  permitaccess(s, o, r) ⇒ policyEnforcing(s, o, r)

∀ s:subject, o:object, r:right
  policyEnforcing(s, o, r) ⇒ requireToAccess(s, o, r)

Next, we need to clearly define the meaning of the `policyEnforcing(s,o,r)` predicate. There are two different ways to do this. The first one consists in following the methodology proposed by van Lamsweerde in [28], where the top-level goal consists in a precise definition of the policy. For doing it, we may consider using the formal policy specifications given by Sandhu in [32]. This way is impractical for many reasons. First of all, many of the Sandhu's specifications are very difficult (if not impossible) to refine using the KAOS methodology. Even if a re-definition of those policies is possible, this would lead us to unexpected results. Consider for example the following PreA$_0$ policy, as defined in [32]:

∀ s:subject, o:object, r:right
  permitaccess(s, o, r) ⇒ ♦ (tryaccess(s, o, r) ∧ (p$_1$ ∧ … ∧ p$_n$))

This policy can't be refined further with the KAOS methodology, so we initially considered giving the following re-definition:

∀ s:subject, o:object, r:right
  permitaccess(s, o, r) ⇒ ♦ tryaccess(s, o, r) ∧ ♦ (p$_1$ ∧ … ∧ p$_n$)

Even if this new definition is easily refinable, it would lead us to unconsistent results. In a PreA$_0$ policy the `tryaccess` action should be followed by a predicates validation. Neither the former nor the latter policy encode this information about sequentiality. The only way we know on how to specify such a constraint in LTL is to encode the policy in the following formula:

$\forall$ s:subject, o:object, r:right
permitaccess(s, o, r) $\Rightarrow$ ♦ (♦ tryaccess(s, o, r) $\wedge$ ($p_1 \wedge \ldots \wedge p_n$))

which is not further refinable and thus unusable.

The second methodology is the following: for each UCON_a 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 model need to be enforced only before the access is actually granted. Instead, each policy pertaining to a PreA_3 model need to be enforced not only before the access, but also after the end of it. Moreover, each OnA 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.

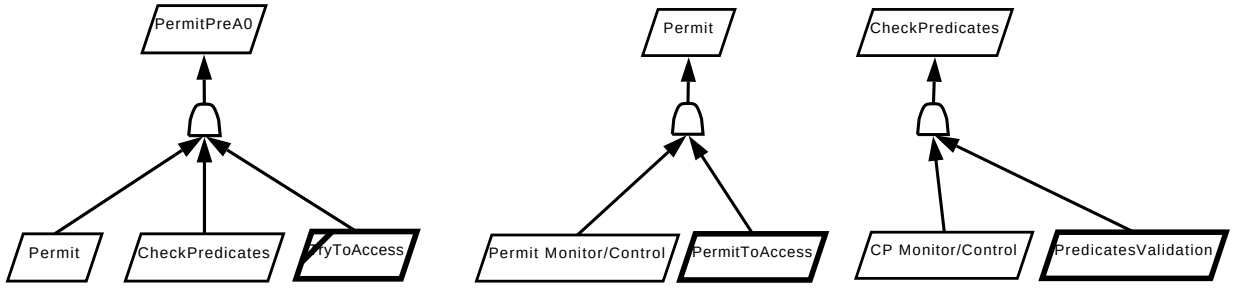
5.1 UCON PreA_0

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 is then the following:

Goal [PermitPreA0]
RefinedTo: [Permit], [CheckPredicates], [TryToAccess]
FormalDef: ($\forall$ s:subject, o:object, r:right)
permitaccess(s, o, r) $\Rightarrow$ • policyEnforcing(s, o, r)

We then apply a first goal refinement as shown in picture 9(a), while the formal sub-goals' definitions follow. We can use tools such as the FAUST toolkit [20] to demostrate that the refinement is correct.



(a) Initial goal refinement of an UCON PreA_0 core model

(b) Completion of the goal refinement of the UCON PreA_0 core model

Figure 9: Goal model for 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)
⇒ • ($p_1 \wedge \ldots \wedge p_n$)

Goal [CheckPredicates]
Refines: [PermitPreA0]
RefinedTo: [CP Monitor/Control], [PredicatesValidation]
FormalDef: (∀ s:subject, o:object, r:right)
($p_1 \wedge \ldots \wedge p_n$)
⇒ • tryaccess(s, o, r)

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

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 picture 9(b) 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 [13]. We identify two requirement goals, [PermitToAccess] and [PredicatesValidation], and assign two agents, the *Reference Monitor* and the *Predicate Validator* to respectively take care to each of them.

Goal [Permit Monitor/Control]
Refines: [Permit]
FormalDef: (∀ s:subject, o:object, r:right, RM:Reference Monitor, PV:Predicate Validator)
permitaccess(s, o, r) ⇔ RM.permitaccess(s, o, r)
($p_1 \wedge \ldots \wedge p_n$) ⇔ PV.validate($p_1 \wedge \ldots \wedge p_n$)

Goal [PermitToAccess]
Refines: [Permit]
FormalDef: (∀ s:subject, o:object, r:right, RM:Reference Monitor, PV:Predicate Validator)
RM.permitaccess(s, o, r)
⇒ • PV.validate($p_1 \wedge \ldots \wedge p_n$)
Resp: Reference Monitor

Goal [CP Monitor/Control]
Refines: [CheckPredicates]
FormalDef: (∀ s:subject, o:object, r:right, PV:Predicate Validator)
($p_1 \wedge \ldots \wedge p_n$) ⇔ PV.validate($p_1 \wedge \ldots \wedge p_n$)

Goal [PredicatesValidation]
Refines: [CheckPredicates]
FormalDef: (∀ s:subject, o:object, r:right, PV:Predicate Validator)
PV.validate($p_1 \wedge \ldots \wedge p_n$)
⇒ • tryaccess(s, o, r)
Resp: Predicate Validator

We are now capable to derive the KAOS agent and operation models. Picture 10 shows the KAOS operation model, together with the agent/responsibility model. As the reader can see, we identify a couple of operations: `PermitAccess` and `PredicateValidation`.

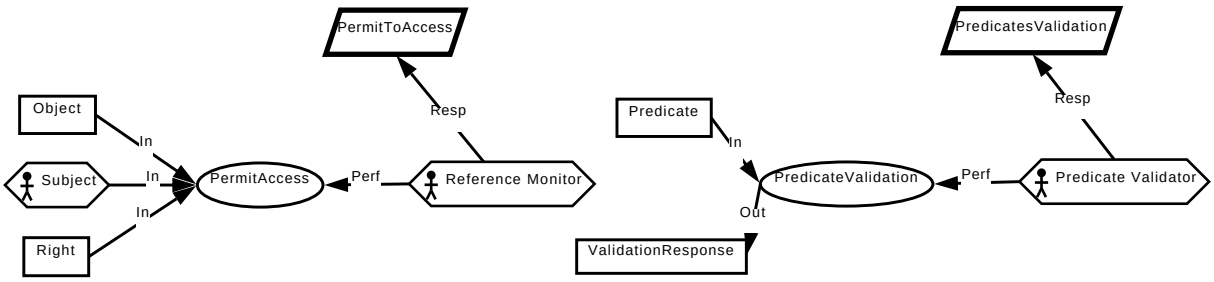


Figure 10: Excerpt of the operation model for an UCON $PreA_0$ enforcement mechanism

Next follows the KAOS operational specification for the UCON $PreA_0$ enforcement mechanism, derived using the KAOS operationalization patterns presented in [14]. 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:
$\neg$ RM.permitaccess(s, o, r)
Domain Post-Condition:
RM.permitaccess(s, o, r)
Input: subject, object, right
ReqPre for [PermitToAccess]:
PV.validate($p_1 \wedge \ldots \wedge p_n$)

Operation: PredicateValidation
Performed By: Predicate Validator
Domain Pre-Condition:
$\neg$ PV.validate($p_1 \wedge \ldots \wedge p_n$)
Domain Post-Condition:
PV.validate($p_1 \wedge \ldots \wedge p_n$)
Input: Predicate
Output: ValidationResponse
ReqPre for [PredicatesValidation]:
tryaccess(s, o, r)

The only difference between these operations and those shown in section 4 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. They are dependent from the order of the operations as specified by the model definition. Other UCON models encode a different sequentiality of the operations. Within the rest of this paper we'll show the KAOS operational specification for all the UCON$_a$ sub-models. We'll show that the derived operations always encode the same state-transitions as specified by those in section 4, 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$_a$ sub-model. A similar approach was introduced in [3]. 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.

5.2 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 5.1, and same can be said for all the next paragraph of section 5. 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) $\Rightarrow \bullet$ policyEnforcing(s, o, r)

We then apply a first goal refinement as shown in picture 11(a), while the formal sub-goals' definitions follow in the text.

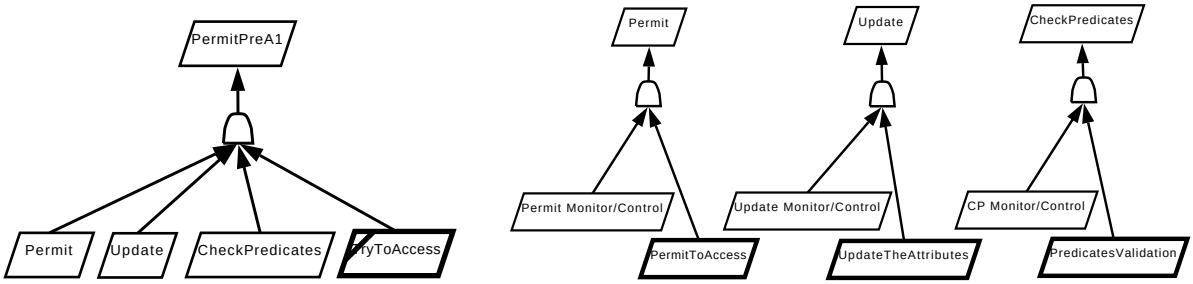
Goal [Permit]
Refines: [PermitPreA1]
RefinedTo:
[Permit Monitor/Control],
[PermitToAccess]
FormalDef: ($\forall$ s:subject,
o:object,
r:right)
permitaccess(s, o, r)
$\Rightarrow \bullet$ update(s,o,r)

Goal [Update]
Refines: [PermitPreA1]
RefinedTo:
[Update Monitor/Control]
[UpdateTheAttributes]
FormalDef: ($\forall$ s:subject,
o:object,
r:right)
update(s, o, r)
$\Rightarrow \bullet$ ($p_1 \wedge \ldots \wedge p_n$)

Goal [CheckPredicates]
Refines: [PermitPreA1]
RefinedTo:
[CP Monitor/Control],
[PredicatesValidation]
FormalDef: ($\forall$ s:subject,
o:object,
r:right)
($p_1 \wedge \ldots \wedge p_n$)
$\Rightarrow \bullet$ tryaccess(s, o, r)

Goal [TryToAccess]
Refines: [PermitPreA1]
FormalDef: ($\forall$ s:subject,
o:object,
r:right)
tryaccess(s, o, r)
$\Rightarrow \bullet$ policyEnforcing(s, o, r)

The goals [Update], [Permit] and [CheckPredicates] are not final goals, so they have to be refined further on. In picture 11(b) is shown the completion of the goal refinement, and the formal definitions of each sub-goals follow in



(a) Initial goal refinement of an UCON PreA_1 core model

(b) Completion of the goal refinement of the UCON PreA_1 core model

Figure 11: Goal model for an UCON PreA_1 core model

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 *Predicate Validator*.

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,
PV:Predicate Validator)
update(s, o, r) ⇔ AM.update(s, o, r)
$(p_1 \wedge \ldots \wedge p_n)$ ⇔ PV.validate$(p_1 \wedge \ldots \wedge p_n)$

Goal [UpdateTheAttributes]
Refines: [Update]
FormalDef: (∀ s:subject, o:object, r:right,
AM:AttributeManager,
PV:Predicate Validator)
AM.update(s, o, r)
⇒ • PV.validate$(p_1 \wedge \ldots \wedge p_n)$
Resp: Attribute Manager

Goal [CP Monitor/Control]
Refines: [CheckPredicates]
FormalDef: (∀ s:subject, o:object, r:right,
PV:Predicate Validator)
$(p_1 \wedge \ldots \wedge p_n)$ ⇔ PV.validate$(p_1 \wedge \ldots \wedge p_n)$
tryaccess(s, o, r) ⇔ tryaccess(s, o, r)

Goal [Predicates Validation]
Refines: [CheckPredicates]
FormalDef: (∀ s:subject, o:object, r:right,
PV:Predicate Validator)
PV.validate$(p_1 \wedge \ldots \wedge p_n)$
⇒ • tryaccess(s, o, r)
Resp: Predicate Validator

Picture 12 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.

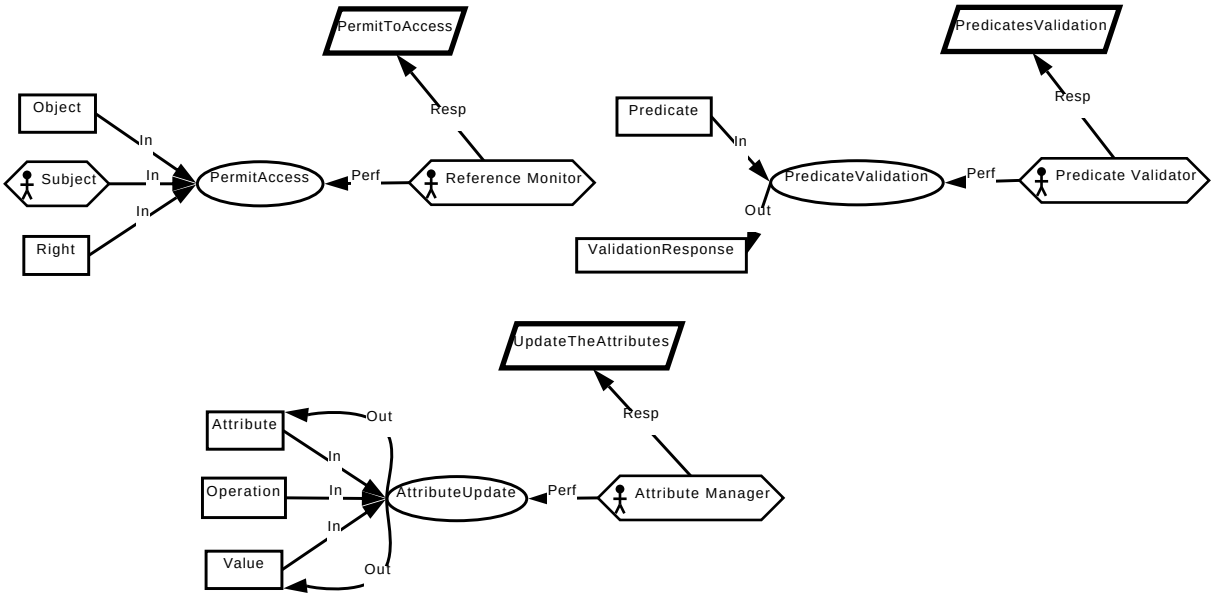


Figure 12: Excerpt of the operation model for an UCON PreA_1 enforcement mechanism

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]:
AM.update(s, o, r)

Operation: AttributeUpdate
Performed By: Attribute Manager
Domain Pre-Condition:
¬ AM.update(att)
Domain Post-Condition:
AM.update(att)
Input: Attribute, Operation, Value
Output: Attribute, Value
ReqPre for [UpdateTheAttributes]:
PV.validate($p_1 \wedge \ldots \wedge p_n$)

Operation: PredicateValidation
Performed By: Predicate Validator
Domain Pre-Condition:
¬ PV.validate($p_1 \wedge \ldots \wedge p_n$)
Domain Post-Condition:
PV.validate($p_1 \wedge \ldots \wedge p_n$)
Input: Predicate
Output: ValidationResponse
ReqPre for [Predicates Validation]:
tryaccess(s, o, r)

As in section 5.1, the only difference between these operations and those shown in section 4 is the specification of the *Required Pre-Condition* clause.

5.3 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: $\texttt{s.expense}' = \texttt{s.expense} + \texttt{o.cost}$.

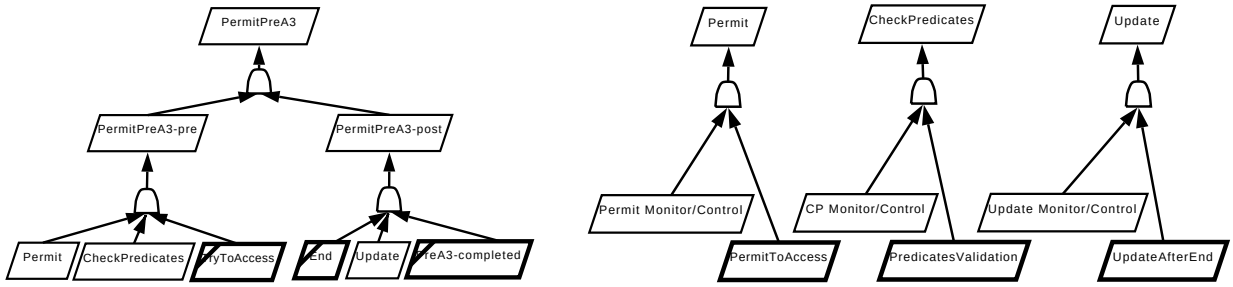
The policy enforcing happens before and after the access is permitted. The top-goal, [PermitPreA3], is easily refined in the goals [PermitPreA3-pre] and [PermitPreA3-post] as specified below.

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

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

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

The first part of the goal refinement is shown in picture 13(a). The formal sub-goals' definitions follow.



(a) Initial goal refinement of an UCON PreA_3 core model

(b) Completion of the goal refinement of the UCON PreA_3 core model

Figure 13: Goal model for an UCON PreA_3 core model

Goal [Permit]
Refines: [PermitPreA3-pre]
RefinedTo: [Permit Monitor/Control], [PermitToAccess]
FormalDef: (∀ s:subject, o:object, r:right)
permitaccess(s, o, r)
⇒ • ($p_1 \wedge \ldots \wedge p_n$)

Goal [CheckPredicates]
Refines: [PermitPreA3-pre]
RefinedTo: [CP Monitor/Control], [PredicatesValidation]
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)
⇒ • policyEnforcing(s, o, r)

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)
⇒ ○ policyEnforcing(s, o, r)

The goals [Permit], [CheckPredicates] and [Update] have to be refined further. In picture 13(b) is shown the completion of the goal refinement, 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*, *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, PV:Predicate Validator)
permitaccess(s, o, r) ⇔ RM.permitaccess(s, o, r)
($p_1 \wedge \ldots \wedge p_n$) ⇔ PV.validate($p_1 \wedge \ldots \wedge p_n$)

Goal [CP Monitor/Control]
Refines: [CheckPredicates]
FormalDef: (∀ s:subject, o:object, r:right, PV:Predicate Validator)
($p_1 \wedge \ldots \wedge p_n$) ⇔ PV.validate($p_1 \wedge \ldots \wedge p_n$)
tryaccess(s, o, r) ⇔ tryaccess(s, o, r)

Goal [PermitToAccess]
Refines: [Permit]
FormalDef: (∀ s:subject, o:object, r:right, RM:Reference Monitor, PV:Predicate Validator)
RM.permitaccess(s, o, r)
⇒ • PV.validate($p_1 \wedge \ldots \wedge p_n$)
Resp: Reference Monitor

Goal [PredicatesValidation]
Refines: [CheckPredicates]
FormalDef: (∀ s:subject, o:object, r:right, PV:Predicate Validator)
PV.validate($p_1 \wedge \ldots \wedge p_n$)
⇒ • tryaccess(s, o, r)
Resp: 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 picture of the KAOS operation model for this UCON sub-model is not shown, since it's pretty much the same as of picture 12. The formal specifications of the three operations are shown next:

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]:
PV.validate($p_1 \wedge \ldots \wedge p_n$)

Operation: PredicateValidation
Performed By: Predicate Validator
Domain Pre-Condition:
¬ PV.validate($p_1 \wedge \ldots \wedge p_n$)
Domain Post-Condition:
PV.validate($p_1 \wedge \ldots \wedge p_n$)
Input: Predicate
Output: ValidationResponse
ReqPre for [PredicatesValidation]:
tryaccess(s, o, r)

Operation: AttributeUpdate
Performed By: Attribute Manager
Domain Pre-Condition:
¬ AM.update(att)
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 4 is the specification of the *Required Pre-Condition* clause.

5.4 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` ∈ `CRL`.

The top goals are following.

Goal [PermitOnA0]
RefinedTo: [PermitOnA0-pre]
[PermitOnA0-post]
FormalDef: (∀ s:subject,
o:object,
r:right)
permitaccess(s, o, r)
⇒ • policyEnforcing(s, o, r) ∧
□ policyEnforcing(s, o, r)

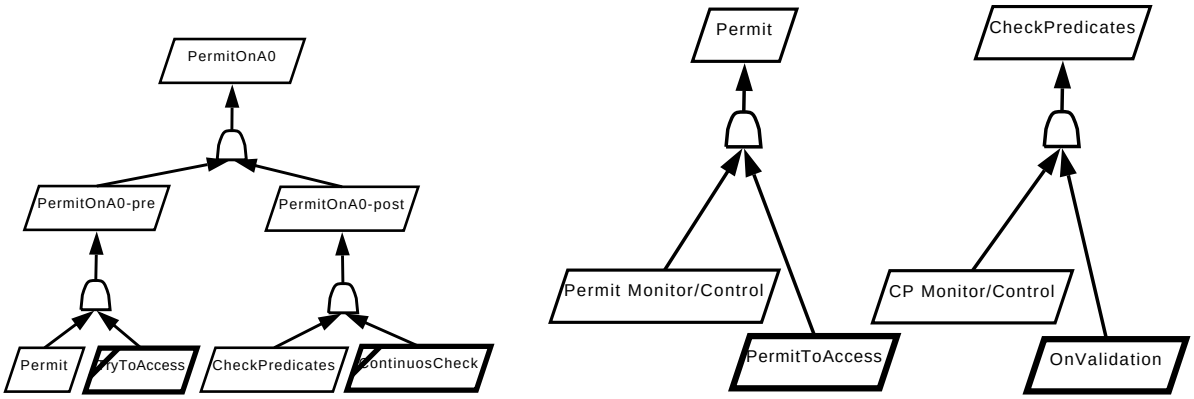
Goal [PermitOnA0-pre]
RefinedTo: [Permit],
[TryToAccess]
FormalDef: (∀ s:subject,
o:object,
r:right)
permitaccess(s, o, r)
⇒ • policyEnforcing(s, o, r)

Goal [PermitOnA0-post]
RefinedTo: [CheckPredicates],
[ContinuosCheck]
FormalDef: (∀ s:subject,
o:object,
r:right)
permitaccess(s, o, r)
⇒ □ policyEnforcing(s, o, r)

The first and second part of the goal refinements are shown in picture 14(a) and 14(b), while the formal sub-goals' definitions follow.

Goal [Permit]
Refines: [PermitOnA0-pre]
RefinedTo: [Permit Monitor/Control],
[PermitToAccess]
FormalDef: (∀ s:subject, o:object, r:right)
permitaccess(s, o, r)
⇒ • tryaccess(s, o, r)

Goal [TryToAccess]
Refines: [PermitOnA0-pre]
FormalDef: (∀ s:subject, o:object, r:right)
permitaccess(s, o, r) ⇒ • policyEnforcing(s, o, r)



(a) Initial goal refinement of an UCON OnA_0 core model

(b) Completion of the goal refinement of the UCON OnA_0 goal model

Figure 14: Goal and operation model for an UCON OnA_0 core model

Goal [CheckPredicates]
Refines: [PermitOnA0]
RefinedTo: [CP Monitor/Control], [OnValidation]
FormalDef: (∀ s:subject, o:object, r:right)
permitaccess(s, o, r)
⇒ □ ($p_1 \wedge \ldots \wedge p_n$)

Goal [ContinuosCheck]
Refines: [PermitPreA0]
FormalDef: (∀ s:subject, o:object, r:right)
($p_1 \wedge \ldots \wedge p_n$)
⇒ □ policyEnforcing(s, o, r)

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

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

Goal [CP Monitor/Control]
Refines: [CheckPredicates]
FormalDef: (∀ s:subject, o:object, r:right, RM:Reference Monitor, PV:Predicate Validator)
permitaccess(s, o, r) ⇔ RM.permitaccess(s, o, r)
($p_1 \wedge \ldots \wedge p_n$) ⇔ PV.validate($p_1 \wedge \ldots \wedge p_n$)

Goal [OnValidation]
Refines: [CheckPredicates]
FormalDef: (∀ s:subject, o:object, r:right, RM:Reference Monitor, PV:Predicate Validator)
RM.permitaccess(s, o, r)
⇒ □ PV.validate($p_1 \wedge \ldots \wedge p_n$)
Resp: Predicate Validator

The requirement goals are [PermitToAccess] and [OnValidation]. The agents assigned to them are respectively the *Reference Monitor* and the *Predicate Validator*.

The KAOS operation model is pretty much the same as of picture 10 and therefore it's not shown here. The KAOS operational specifications for the UCON OnA_0 enforcement mechanism follow in the text.

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]:
tryaccess(s, o, r)

Operation: PredicateValidation
Performed By: Predicate Validator
Domain Pre-Condition:
¬ RM.permitaccess(s, o, r)
Domain Post-Condition:
RM.permitaccess(s, o, r)
Input: Predicate
Output: ValidationResponse
ReqPost for [OnValidation]:
PV.validate($p_1 \wedge \ldots \wedge p_n$)

The only difference between these operations and their equivalent shown in section 4 is the specification of the *Required Pre-Condition* and *Required Post-Condition* clauses.

5.5 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 5.4, with the further constraint that Bob can't 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` $+1$.

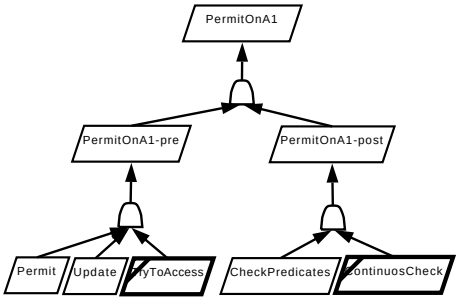
The policy enforcing happens before and during the access is permitted. The top goals follow:

Goal [PermitOnA1]
RefinedTo: [PermitOnA1-pre]
[PermitOnA1-post]
FormalDef: (∀ s:subject,
o:object,
r:right)
permitaccess(s, o, r)
⇒ • policyEnforcing(s, o, r) ∧
□ policyEnforcing(s, o, r)

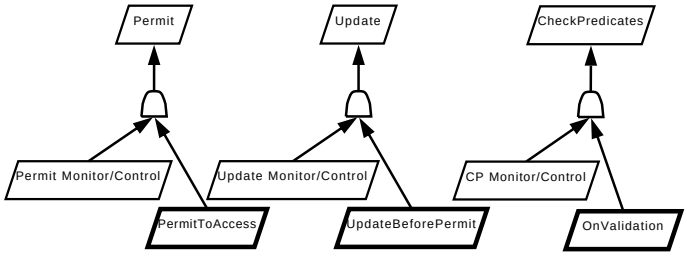
Goal [PermitOnA0-pre]
RefinedTo: [Permit],
[Update],
[TryToAccess]
FormalDef: (∀ s:subject,
o:object,
r:right)
permitaccess(s, o, r)
⇒ • policyEnforcing(s, o, r)

Goal [PermitOnA0-post]
RefinedTo: [CheckPredicates],
[ContinuosCheck]
FormalDef: (∀ s:subject,
o:object,
r:right)
permitaccess(s, o, r)
⇒ □ policyEnforcing(s, o, r)

The first and second part of the goal refinements are shown in picture 15(a) and 15(b), while the formal sub-goals' definitions follow.



(a) Initial goal refinement of an UCON OnA_1 core model



(b) Completion of the goal refinement of the UCON OnA_1 goal model

Figure 15: Goal and operation model for an UCON OnA_1 core model

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]
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)
⇒ • policyEnforcing(s, o, r)

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)
update(s, o, r) ⇔ AM.update(s, o, r)

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 refinement of the [PermitOnA1-post] goal is the same as the refinement of the [PermitOnA0-post] goal of section 5.4 and is not shown for brevity.

The requirement goals are [PermitToAccess], [UpdateBeforePermit] and [OnValidation]. The agents assigned to them are respectively the *Reference Monitor*, the *Attribute Manager* and the *Predicate Validator*.

The KAOS operation model, is the same as of picture 12. The formal specification of the KAOS operational specification for the UCON OnA_1 enforcement mechanism follows.

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]: AM.update(s, o, r)

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

Operation: PredicateValidation
Performed By: Predicate Validator
Domain Pre-Condition: ¬ RM.permitaccess(s, o, r)
Domain Post-Condition: RM.permitaccess(s, o, r)
Input: Predicate
Output: ValidationResponse
ReqPost for [OnValidation]: PV.validate($p_1 \wedge \ldots \wedge p_n$)

As usual, the only difference between these operations and their equivalents shown in section 4 is the specification of the *Required Pre-Condition* and *Required Post-Condition* clauses.

5.6 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:

Goal [PermitOnA2]
RefinedTo: [PermitOnA2-pre], [PermitOnA2-post], [PermitOnA2-on]
FormalDef: (∀ s:subject, o:object, r:right)

permitaccess(s, o, r) ⇒ • policyEnforcing(s, o, r) ∧ ○ policyEnforcing(s, o, r) ∧ □ policyEnforcing(s, o, r)

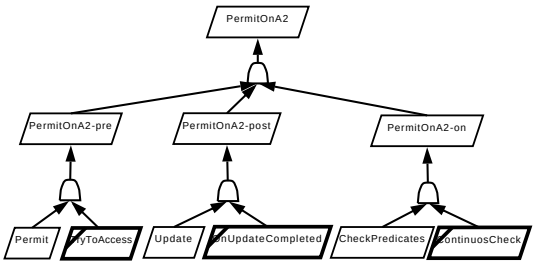
Which is easily refined in:

Goal [PermitOnA2-pre]
RefinedTo: [Permit]
[TryToAccess]
FormalDef: (∀ s:subject,
o:object,
r:right)
permitaccess(s, o, r)
⇒ • policyEnforcing(s, o, r)

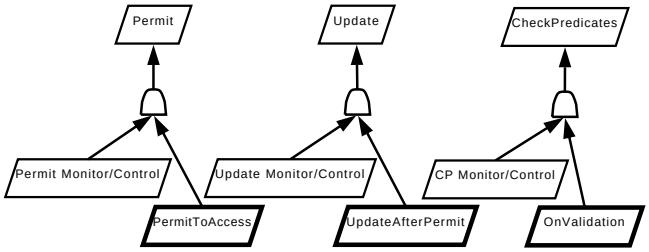
Goal [PermitOnA2-post]
RefinedTo: [Update]
[OnUpdateCompleted]
FormalDef: (∀ s:subject,
o:object,
r:right)
permitaccess(s, o, r)
⇒ ○ policyEnforcing(s, o, r)

Goal [PermitOnA2-on]
RefinedTo: [CheckPredicates]
[ContinuousCheck]
FormalDef: (∀ s:subject,
o:object,
r:right)
permitaccess(s, o, r)
⇒ □ policyEnforcing(s, o, r)

The first and second part of the goal refinements are shown in picture 16(a) and 16(b), while the formal sub-goals' definitions follow. The refinements of the goals [PermitOnA2-pre] and [PermitOnA2-on] are not shown here, since are the same as the refinement of respectively [PermitOnA0-pre] and [PermitOnA0-post] of section 5.4.



(a) Initial goal refinement of an UCON OnA_2 core model



(b) Completion of the goal refinement of the UCON OnA_2 goal model

Figure 16: Goal and operation model for an UCON OnA_2 core model

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

Goal [OnUpdateCompleted]
Refines: [PermitOnA2-post]
FormalDef: (∀ s:subject, o:object, r:right)
update(s, o, r)
⇒ ○ policyEnforcing(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 [UpdateBeforePermit]
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 requirement goals are [PermitToAccess], [UpdateAfterPermit] and [OnValidation]. The agents assigned to them are respectively the *Reference Monitor*, the *Attribute Manager* and the *Predicate Validator*.

The KAOS operation model, is the same as of picture 12. The formal specification of the KAOS operational specification for the UCON OnA_2 enforcement mechanism follows.

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]:
AM.update(s, o, r)

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

Operation: PredicateValidation
Performed By: Predicate Validator
Domain Pre-Condition:
¬ RM.permitaccess(s, o, r)
Domain Post-Condition:
RM.permitaccess(s, o, r)
Input: Predicate
Output: ValidationResponse
ReqPost for [OnValidation]:
PV.validate($p_1 \wedge \ldots \wedge p_n$)

As usual, the only difference between these operations and its equivalent shown in section 4 is the specification of the *Required Pre-Condition* and *Required Post-Condition* clauses.

5.7 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 5.5 can be completed with the following *postUpdate*: $\texttt{s.accessed\_files}' = \texttt{s.accessed\_files} - 1$.

The policy enforcing happens before, during and after the access is permitted. The top goal is:

Goal [PermitOnA3]
RefinedTo: [PermitOnA3-pre], [PermitOnA3-on], [PermitOnA3-post]
FormalDef: (∀ s:subject, o:object, r:right)
permitaccess(s, o, r) ⇒ • policyEnforcing(s, o, r) ∧ □ policyEnforcing(s, o, r) ∧ ◊ policyEnforcing(s, o, r)

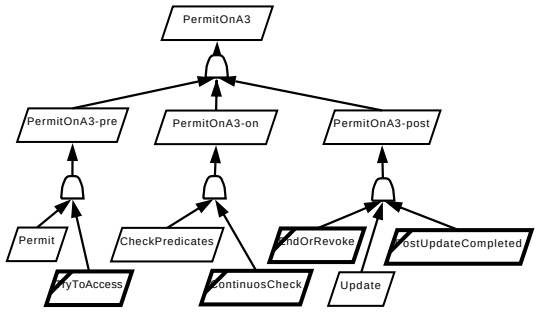
Which is easily refined in:

Goal [PermitOnA3-pre]
RefinedTo: [Permit]
[TryToAccess]
FormalDef: (∀ s:subject,
o:object,
r:right)
permitaccess(s, o, r)
⇒ • policyEnforcing(s, o, r)

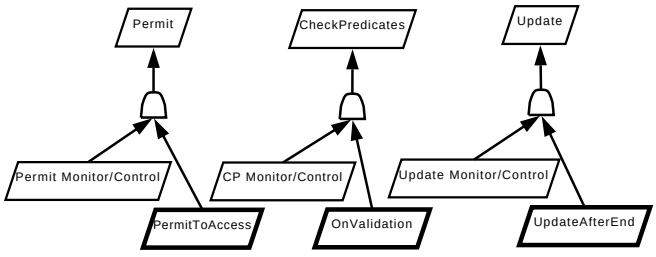
Goal [PermitOnA3-on]
RefinedTo: [CheckPredicates]
[ContinuousCheck]
FormalDef: (∀ s:subject,
o:object,
r:right)
permitaccess(s, o, r)
⇒ □ policyEnforcing(s, o, r)

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

The first and second part of the goal refinements are shown in picture 17(a) and 17(b), while the formal sub-goals' definitions follow. The refinements of the goals [PermitOnA3-pre] and [PermitOnA3-on] are not shown here, since are the same as the refinement of respectively [PermitOnA0-pre] and [PermitOnA0-post] of section 5.4.



(a) Initial goal refinement of an UCON OnA_3 core model



(b) Completion of the goal refinement of the UCON OnA_3 goal model

Figure 17: Goal and operation model for an UCON OnA_3 core model

Goal [EndOrRevoke]
Refines: [PermitOnA3-post]
FormalDef: (∀ s:subject,
o:object,
r:right)
permitaccess(s, o, r)
⇒ ◊ 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)
⇒ ◊ policyEnforcing(s, o, r)

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 requirement goals are [PermitToAccess], [OnValidation] and [UpdateAfterEnd]. The agents assigned to them are respectively the *Reference Monitor*, the *Predicate Validator* and the *Attribute Manager*.

The KAOS operation model, is the same as of picture 12. The formal specification of the KAOS operational specification for the UCON OnA_3 enforcement mechanism follows.

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]:
tryaccess(s, o, r)

Operation: PredicateValidation
Performed By: Predicate Validator
Domain Pre-Condition:
¬ RM.permitaccess(s, o, r)
Domain Post-Condition:
RM.permitaccess(s, o, r)
Input: Predicate
Output: ValidationResponse
ReqPost for [OnValidation]:
PV.validate($p_1 \wedge \ldots \wedge p_n$)

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

As usual, the only difference between these operations and its equivalent shown in section 4 is the specification of the *Required Pre-Condition* and *Required Post-Condition* clauses.

5.8 Denying and Revoking the access

A careful reader should 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 in a simple way the refinements of `DenyAccess` and `RevokeAccess`.

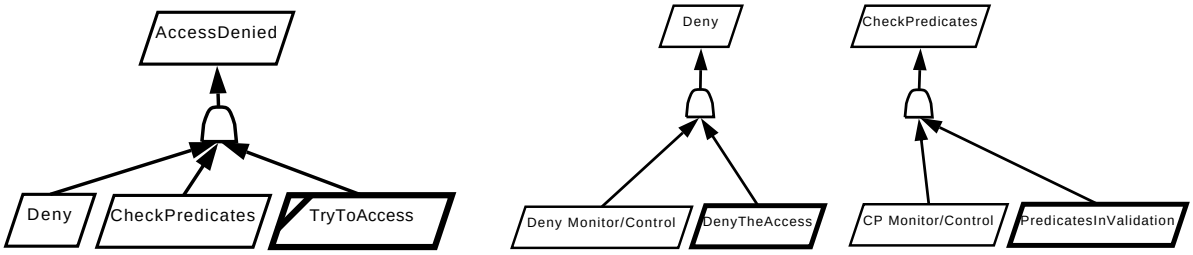
5.8.1 Denying the access

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 a *PreA* policy. The refinement shown here is valid for all the UCON *PreA* 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 picture 18(a).



(a) Initial goal refinement for the deny access goal

(b) Completion of the goal refinement for the deny access

Figure 18: Goal model for the access denied

Goal [Deny]
Refines: [AccessDenied]
RefinedTo: [Deny Monitor/Control], [DenyTheAccess]
FormalDef: (∀ s:subject, o:object, r:right)
denyaccess(s, o, r)
⇒ • ($\neg p_1 \vee \ldots \vee \neg p_n$)

Goal [CheckPredicates]
Refines: [AccessDenied]
RefinedTo: [CP Monitor/Control], [PredicatesInValidation]
FormalDef: (∀ s:subject, o:object, r:right)
($\neg p_1 \vee \ldots \vee \neg p_n$)
⇒ • 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 picture 18(b) 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.

Goal [Deny Monitor/Control]
Refines: [Deny]
FormalDef: (∀ s:subject, o:object, r:right, RM:Reference Monitor, PV:Predicate Validator)
denyaccess(s, o, r) ⇔ RM.denyaccess(s, o, r)
($\neg p_1 \vee \ldots \vee \neg p_n$) ⇔ PV.validate($\neg p_1 \vee \ldots \vee \neg p_n$)

Goal [DenyTheAccess]
Refines: [Deny]
FormalDef: (∀ s:subject, o:object, r:right, RM:Reference Monitor, PV:Predicate Validator)
RM.denyaccess(s, o, r)
⇒ • PV.validate($\neg p_1 \vee \ldots \vee \neg p_n$)
Resp: Reference Monitor

Goal [CP Monitor/Control]
Refines: [CheckPredicates]
FormalDef: (∀ s:subject, o:object, r:right, PV:Predicate Validator)
($\neg p_1 \vee \ldots \vee \neg p_n$) ⇔ PV.validate($\neg p_1 \vee \ldots \vee \neg p_n$)

Goal [PredicatesInValidation]
Refines: [CheckPredicates]
FormalDef: (∀ s:subject, o:object, r:right, PV:Predicate Validator)
PV.validate($\neg p_1 \vee \ldots \vee \neg p_n$)
⇒ • tryaccess(s, o, r)
Resp: Predicate Validator

We are now capable to derive the KAOS agent and operation models. Picture 19 the operation model, together with the agent/responsibility model. We identify a couple of operations.

Next follows the formal specification of the operations.

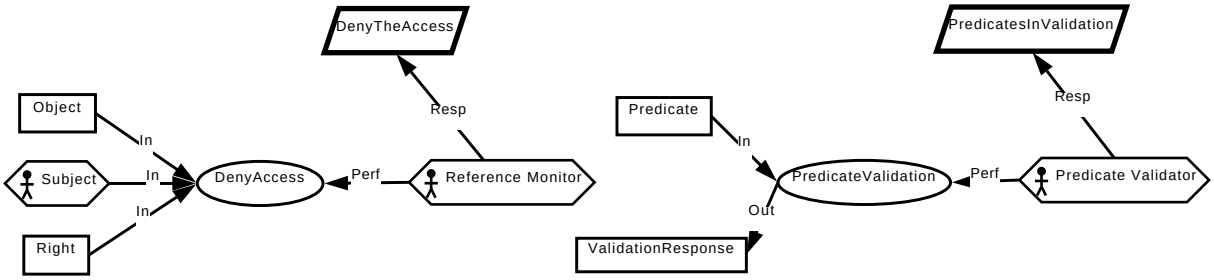


Figure 19: Operation model for an enforcing mechanism to deny an access

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 [DenyTheAccess]:
PV.validate($\neg$ $p_1 \vee \ldots \vee \neg$ p_n)

Operation: PredicateValidation
Performed By: Predicate Validator
Domain Pre-Condition:
$\neg$ PV.validate($\neg$ $p_1 \vee \ldots \vee \neg$ p_n)
Domain Post-Condition:
PV.validate($\neg$ $p_1 \vee \ldots \vee \neg$ p_n)
Input: Predicate
Output: ValidationResponse
ReqPre for [PredicatesInValidation]:
tryaccess(s, o, r)

If we don't consider the predicates to be (in)validated, there is no difference between the `DenyAccess` operation as specified here and the one shown in section 4.

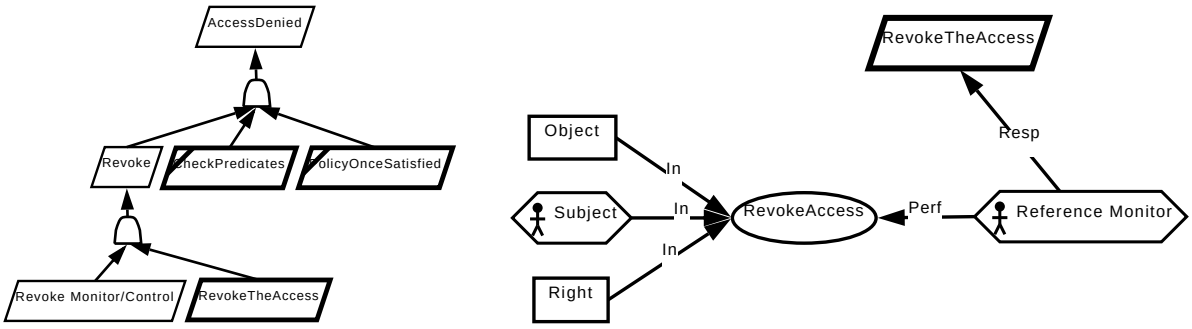
5.8.2 Revoking the access

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 *OnA* policy. The refinement shown here is valid for all the *OnA* models.

The top goal is the following:

Goal [AccessRevoked]
RefinedTo: [Revoke], [CheckPredicates], [PolicyOnceSatisfied]
FormalDef: ($\forall$ s:subject, o:object, r:right)
denyaccess(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 picture 20(a).



(a) Goal refinement for the revoke access goal

(b) Operation model for an enforcing mechanism to deny an access

Figure 20: Goal model and operation model for the access revocation

Goal [Revoke]
Refines: [AccessRevoked]
RefinedTo: [Revoke Monitor/Control], [RevokeTheAccess]
FormalDef: (∀ s:subject, o:object, r:right)
revokeaccess(s, o, r)
⇒ • (¬ p_1 ∨ . . . ∨ ¬ p_n)

Goal [CheckPredicates]
Refines: [AccessRevoked]
FormalDef: (∀ s:subject, o:object, r:right)
(¬ p_1 ∨ . . . ∨ ¬ p_n)
⇒ • On-policySatisfied(s, o, r)

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

The `On-policySatisfied(s, o, r)` predicate used in the goals [CheckPredicates] and [PolicyOnceSatisfied] is a place-holder dependent from the type of UCON OnA 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 sub-goal follows in the text. As usual, we apply accuracy and actuation goals to resolve the lack of monitorability and controllability. We identify the requirement final goal [RevokeTheAccess], and assign it the *Reference Monitor* agent.

Goal [Revoke Monitor/Control]
Refines: [Revoke]
FormalDef: (∀ s:subject, o:object, r:right, RM:Reference Monitor)
revokeaccess(s, o, r) ⇔ RM.revokeaccess(s, o, r)
(¬ p_1 ∨ . . . ∨ ¬ p_n) ⇔ PV.validate(¬ p_1 ∨ . . . ∨ ¬ p_n)

Goal [RevokeTheAccess]
Refines: [Revoke]
FormalDef: (∀ s:subject, o:object, r:right, RM:Reference Monitor)
RM.revokeaccess(s, o, r)
⇒ • PV.validate(¬ p_1 ∨ . . . ∨ ¬ p_n)
Resp: Reference Monitor

We are now capable to derive the KAOS agent and operation models. Picture 20(b) shows them, together with the agent/responsibility model. The formal specification of the operation identified follows.

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 [RevokeTheAccess]:
PV.validate(¬ p_1 ∨ . . . ∨ ¬ p_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 4.

6 Related Work

The work reported in sections from 4 to 5 is associated to two strands of related work: 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 [3]. 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 [22], 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 [24], 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 [11] 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 [31] and [15] we have reviewed in section 3.1.

7 Conclusion and Future Work

This paper has presented 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$_a$ usage control policies. We have concentrated on the UCON model proposed by Park and Sandhu [19] and studied its application for the case of Data Grid Management Systems (DGMS). Our approach consists in applying the KAOS requirements-engineering methodology [27] to the design of the enforcement mechanism. The starting point is the definition of an abstract specification of the enforcement mechanism. Then, we applied KAOS to each of the UCON$_a$ sub-models to prove that the specification is correct. The UCON$_a$ policies can be refined into concrete ones — which could be enforced by the resulting system — by applying KAOS goal refinement. KAOS offers a formal language to represent the goal based on temporal logic, close to the formal language used to give semantic to UCON models. The refinement method also includes strategies and patterns to guide the refinement process, and there is tool support aiding the user in this process.

The low-level policies will be assigned to software agents responsible of executing the operations that enforce the concrete policies. Our resulting architecture consists of three agents: an *Attribute Manager*, responsible of updating attributes associated to subjects and objects; a *Predicate Validator*, responsible of validating policy predicates; and a *Reference Monitor*, acting as a gateway for all the usage decisions of the DGMS.

Since different UCON models encode a different sequentiality of the operations, we showed that the derived operations always encode the same state-transitions as specified by those of our specification, but since the sequentiality of the single operations is different a model from each other, the *Required Pre-, Post-* and *Trigger* Conditions are 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$_a$ sub-model. This technique is very similar to the one used in [3]. A possibility for the encoding of such *strategy* directly in the policy is the use the POLPA language.

Our future works will follow two main strands: first of all we'll use KAOS to formally analyse the requirements for an enforcement mechanism of a complete UCON$_{abc}$ model, thus including not only Authorizations (UCON$_a$), but also oBligations (UCON$_b$) and Conditions (UCON$_c$). Then, we'll work on the definition of an architecture with the final objective to either propose a prototype, or extensions to the already developed implementations. We'll review the already deployed policy languages and analyse their capacity to encode UCON policies, keeping in consideration the OGSA recommendation on the use of standards. Finally, we'll keep analysing ways to control the policy granularity by using Semantic Grid techologies, as well as issues regarding the evaluation of cuncurrent policies. Other usage control techniques, like Pretschner's [21], will be analysed to discover potential benefits.

References

[1] 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.

[2] 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.

[3] 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.

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

[5] 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.

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

[7] 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.

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

[9] 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.

[10] Ian Foster, Carl Kesselman, Gene Tsudik, and Steven Tuecke. A security architecture for computational grids. In *CCS '98: Proceedings of the 5th ACM conference on Computer and communications security*, pages 83–92, New York, NY, USA, 1998. ACM.

[11] 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.

[12] 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.

[13] E. Letier and A. van Lamsweerde. Agent-based tactics for goal-oriented requirements elaboration. 2001.

[14] 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.

[15] 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.

[16] Jonathan D. Moffett and Morris S. Sloman. Policy Hierarchies for Distributed System Management. *IEEE JSAC Special Issue on Network Management*, 11(9), 11 1993.

[17] 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.

[18] OASIS. Oasis extensible access control markup language (xacml) tc. http://www.oasis-open.org/committees/xacml, 2005.

[19] 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.

[20] 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.

[21] Alexander Pretschner, Manuel Hilty, and David Basin. Distributed Usage Control. *Communications of the ACM*, September 2006.

[22] 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.

[23] Ravi S. Sandhu and Jaehong Park. Usage Control: A Vision for Next Generation Access Control. In *MMM-ACNS*, pages 17–31, 2003.

[24] 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.

[25] 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.

[26] 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), June 2004.

[27] A. van Lamsweerde. Requirements Engineering in the Year 00: A Research Perspective. In *International Conference on Software Engineering*, pages 5–19, 2000.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.