Evolving Access Control: Formal Models and Analysis

Clint V. Sieunarine

Linacre College

A thesis submitted in partial fulfilment of the requirements for the degree of

Doctor of Philosophy

Hilary Term 2011

University of Oxford

Evolving Access Control: Formal Models and Analysis

Clint V. Sieunarine
Linacre College

A thesis submitted in partial fulfilment of the requirements for the degree of

Doctor of Philosophy

Hilary Term 2011

University of Oxford

Abstract

Any model of access control has two fundamental aims: to ensure that resources are protected from inappropriate access and to ensure that access by authorised users is appropriate. Traditionally, approaches to access control have fallen into one of two categories: discretionary access control (DAC) or mandatory access control (MAC). More recently, role-based access control (RBAC) has offered the potential for a more manageable and flexible alternative. Typically, though, whichever model is adopted, any changes in the access control policy will have to be brought about via the intervention of a trusted administrator. In an ever-more connected world, with a drive towards autonomic computing, it is inevitable that a need for systems that support automatic policy updates in response to changes in the environment or user actions will emerge. Indeed, data management guidelines and legislation are often written at such a high level of abstraction that there is almost an implicit assumption that policies should react to contextual changes. Furthermore, as access control policies become more complicated, there is a clear need to express and reason about such entities at a higher level of abstraction for any meaningful analysis to be tractable, especially when consideration of complex state is involved. This thesis describes research conducted in formalising an approach to access control, termed evolving access control (EAC), that can support the automatic evolution of policies based on observed changes in the environment as dictated by high-level requirements embodied in a *metapolicy*. The contribution of this research is a formal, conceptual model of EAC which supports the construction, analysis and deployment of metapolicies and policies. The formal EAC model provides a framework to construct and describe metapolicies and to reason about how they manage the evolution of policies. Additionally, the model is used to analyse metapolicies for desirable properties, and to verify that policies adhere to the high-level requirements of the metapolicy. Furthermore, the model also allows the translation of verified policies to machine-readable representations, which can then be deployed in a system that supports fine-grained, dynamic access control.

Declaration

Except where otherwise stated in the text, this thesis is the result of my own work and is not the outcome of work done in collaboration. This thesis is not substantially the same as any that I have submitted or am currently submitting for a degree, diploma or any other qualification at any other university. No part of this thesis has already been or is being concurrently submitted for any such degree, diploma or any other qualification.

Acknowledgements

I would like to express my sincere gratitude to Dr. Andrew Simpson, my supervisor; he has provided invaluable advice and encouragement throughout the development of this thesis.

I am also indebted to Dr. Jeremy Bryans, Dr. Ivan Flechais, Dr. Steve McKeever, Dr. David Power and Dr. Mark Slaymaker for their comments, feedback and guidance.

Additionally, I would like to acknowledge colleagues, students, staff and lecturers associated with the Software Engineering Programme for making my daily routine so much more enjoyable.

Furthermore, I must thank my friends and relatives, especially the few still here in Oxford and London, and those now situated around the world: Boston, Düsseldorf, New York City, Melbourne, Minneapolis, Montreal, San Francisco, Toronto and last, but certainly not least, Trinidad. They have all provided varying levels of joy, relief and excitement during the long four years of this degree.

Finally, to my parents, Jewan and Neresha Sieunarine, my brother, Jessel Sieunarine, and my sister, Dr. Nailah Sieunarine—your love and support have driven me to excel far beyond my own expectations, thank you.

The work described was funded by a Teaching Assistantship from the Software Engineering Programme, headed by Professor Jim Davies, at the Department of Computer Science, University of Oxford.

Overall, I am eternally grateful for this consequence of serendipity.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Rapid advances in technology have created an escalating dependency on data. Massive amounts of data need to be shared, stored, analysed or processed, and arise from many diverse sources such as online transaction logs, telescope imagery, medical records, web pages and music files. Crucially, a significant percentage of this data involves highly sensitive and personal information, which is used by government, healthcare, academic, financial and commercial organisations. As the importance of data management continues to increase in these contexts, so does the need for assurance that data access and sharing is *appropriate*; often, of course, this requirement is driven by national or international legislation. For example, the United Kingdom Data Protection Act [4] requires that the appropriate technical and organisational measures are taken against unauthorised or unlawful processing of personal data and against accidental loss or destruction of, or damage to, such data. Similarly, the United States Sarbanes-Oxley Act [5] mandates that management document and evaluate the effectiveness of its internal technological controls, as well as create and maintain satisfactory procedures for financial reporting.

As data access requirements become more complex as a result of such legislation, standards and guidelines, it will become increasingly important to develop tools and technologies to support the secure sharing of data, specifically, to ensure that low-level implementations of access control policies are consistent with the high-level requirements that prescribe boundaries of permissible behaviour. Certainly, there will be a need to bridge the different levels of abstractions at which policies are motivated, described, implemented and enforced as identified in [6, 7, 8, 9, 10]. Ideally, the access control systems that protect these shared data sources would have the potential to guarantee that access control policies which have been deployed are, in some sense, "correct"; in reality, this is often not the case.

Our concern in this thesis is the provision of what might be termed *evolving access control* (EAC), in which what is permissible for users and applications by way of access to data may change dynamically, depending on context. Our notion of context-sensitivity is strictly broader than that of, for example, the context-sensitive role-based access control of [11], in which roles and permissions are resolved in terms of the context—we concern ourselves also with actions and information drawn from the environment in which the system

is deployed. In light of this, EAC is so-called because access control policies, at a low level, *automatically* evolve on the basis of observed actions guided by requirements, at a high level, captured in a *metapolicy*—there is no need for intervention by, for example, database administrators or applications in order to bring about a change. Metapolicies prescribe the relationship between policies: after Officer X has been out of contact for half an hour, any access that was previously possible is now denied; once Professor Y has accessed 10 images, she can access no more of them; when the network capacity increases, Doctor Z can access data once again. Thus, these metapolicies are necessarily written at a higher level of abstraction—and, as such, are intended to be closer to the level at which requirements might be captured—than policies.

Of course, providing a system with the functionality to adapt access control policies automatically—without human intervention—means that the need for assurance that the *correct* policy is in place necessarily increases: ensuring that certain fundamental properties hold in every potential state, for example, is essential: we would not want our protection mechanism to evolve into a state that provided little security, for example. Moreover, access control requirements are emerging that demand assurance that such sharing of data is private, legal and ethical—influenced by, for example, the aforementioned Data Protection Act and Sarbanes-Oxley Act.

Hence the driver for a *model-driven* approach to evolving access control: raising the level of abstraction for data owners and policy writers, with a view to giving some assurance that data sharing is appropriate, will be key to the long-term success of our work. Our focus in this thesis, however, is the development of a formal model that investigates the behaviour of metapolicies and the evolution of policies in support of our long-term vision.

1.2 Contribution

Access control policies can be conceptualised as documents which prescribe who can access what within a protected system. Our approach relies on the notion that at any moment in time, a single access control policy has been deployed in such a system. Any changes in the system context automatically triggers an evolution of this policy to reflect the guidelines captured in a metapolicy.

Needless to say, that current policy must be correct. Incorrect policies lead to security breaches as unauthorised individuals can access private data. Incorrect policies also lead to system malfunctions as authorised users cannot access requisite data, especially in cases of emergency. Furthermore, in discretionary contexts, permissions may grow and shrink, making it unclear which actions are permitted at any given time. It also becomes difficult to ascertain if these policies still adhere to necessary security requirements. As a result, proving correctness of access control policies is difficult. Moreover, there are limits to the proof of such correctness: the *safety problem* of [12], defined as the possibility that an access control policy can reach a state in which any subject can obtain a certain right to access an object, is undecidable.

Nevertheless, while *proof* is not attainable in our sphere of work, properties of access control models can still be checked and verified (as done in, for example, [13, 14, 15, 16]), providing greater *assurance* that such models function correctly. With this in mind, we therefore seek to answer the following research question.

> *Can we develop a formal model for the construction, analysis, and automatic deployment of EAC metapolicies and policies, which provides assurance that the sharing of sensitive data is appropriate?*

The work presented in this thesis aims to provide assurance to policy writers and data source owners that the low-level implementations of access control policies correctly meet the high-level requirements embodied in an EAC metapolicy. The need to close this separation between requirements and implementations is not a recent issue, and, indeed, several efforts have been dedicated to this problem, most notably those of [6, 7, 8, 9, 10]. Our approach, however, is novel as it couples the ability to capture high-level requirements in metapolicies, and to record events in an audit data source, which, in tandem, enable low-level policies to automatically evolve on the basis of observations. Ultimately, we want to establish a stable platform for the development of tools that can construct, analyse and deploy metapolicies and its associated policies, and, hence, ensure that the sharing of sensitive data is appropriate. Thus, the contribution of this thesis is a formal model of EAC that forms the abstract framework underpinning the implementation of such tools, and provides a three-pronged end-to-end process that aids in answering the research question.

First, the formal model captures our EAC approach and allows the construction of metapolicies and algorithms that drive the evolution of policies. We use a specification language (based on set theory and predicate logic) to formalise this abstraction so that we can reason about metapolicies, policies and their behaviour. The use of formal methods to logically represent access control systems is certainly not new (see, for example, [17, 18, 19]), and has the potential to reduce design errors by using techniques and tools for specifying, analysing and verifying such systems. According to Landwehr in [20], not only are formal security models necessary for the design of reliable systems but also to convince others that such systems are indeed secure. While the use of formal methods does not guarantee correctness [21], it emphasises greater understanding of our abstraction by exposing incompleteness, ambiguities, and inconsistencies that might otherwise go undetected [22].

Second, the formal model supports the analysis and verification of EAC metapolicies and policies based on certain properties and requirements. In EAC, metapolicies capture the relationship between access control policies as they evolve during the lifetime of the system. Traditionally, access control models have typically been classified as either discretionary or mandatory [23], but the emergence of role-based access control (RBAC) [24] has provided a more recent alternative which works at a higher level of abstraction. While it is important to demonstrate that policies captured in terms of these models are correct and perform as intended, it is even more critical in the case of our dynamic approach as policies can change automatically without the manual intervention of system administrators. Such autonomy gives rise to a level of complexity that can introduce subtle errors, some of which

might eventually lead to compromises (or a failure to adhere to security requirements). Accordingly, we have identified key properties that metapolicies should possess which provide some protection against such problems. Analysis thus involves an investigation of these main properties by ensuring that the automatic evolution of policies is correct, and by verifying that evolutions do not result in corrupted or unexpected policies that can potentially violate requirements.

Finally, our model provides a means of transforming our formal representations into machine-readable ones. A key aim of software engineering is to empower developers to create systems that function reliably from design to implementation despite vast complexity. Thus, having constructed and analysed metapolicies and policies using our EAC abstraction, we aim to validate our approach by deploying verified metapolicies and policies in a real system.

1.3 Structure of the Thesis

The remainder of the thesis is organised as follows. Chapter 2 explores the background and context for this thesis in greater detail, including an overview of current access control models and concepts, a description of immediate and related work, a high-level explanation of EAC, and a presentation of our research methodology. Chapter 3 then investigates the nature of metapolicies, explains the advantages of our approach, outlines classes of data source requirements that can be captured by metapolicies, and introduces four small models that drive development of our EAC abstraction. Chapter 4 presents our contribution: an EAC abstraction. In this chapter, we formally describe and construct several structures and algorithms that support the automatic evolution of policies. Then, we propose how we may analyse and verify key properties that metapolicies and policies should possess. Next, we suggest how we can convert our version of policies to machine-readable versions so that they can be deployed in real systems. This chapter concludes with a small model example illustrating the use of the abstraction. Chapter 5 then presents a case study based on a real-world scenario as a more practical means of validating the work of Chapter 4. Finally, in Chapter 6, we summarise our contribution, comment on the overall research process and tools, and outline potential areas of future work.

2

Foundations and Context

2.1 Introduction

From gaming networks to government ministries to global corporations, loss of electronic data occurs frequently around the world. Numerous accounts of such data loss have directly motivated the need for middleware that can facilitate the secure sharing of large quantities of private data, with the intention that security breaches are mitigated while allowing appropriate access to data.

Automated systems have thus been developed to deal with secure sharing and aggregation of data. In such systems, data can be accessed at several locations, once the appropriate permissions are in place—as several data sources are networked to communicate with each other. As an example, the Generic Infrastructure for Medical Informatics (GIMI) project team [25, 26] developed the service-oriented interoperability framework (sif) [27], which facilitates the secure aggregation of data from disparate, heterogeneous data sources. The access control policies of a sif deployment must adhere to strict guidelines as to who can access what. The access control subsystem should never attain a state which allows unauthorised access to resources. But how can one ensure that the access control mechanism works as intended? Moreover, how can we provide assurance that the sharing of sensitive data meets certain security requirements? This is where formal methods can play a significant role—one of the primary goals behind our research is to develop a formal abstraction of context-sensitive access control, which involves the construction of metapolicies and policies in a suitable executable language that can facilitate its analysis. While our research has been motivated by case studies from GIMI and validated within this context, the contribution itself is, to a large extent, generic, and, as such, more broadly applicable. Certainly, the classes of applications that may be supported are not limited to healthcare; further with appropriate transformations in place, it should be possible to translate metapolicies and policies into other formal languages, such as the Security Policy Assertion Language (SecPAL) [28], or the Enterprise Privacy Authorization Language (EPAL).[1]

This chapter provides the reader with the necessary background information that places our work in context under the broad field of secure data management. We begin with

[1] http://www.zurich.ibm.com/security/enterprise-privacy/epal/Specification/

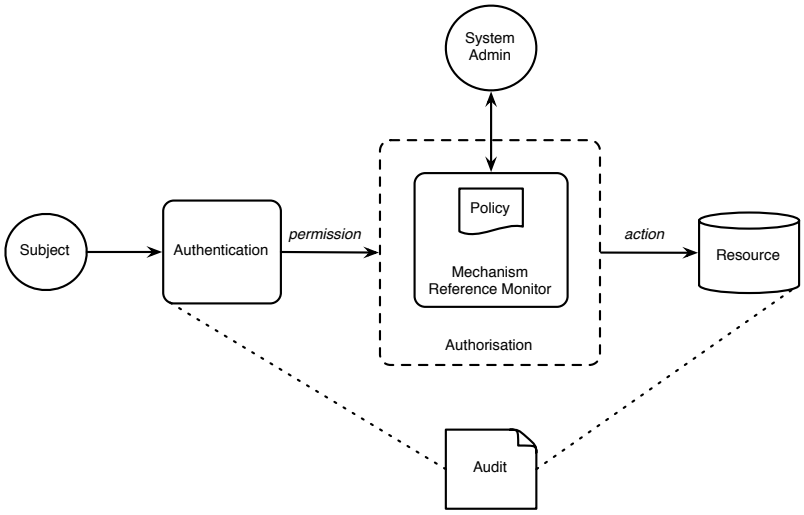


Figure 2.1: A typical access control system

a brief review of fundamental access control concepts. We then present the immediate context of our research: the work undertaken at the University of Oxford, which involves the development of middleware and mechanisms that create a deployment platform and testbed for our research. The relevance of this thesis to the wider field of access control is then introduced by means of a literature survey grouped into four categories. We conclude this chapter with a description of our research methodology.

2.2 Access Control

Secure data management revolves around the notion of three requirements: any information system must protect data against unauthorised disclosure (*secrecy*) and unauthorised modifications (*integrity*), while ensuring availability of such data to authorised users (*availability*) [1]. Access control regulates all accesses to an information system by ensuring that only authorised accesses can be executed. In this section, we present only a subset of the main access control concepts that will be referenced throughout this thesis.

A *resource*, also known as an *object*, is the target of an access request and contains data. Examples of such resources include files in an operating system, segments or pages in shared memory, and values in a database record. A *subject* often equates to a user and generally in the form of a person, process, or device. This entity asks the access control system for permission to access a resource. An *action* is an operation which occurs once a subject has been granted access to a resource. Typical actions include read, write or execute operations. A *permission* or *authorisation* is a tuple which embodies a subject, resource and action, and defines which subjects can perform which actions on which resources.

An access control system or framework protects system resources against inappropriate or undesired subject access. Operating systems use access control to protect files and directories. Database systems utilise access control to manage access to the data in tables and views. Such systems typically comprise of *authentication*, *authorisation* and *audit* com-

	File A	File B	File C
Alice	Own Read Write	Write	Read Write
Bob	Read Write	Own Read Write	Own Read Write

Figure 2.2: An access matrix

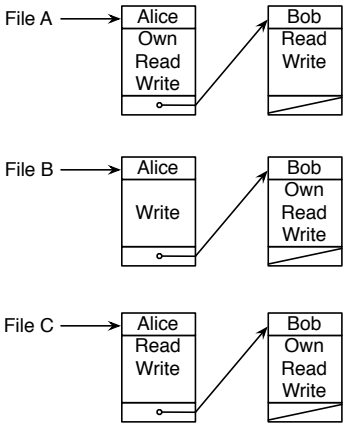


Figure 2.3: Access control lists

ponents [29] as shown in Figure 2.1. Authentication deals with verifying the identity of a subject [30]. Several methods of authentication are in use today, and they include passwords and personal identification numbers (PIN), smart cards and even biometric properties such as fingerprints and retina scans. Audit relies on using logs and audit trails to hold a subject accountable for its actions on a system [31]. Audit records can be used to detect security violations or replicate security incidents. Typically, only system administrators should have access to audit data. Authorisation, on the other hand, manages what a subject can do and access within the context of the system provided that the subject has been authenticated [23].

Access control systems span three levels of abstraction that support its development: *policies*, *models* and *mechanisms* [1]. Policies are high-level rules that dictate how access control should be regulated. Models are formal representations of these policies, which support the proof of security properties of an access control system. Mechanisms define the software and hardware that enforce the rules imposed by the policy and formally specified in the model.

2.2.1 Discretionary Access Control (DAC)

DAC is one of three main classes of access control policies. Such policies are authorisation-based where rules explicitly state which resources can or can not be accessed by subjects. The ability to grant, or *delegate*, an access right to another user is the discretionary part of the control. Early DAC policies were based on the access matrix model.

The access matrix is an abstract, conceptual model that describes the actions that a subject can perform on a resource as formalised by the Harrison, Ruzzo and Ullman (HRU) model in [12]. Each row in the matrix represents the rights for a subject, whereas each column is concerned with a particular resource, as shown in Figure 2.2. Therefore, each cell of the matrix designates the access authorised for that subject (in the row) to that resource (in the column) [23]. This matrix embodies the aim of access control—to ensure that only these subjects can perform actions on those resources as specified in the respective matrix cell.

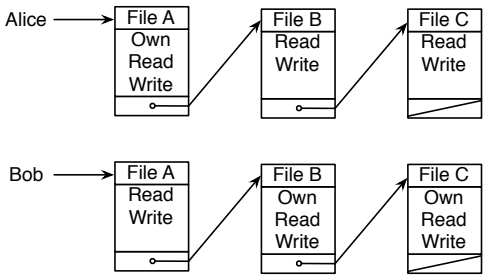


Figure 2.4: Capability lists

Subject	Access Mode	Resource
Alice	Own	File A
Alice	Read	File A
Alice	Write	File A
Alice	Write	File B
Alice	Read	File C
Alice	Write	File C
Bob	Read	File A
Bob	Write	File A
Bob	Own	File B
Bob	Read	File B
Bob	Write	File B
Bob	Own	File C
Bob	Read	File C
Bob	Write	File C

Figure 2.5: Authorisation relations

In real systems, of course, the access matrix is hardly implemented as a matrix because of its sheer size. There are three practical solutions to implementing the access matrix. The first is *access control lists* (ACL), which are linked lists for every resource, indicating which subjects are allowed access (these are the matrix columns), as shown in Figure 2.3. Modern operating systems typically utilise this ACL-based approach. For example, on the Windows operating system, if Alice creates a resource, file F, she becomes the owner of F. She can then modify this permission to grant access to others. File F can now be associated with a list of users and groups to which owner Alice has granted some level of access rights. In traditional Unix-based systems, the access control architecture of users, groups, and read-write-execute permissions is also a form of this ACL approach.

In Figure 2.4, the second solution is *capability lists*, which are similar to access control lists in that they are also linked lists but relate each subject to all resources that can be accessed by that subject (these are the matrix rows).

The final solution is *authorisation relations* as represented by the table in Figure 2.5, which are all the possible (subject, action, resource) tuples of the access matrix in a table format. Relational database management systems typically use these representations.

DAC policies have two main disadvantages. The first is that DAC policies do not differentiate between users and subjects. Users are entities that can log into a system with suitable authorisations to act on resources in that system. Users can then initiate processes (which now become subjects) that act on behalf of that user. Since DAC ignores this distinction between users and subjects, they become susceptible to Trojan Horses. The second disadvantage of DAC policies is their inability to control the dissemination of information—once information has been accessed by a process, such processes can leak information to unauthorised users [1].

In the next section, we see how the next main class of policies addresses these issues.

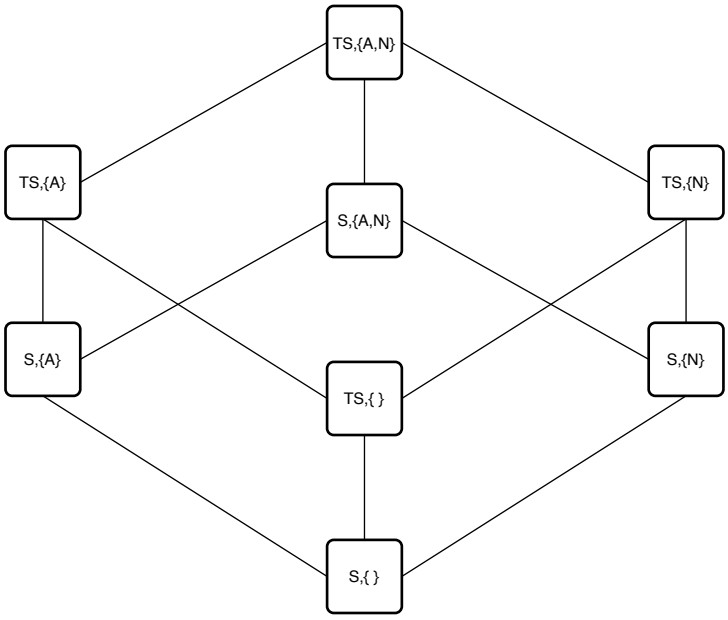


Figure 2.6: A security lattice [1]

2.2.2 Mandatory Access Control (MAC)

MAC policies are prescribed by a central authority instead of the individual owner of the resource (who also lacks the permissions to modify access rights) [23]. One of the most standard forms of MAC policies are multi-level security policies, which limit the dissemination of information as well as identify a difference between users (as human beings) and subjects (as processes) unlike DAC policies.

Subjects and resources of multi-level security policies are assigned security *designations* (or clearances), which form a partially ordered set (poset) with an inherent dominance relationship. Each designation is comprised of two entities: the first is a *classification* such as Top Secret (TS), Secret (S), Confidential (C), and Unclassified (U), such that TS > S > C > U. The other is a *category set* such as Nuclear (N), Army (A), NATO (T) and so on, which forms an unordered set. A complete security designation is thus a (classification, category) pair. A subject can only access a resource if his or her security designation dominates the security designation of that resource. For example, one designation, (classification A, category 1), dominates the other, (classification B, category 2), if and only if classification A is higher than or equal to classification B, and category 1 includes category 2 as a subset [32]. Such designations, together with the dominance relationship, form a lattice [33] as shown in Figure 2.6—we see that designation (TS,{A,N}) dominates designation (S,{N}).

The dominance of designations can differ based on the goals of the MAC policy. Secrecy MAC policies regulate the flow of information, and originate from the model that was first described by Bell and La Padula in [34]. The Bell-La Padula model embodies two major principles: “no-read-up”, meaning that resources can be read only by subjects with clearance greater than or equal to the resource’s classification, and “no-write-down” meaning that resources can be written to only by subjects with clearance less than or equal to that of

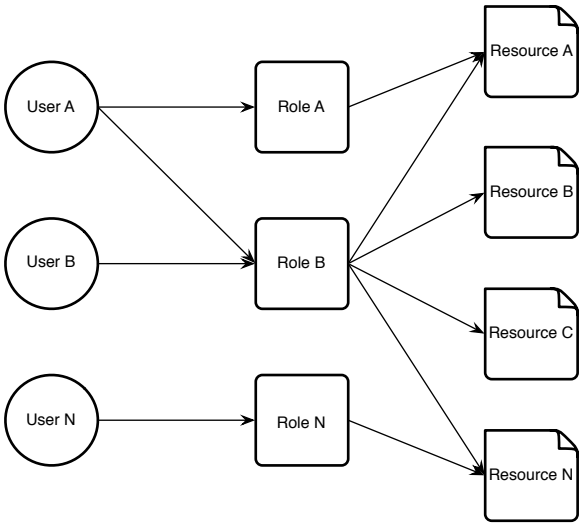


Figure 2.7: Role-based access control [1]

the resource's classification. Conversely, integrity MAC policies, based on the Biba model [35], enforce the integrity of information and prevent indirect modifications of resources by employing the use of a "no-read-down" (a subject with lower clearance than a resource can have read access to that resource) and "no-write-up" (a subject with higher clearance than a resource can have write access to that resource) convention.

Even though MAC policies address the issues of DAC policies, they still suffer from the shortcomings of its own design in that it becomes too rigid to be used commercially—consequently, MAC policies are typically used in hierarchical, military-like organisations. Additionally, even though MAC policies provide assurance that the flow of information is appropriate, covert channels that are not intended for normal communication can still be misused to infer information.

The strength of MAC and the flexibility of DAC can be combined to offer a single policy, where a DAC policy delivers a finer granularity of control within the general boundaries prescribed by the MAC policy [36]. Examples of this type of access control exist, and include the Brewer-Nash model [37] (also known as the Chinese Wall Security Policy), which was developed to reduce conflict of interest in commercial organisations especially insider trading in financial institutions. The Usage Control Model (UCON) [38, 39], commonly used for digital rights management (DRM), and most recently, the Times-Based Usage Control Model (TUCON) [40] are also instances of this type of enhanced policy.

2.2.3 Role-Based Access Control (RBAC)

RBAC is the third main class of access control policies and is an association of access rights with the roles that individuals may possess within an organisation [24]. As shown in Figure 2.7, these roles can then be assigned permissions to access resources. For example, system administrators, such as User A, are hired to ensure proper deployment and continuous maintenance of computer systems. Sometimes, according to the size of the organisation,

several system administrators can be present, such as Users A and B. These individuals will assume the role "system administrator"—Role B, which will normally allow them full access rights according to the RBAC policy. Similarly, there will be roles for "employees", "managers" and "supervisors", each with their own sets of permissions. We therefore see that RBAC, at its most basic level, concerns itself with two relations: one between users and roles, and another between roles and permissions.

RBAC offers several advantages. The management of authorisation is simplified, especially when a new employee joins or leaves an organisation—a role can be easily assigned to that individual or removed. The use of roles in this manner also enforces the principle of least privilege: minimal privileges allow a user to perform some task that mitigates the dangers of unintended errors. Furthermore, RBAC supports separation of duties which is the principle that no user can be assigned a role that enables him or her to execute a task that can be deemed as malicious; instead roles are decomposed, for example, the individual authorising a paycheque should not be the same individual who prepares them [1].

Although RBAC increases manageability, it lacks expressiveness—Tripunitara and Li in [41] were able to show that RBAC is limited in its expressive capabilities when compared to a version of DAC. Bacon et al. have also concluded that the Organization for the Advancement of Structured Information Standards (OASIS) model of RBAC is insufficient in practice to meet the needs of applications such as electronic health records (EHR) because of its constrained expressiveness [42]. Examples of systems employing some form of RBAC include Microsoft Active Directory, Microsoft SQL Server, and Oracle DBMS.

2.2.4 Extensible Access Control Markup Language (XACML)

XACML is a general-purpose language for implementing access control policies [43]. Based entirely on eXtensible Markup Language (XML),[2] XACML is fully extendable and provides sufficient features that allow it to describe two languages for an application. The first language, called the policy language, allows one to define general access control requirements. The second language, known as the request/response language, is used to formulate queries. As a simple use case, an access control mechanism configured to use XACML accepts a user request for some resource—this request is specified in the request/response language of XACML. The access control mechanism then consults an XACML policy, written in the policy language, and evaluates the request. The mechanism returns a response indicating if that user is permitted or denied access to that resource.

XACML allows significant search and storage abilities by using attributes (or tags) to categorise content as it draws from the extensive attribute management approach of XML. As an example, we provide a brief description of an XACML policy showing the relevant attributes (details of these attributes are explained in Chapter 7). The XACML policy listed overleaf is taken from the Programmer's Guide to Sun's XACML Implementation,[3] which allows all logins to a sample server from 9 a.m. to 5 p.m.

[2] http://www.w3.org/XML/

[3] http://sunxacml.sourceforge.net/guide.html

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <Policy xmlns="urn:oasis:names:tc:xacml:1.0:policy"
3    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4    xsi:schemaLocation="urn:oasis:names:tc:xacml:1.0:policy
5    cs-xacml-schema-policy-01.xsd"
6    PolicyId="SamplePolicy"
7    RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:first-applicable">
8    <Target>
9      <Subjects>
10       <AnySubject/>
11     </Subjects>
12     <Resources>
13       <Resource>
14         <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
15           <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">S-Server</AttributeValue>
16           <ResourceAttributeDesignator
17             DataType="http://www.w3.org/2001/XMLSchema#string"
18             AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
19         </ResourceMatch>
20       </Resource>
21     </Resources>
22     <Actions>
23       <AnyAction/>
24     </Actions>
25   </Target>
26   <Rule RuleId="LoginRule" Effect="Permit">
27     <Target>
28       <Subjects>
29         <AnySubject/>
30       </Subjects>
31       <Resources>
32         <AnyResource/>
33       </Resources>
34       <Actions>
35         <Action>
36           <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
37             <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">login</AttributeValue>
38             <ActionAttributeDesignator
39               DataType="http://www.w3.org/2001/XMLSchema#string"
40               AttributeId="ServerAction"/>
41           </ActionMatch>
42         </Action>
43       </Actions>
44     </Target>
45     <Condition FunctionId="urn:oasis:names:tc:xacml:1.0:function:and">
46       <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:time-greater-than-or-equal">
47         <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:time-one-and-only">
48           <EnvironmentAttributeDesignator
49             DataType="http://www.w3.org/2001/XMLSchema#time"
50             AttributeId="urn:oasis:names:tc:xacml:1.0:environment:current-time"/>
51         </Apply>
52         <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#time">09:00:00</AttributeValue>
53       </Apply>
54       <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:time-less-than-or-equal">
55         <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:time-one-and-only">
56           <EnvironmentAttributeDesignator
57             DataType="http://www.w3.org/2001/XMLSchema#time"
58             AttributeId="urn:oasis:names:tc:xacml:1.0:environment:current-time"/>
59         </Apply>
60         <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#time">17:00:00</AttributeValue>
61       </Apply>
62     </Condition>
63   </Rule>
64   <Rule RuleId="FinalRule" Effect="Deny"/>
65 </Policy>
```

In lines 1 to 7, the policy header declares a unique policy identifier and the type of rule-combining algorithm. The target component for this policy acts as a filter in lines 8 to 25, allowing all incoming requests by any subject to perform any action but only on resource "S-Server" (line 15). Rules are at the heart of XACML policies—they return a permit or deny response to the request. The first rule (line 26) is only used if the incoming request would like to perform an action of "login" (line 37), and returns a permit response subject to its condition component. We note that the condition component for this rule, in lines 45 to 62, further restricts the ability to execute this action between the hours of 9 a.m. (line 52)

and 5 p.m. (line 60). A second default rule is also declared on line 64, which always returns a deny response in the case that the first rule does not apply. On line 7, we observe that the rule-combining algorithm employed is the "first-applicable"—rules must be evaluated in the order that they are listed, and the first rule that applies to the request is the decision of the policy. There are several types of rule and policy-combining algorithms that handle the case of multiple rules or policies returning a response. The aim of XACML policies, such as this, is to return a single authorisation decision for any request.

It is interesting to note that because XACML offers a powerful framework for expressing policies, even relatively simple policies such as this can get unwieldy, which inevitably increases the chances that mistakes are made during creation. Moreover, we note that these policies are static and require manual intervention for an update, which becomes tedious and further prone to error when dealing with constantly changing requirements.

2.3 Immediate Research

In this section, we describe how the work in this thesis is related to immediate research at the University of Oxford. The Generic Infrastructure for Medical Informatics (GIMI) project was concerned with the development of a generic, reliable middleware layer able to facilitate the sharing and aggregation of data from heterogeneous data sources. The crucial driver behind GIMI was ensuring legal and ethical access to such sensitive data using a mechanism which offers fine-grained and dynamic access control to resources [25, 26, 44]. GIMI was a collaborative project funded by the Technology Strategy Board.[4]

The goals of the project gave rise to two distinct but complementary technologies. While the former is aimed at researchers, application developers and domain specialists, the latter is aimed at data owners, although both can work together:

- sif (service-oriented interoperability framework) [45] and
- evolving access control.

2.3.1 sif

In [27], the sif framework is described as a "dependable middleware layer capable of supporting data sharing across disparate sources via fine-grained access control mechanisms". At a high level of abstraction, Figure 2.8 illustrates how sif operates. Data is accessed via an internal interface, *I*. The local policy, *P*, at any site or node regulates the access to the data source. The external interface, *E*, is a web service which interacts with other nodes. This design allows each node to maintain control of its access control policies and data [45]. The access control mechanism of sif uses XACML for fine-grained access control.

[4] http://www.innovateuk.org

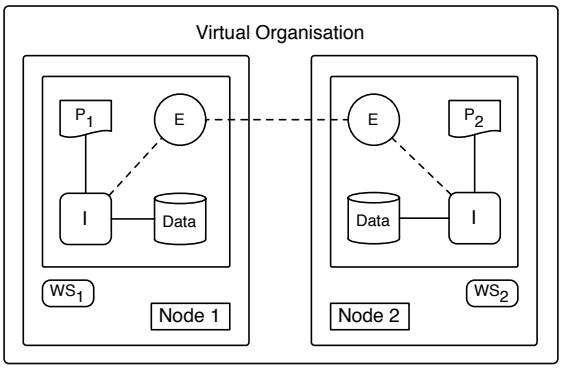


Figure 2.8: The sif view of a virtual organisation [2]

The motivation for sif is described in [2], in which several user scenarios pertaining to information security requirements are presented. A short overview of each scenario (given in terms of a healthcare context) is reproduced below for the sake of convenience.

- *Distributed queries of patient data.* An authorised user wishes to query data that would span across several hospital data sources. Each hospital decides its own policy for data access. The user would receive the confederated results containing only that data that he or she is permitted to access.
- *Working at a remote site.* If a doctor is working at a remote hospital then that doctor should be able to access data from his or her local hospital, even though the request may be subjected to a policy which differs from the one used at the local hospital.
- *Delegation of access permissions.* A senior health professional can grant access to certain data to a colleague provided that this access is temporary, and be granted to either a named individual or a group.
- *External access.* The hospital would use a different policy for external access as these are requests coming from outside the present virtual organisation. Such requests would include either a health professional working from home or even an individual patient wishing to view his or her own records.
- *Modification of data.* Each hospital is responsible for the data that is stored and who accesses what and as such, it should keep a record of all modifications made. For example, a doctor wants to modify the data stored on a patient and will only be able to do so if that hospital's policy allows it.
- *Transferring patient records.* If a patient relocates and is now being treated at another hospital, then this will involve modification of data as ownership has been changed. A distributed query may also be involved as data may already be present at other hospitals.

The sif architecture shown in Figure 2.9 has the ability to support mainly two types of users—application developers and data owners. Furthermore, the sif plug-in architecture

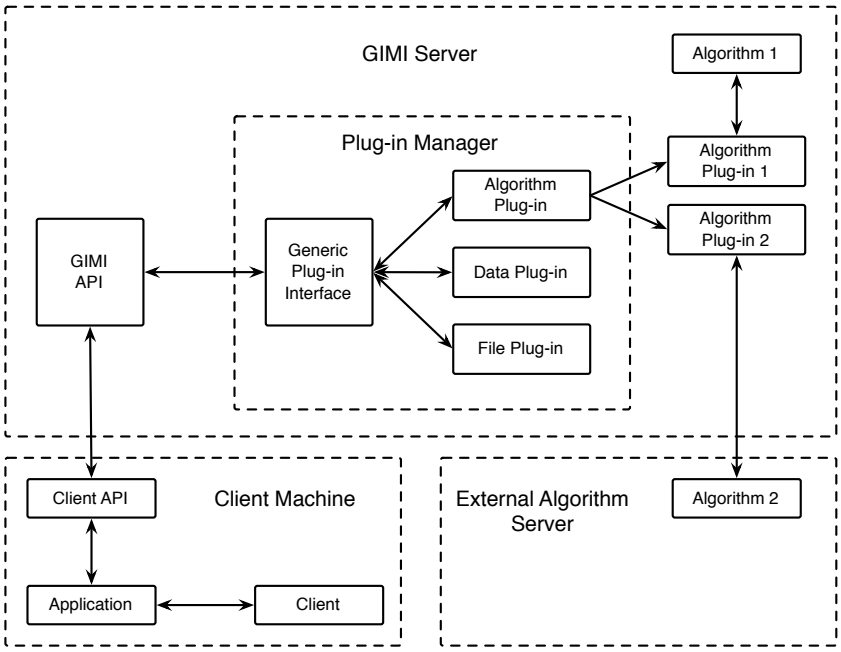


Figure 2.9: The sif plug-in architecture [3]

supports three types of plug-ins: algorithm plug-ins, data plug-ins and file plug-ins, and allows developers to add their own functionality. Algorithm plug-ins are constructed to execute bespoke algorithms, with a breast segmentation function which analyses relevant images being one example. Data plug-ins are able to expose any standard, structured data source. For example, the generic SQL database plug-in acts as a standard interface to a SQL data source. File plug-ins, on the other hand, can be used to connect to real or virtual file systems. For instance, for Picture Archiving and Communications Systems (PACS), a file plug-in has been constructed to retrieve Digital Imaging and Communications in Medicine (DICOM) files after a successful query using a data plug-in [3]. We notice that the sif API represents the sole interface to the middleware and therefore to the data at remote sites. As well as an exposure and transfer of shared data, support for homogeneous (when schemas across sources match) and heterogeneous (when schemas across sources differ) aggregation and federation of that data exists [45].

The infrastructure has been developed so that it aligns with the needs of clinical researchers, commercial organisations participating in the medical domain, healthcare providers, and those included in providing training facilities—all of which are concerned with accessing and sharing confidential data (see [46, 47, 48]).

2.3.2 EAC Mechanism

The GIMI project also gave rise to an evolving access control mechanism, which has been developed side-by-side with the sif middleware. This EAC prototype can be configured as either a module which communicates with the plug-in manager of sif, or as a standalone unit which sits in front of the data source. Figure 2.10 shows the former configuration of this prototype, and has the following workflow.

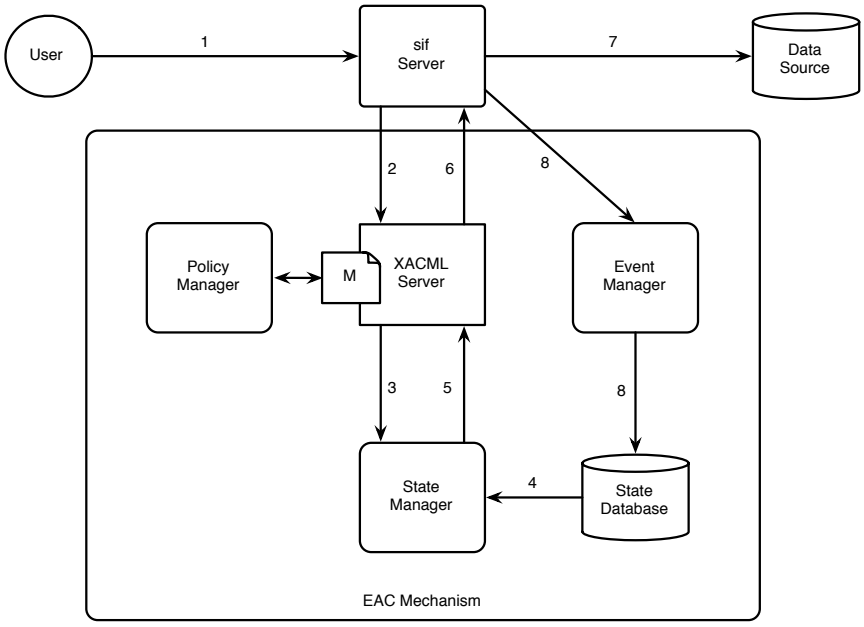


Figure 2.10: The EAC prototype

1. A user sends a request to access a resource in the Data Source (DS).
2. The sif Server (SS) forwards all user requests to the XACML Server (XS).
3. The XS, in turn, queries the State Manager (SM) for information about that user and requested resource.
4. The SM obtains this information from the State Database (SD). The SD is an audit history of user accesses and external system events.
5. The SM returns relevant state data back to the XS concerning its original query.
6. The XS is now able to make a decision based on the requirements in the metapolicy, M, and the data received from the SM—it then sends a response to the SS.
7. The SS enforces the decision by allowing or denying access to resources in the Data Source (DS).
8. This access is then logged in the SD via the Event Manager (EM).

We note that the Policy Manager (PM) is used to store, load and modify metapolicy requirements. In this thesis, we describe an abstraction in support of this EAC mechanism, specifically, the construction and analysis of these metapolicies, and how they manage the automatic evolution of policies. The use of metapolicies in this manner seeks to address the conceptual gap between high-level requirements and low-level implementations. Essentially, our abstraction provides a platform for the development of tools that can facilitate the analysis of metapolicies. One such tool, for example, can act as an interface to the PM and provide a means for the construction, analysis and deployment of metapolicies and policies.

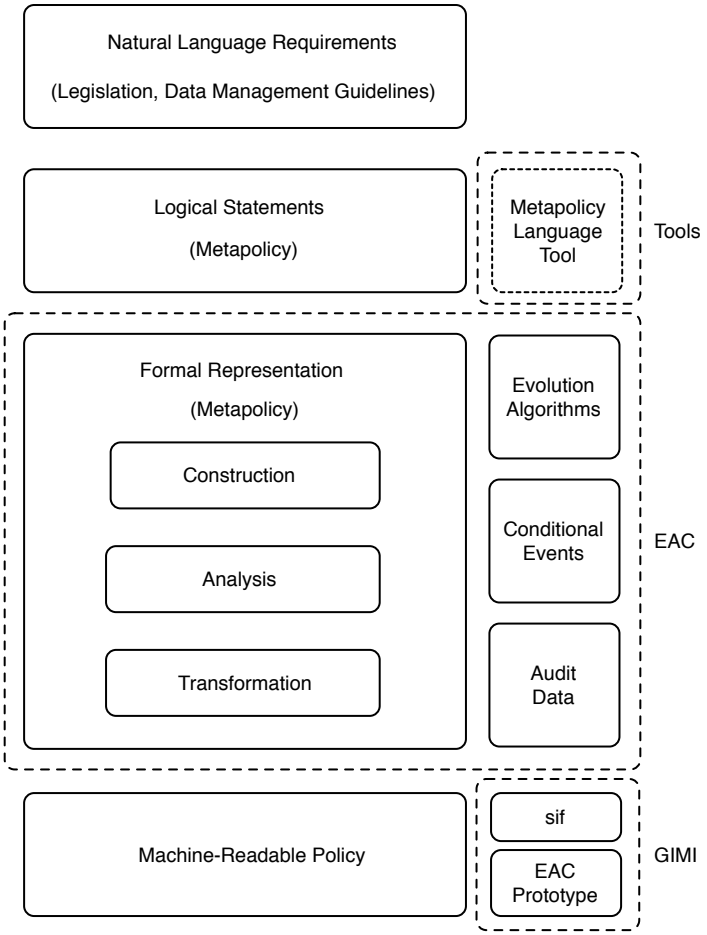


Figure 2.11: The conceptual gap between requirements and implementations

2.4 EAC Overview

Figure 2.11 illustrates how the work in this thesis, an EAC abstraction, forms a platform for the development of tools and technologies that can bridge the conceptual gap between high-level requirements and low-level policy implementations. Conceptually, this gap mirrors the relationship between a metapolicy at one level and a policy at another lower level. Formally, however, our EAC abstraction assumes one intermediate level where a metapolicy can be represented by a directed graph in which nodes are policies in different states during evolution, and edges are conditional events which trigger evolutions. We choose to formally model metapolicies in this manner as this facilitates effective model checking, where an unbounded system can be captured by constructing a specification that makes the system finite [49].

At the very highest level of Figure 2.11 are legal and ethical guidelines, as well as data management requirements which are written in natural language. At the next lower level are logical statements, possibly embodied in metapolicies, that represent the natural language of guidelines and requirements. The implementation of suitable tools that can perform this mapping from natural language to logical or even formal representations so that metapolicies can be automatically analysed is outside the scope of this thesis as this is a grand challenge in itself. However, the underlying theory behind such a tool that can perform analysis on

metapolicies, once converted to an appropriate representation, forms the fundamental work of this thesis: our EAC abstraction. The abstraction is the core component in this "stack" of Figure 2.11, and is used to demonstrate how metapolicies and policies can be formally captured so that they can be analysed and verified respectively, and, crucially, how the system evolves from one policy to the next based on observations in an audit data source and requirements in a metapolicy. At the very lowest level are implementations of these requirements captured in machine-readable policies. In our approach, these policies would be checked for correctness, and then be deployed one by one in appropriate middleware infrastructures according to the current state of the system.

2.5 Related Research

In this section, we consider related research from the wider research community. We examine four broad categories of research areas pertaining to novel access control models, mechanisms and practices. These four categories include audit-based access control, metapolicies and policy management, specification and formalisation of access control models, and dynamic and context-sensitive access control.

2.5.1 Audit-Based Access Control

Verhanneman et al. [7] have developed an architecture to support evolution of access control policies by means of requirements traceability. Requirements traceability is a chronological timeline from the origin of a requirement to its implementation, and thus a form of documented history. While this is not closely aligned to the goals of this thesis, the motivation behind this group's general body of work is broadly sympathetic to ours—attempting to use an audit record of transaction history to evolve an access control policy.

Dekker and Etalle [50] present a formal language and framework for a posteriori access control using the example of an electronic health record (EHR) system. The framework is based on a set of agents executing actions, which can be audited to ensure that their actions comply with the relevant policy. By having a log of actions, agents can prove accountability to auditors at any time, unlike traditional a priori access control, which typically uses a sole authority to verify authorisation at the instant that access is requested. Similarly, Røstad and Edsberg [51] investigate audit trails for patterns in user activity, which can possibly aid in designing better access control mechanisms for the healthcare sector. In another paper, Røstad and Nytrø [8] have recognised the need to create dynamic access control rules. By combining workflow information from medical guidelines, observations and audit logs, they attempt to minimise the effects of Povey's "optimistic" security paradigm [52] (the examination of logs for legitimate transactions after access) as it constitutes a security risk with potential for abuse. In [10], Verhanneman et al. have also identified this need for fine-grained, dynamic and adaptable access control policies to reflect constantly changing healthcare legislation.

Ravari et al. [53] have applied temporal constraints to a user's history of accesses producing the Generalized Temporal History Based Access Control (GTHBAC) model. GTHBAC complements the specification of authorisations (user-defined rules) by imposing time intervals over a user's history of accesses, which limits the validity of these authorisations and thus makes GTHBAC dynamic.

In this thesis, we concentrate on providing assurance that the use of audit data to evolve policies automatically based on high-level data source requirements captured in a metapolicy is correct. In so doing, we provide traditional a priori access control without the use of agents, audit trail analysis or time-constrained access histories.

2.5.2 Metapolicies and Policy Management

Hosmer in [54] was one of the first to document the role of metapolicies in policy management for access control. She describes how metapolicies can coordinate the security policies of multiple systems and builds the infrastructure for future research in metapolicy theory as reported in [55, 56, 57]. Avitabile [58] also deals with policy management by presenting an examination into the requirements for creating metapolicies where "policies govern the policy life cycle". Furthermore, Belokosztolszki and Moody [59] describe an architecture and language for specifying metapolicies in distributed access control systems. Additionally, Bell mathematically describes and models a multipolicy machine in [60] as a conceptual design incorporating methods of solving such issues as policy combination, policy conflict, conflict resolution, and policy evolution and precedence.

In this thesis, however, the term metapolicy refers to a description of the relationship between policies in different states, and captures the evolution in a labelled state transition system. We also focus on one policy protecting a single data source. Additionally, the autonomic nature of the EAC mechanism implies policy management is automatically handled, hence the motivation for this thesis—that such policies produced are accurate, reflecting the high-level requirements as mandated by the data source owner in a metapolicy.

2.5.3 Formalisation and Specification

Many papers have been written on the formalisation of access control models and policies; in this section, however, we only highlight those most relevant to our research. First, Power et al. [6] explore the feasibility of describing self-modifying policies. In their work, they aim to bridge the disconnect between data management legislation and requirements (at a high level of abstraction) and actual policy implementation (at a lower level of abstraction). A policy language and associated tool allow the capture and verification of these self-modifying access control policies. Ni et al. attempt to connect the divide between privacy policies and access control policies using a model that is able to capture obligation requirements (actions that must be performed sometime during access control evaluation), which are typically implemented in policy enforcement engines of access control mechanisms [61].

Zhang et al. [13, 15, 62] have presented a framework for evaluating and generating access control policies using the RW formal language [63]. RW is based on propositional logic and designed for modelling access control policies, as well as verifying their properties. The framework is also supported by a model-checking tool which can convert a policy written in the RW language into an XACML policy. Additionally, Bharadwaj and Baras [9] describe an architecture and mathematical framework for combining access control policies to enable an evolution of policies to suit changing requirements, in addition to verification of the resulting policies. Qunoo et al. [64] have also developed a modelling language and tool, known as X-Policy, to verify dynamic access control policies for web-based collaborative environments, and discuss challenges in dealing with state explosion issues in their model checking process. One of our aims in this thesis is closely related to the goals of these papers—to deploy verified access control policies in security-critical systems.

Ray [65] has formalised algorithms for a concurrent and real-time update of access control policies. Furthermore, Ray and Toahchoodee in [66] have proposed a formal spatio-temporal role-based access control model that is suitable for pervasive computing environments—this model incorporates two environmental factors in making access control decisions: time and location of entities. Fisler et al. [67] also present a formal analysis for access control policies in their dynamic environments using state machines. While their research is also sympathetic to ours, none of these approaches use audit data in granting or denying access, which we see as a crucial element in supporting access control.

Koch et al. [68] have developed a formal way to specify the evolution of access control policies using graph theory. They have modelled states as graph nodes and evolutions by graph transformations. The formalism is based on the semantics of graph transformation systems, where evolutions are described in terms of addition and deletion of rules and constraints over time—this ensures that coherence is maintained. The authors have also further demonstrated in [69], using graph theory and transformations, how to integrate two policies and possible strategies for dealing with conflicting rules and inconsistencies. These strategies, they claim, can be seen as "metapolicies". Later, Koch et al. have managed to combine their graph theory work into one framework which can potentially specify several access control models in [70]. In [71], Sandhu has also briefly explored the link between graph theory and access control models, concentrating more on the safety problem [12] and on dynamic role hierarchies.

Bryans et al. have employed the use of the Vienna Development Model Specification Language (VDM-SL) [72] and VDM++ [73] in specifying and analysing access control policies which have been written in XACML [74, 75, 76]. In [77], Bryans has also used Communicating Sequential Processes (CSP) [78] to reason about XACML policies. Similarly, Slaymaker et al. [79, 80] have formalised RBAC and XACML, and describe how different representations of policies can be translated to a normal form for comparison. Their formalisms can be employed in transforming our EAC policies to XACML (as described in Appendix B.7).

The work presented in this thesis is different from these approaches in that we formalise an EAC abstraction so that we can verify that the automatic evolution of policies consistently adheres to requirements embodied in a metapolicy, and that the transitioning to other states

based on observations in an audit data source is correct.

2.5.4 Dynamic and Context-Sensitive Access Control

Several papers have also been published concerning dynamic and context-sensitive access control. Location-Based Access Control (LBAC) has been discussed in several publications including [81, 82, 83], where access is contingent upon the location of users. In [84], Yu et al. present LTAM, a Location-Temporal Authorisation Model that concentrates on using the location of users and time of usage when controlling access to resources. Pu et al. [85] consider the Context Access Control Model (CACM), which combines context information with the Usage Control (UCON) model [38].

Context-sensitive access control has already been considered in numerous scenarios [86, 87, 88, 89, 90, 91, 92, 93, 94]. Notable solutions that have been proposed for tackling current context-sensitive access issues include that of Giuri and Iglio [95], in which the permission of an object may depend on the content of the object itself. For example, in a healthcare organisation, the doctor is only allowed to access and update patient records related to his or her patient. Additionally, Woo and Lam [96] have designed a distributed authorisation service utilising their Generalized Access Control List (GACL) language and authenticated delegation. In this design, GACL is used to capture authorisation requirements while authenticated delegation enables any server to delegate authorisation operations to dedicated authorisation servers.

Many context-aware access control models have been extensions of the RBAC model. Bhatti et al. [97] have developed X-RBAC, an XML-based role-based access control (RBAC) policy specification framework for enforcing access control in dynamic XML-based services. In [98], Zhang and Parashar consider the use of the Dynamic Role-Based Access Control (DRBAC) model, which allows context-aware access control for pervasive applications. The DRBAC model uses state machines to preserve changing permissions for subjects and resources. While this paper concentrates on dynamic access control for pervasive applications, the use of state machines in its role and permission assignments is akin (as we shall see) to our use of state machines as metapolicies that capture the evolution of policies. Bertino et al. [99] have included the time dimension to the RBAC model to form the Temporal Role-Based Access Control (TRBAC) model, while Joshi et al. [100] have extended this work by proposing the Generalised Temporal Role-Based Access Control (GTRBAC) model. Finally, Covington et al. [101] have proposed the Generalized Role-Based Access Control (GRBAC) model. In this model, RBAC is enhanced by applying roles to all entities in a system (in RBAC, only subjects are used as roles). Three types of roles are defined: subject, environment, and object roles. Environment roles capture environmental information, such as time of day or weather conditions, which can be used to mediate access control. In [102], Covington et al. subsequently implement the GRBAC model. The RBAC model has also been extended to include spatial information as documented in [103, 104]. Furthermore, both time and location have been integrated into RBAC, and addressed in works such as [105, 106].

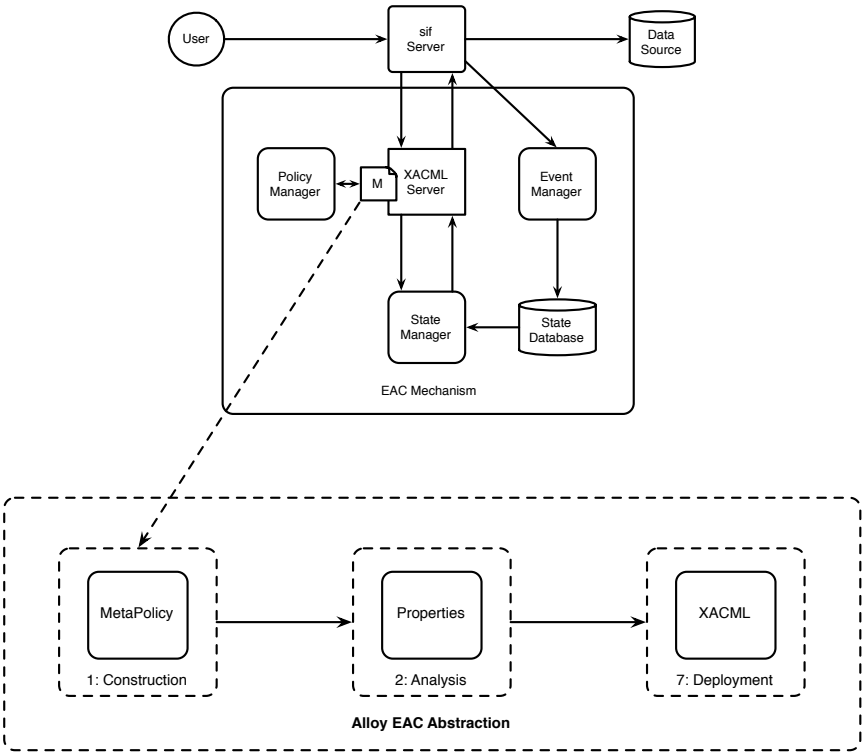


Figure 2.12: The research methodology

These research efforts require supplementary factors when evaluating access control decisions. However, neither the contribution of Giuri and Iglio nor the contribution of Woo and Lam considers context information as the main influence in their access control mechanism. In GRBAC, the definition of environment roles allows the model to partially address the challenge described and makes it more expressive and easier to use compared to RBAC. However, it is not feasible in practice because the vast number of potential environment roles make the system difficult to manage. The goals of GRBAC are also closely aligned with that of EAC. However, there are key differences which make EAC more practical—the incorporation of a metapolicy and audit data can automatically create policies that adhere to data source requirements.

Having described the background of, and related literature associated with, our work, we now progress to the next section which explains how we seek to answer our research question.

2.6 Methodology

We recall that our thesis aims to answer the following research question: can we develop a formal model for the construction, analysis, and automatic deployment of EAC metapolicies and policies, which provides assurance that the sharing of sensitive data is appropriate?

In answering this question, we shall utilise the following methodology as illustrated in Figure 2.12. The Alloy specification language [107] is used to capture our EAC abstraction, producing a formal executable model. This facilitates the construction of metapolicies, and the development of associated structures and algorithms that support policy evolution.

Our formal executable model also provides a framework for analysis and verification. Analysis involves a check of all system states prior to deployment, which confirms that metapolicies conform to certain desirable properties that can enforce correct evolutions. Verification, on the other hand, ensures that EAC policies adhere to the data source requirements of the metapolicy for a finite number of evolutions.

Verified policies are then translated to machine-readable XACML policies, which can then be deployed in an access control system such as sif.

In effect, development of this model allows us to perform analysis on small examples of metapolicies, which enables us to better understand how the system transitions from one complex state to the next, and produces a correct and elegant abstraction. Throughout, the work will be driven and validated by requirements and use cases derived from real-world applications.

2.7 Summary

This chapter provides the necessary background and general context for this thesis. We explained key access control ideas and theories including access control systems, policies (DAC, MAC, and RBAC), mechanisms, and models. Moreover, the XACML language has been briefly introduced with a description of an example XACML policy.

Then, the work in this thesis has been placed in context with respect to immediate research at the University of Oxford: the sif middleware, developed under the GIMI project, has been examined using several user scenarios concerning the secure sharing of healthcare data. We then described the sif architecture as supporting two types of users (application developers and data owners), and three types of plug-ins (algorithm, data and file plug-ins). We also provided an overview of an EAC prototype which has been implemented alongside sif. An overview of our EAC abstraction, formalised later on, is then presented and has been specifically developed for the analysis of metapolicies.

Next, we explored our area of study by conducting a literature survey. Related research has been grouped under four categories: audit-based access control, metapolicies and policy management, formalisation and specification, and dynamic and context-sensitive access control. Several papers in this section have revealed that formal models of access control is an active, ongoing field of research as the need to securely handle large amounts of sensitive data increases.

Finally, we discussed the methodology utilised in this thesis. The Alloy language is used to construct and analyse metapolicies, and verify and deploy policies because of its concise modelling notation and associated analysis tool, the Alloy Analyzer.

In the next chapter, we consider the need for evolving access control by describing how metapolicies can bridge the conceptual gap between high-level ethical and legal guidelines and low-level policy implementations, and the advantages of such an approach. We also present motivating examples of classes of access control requirements that will be used to drive and validate our work.

3

Evolving Access Control

3.1 Introduction

Evolving access control (EAC) is an approach to context-sensitive authorisation, which utilises the notion of a metapolicy and access history to automatically adapt access control policies. This autonomous evolution of policies can occur in response to user behaviour or as a result of changes in several environmental conditions, potentially producing a continuous sequence of evolutions with minimal intervention from system administrators. For example, a data source owner can impose the high-level requirement (in a metapolicy) that allows a user to download a file once. Once the user has accessed this file, a low-level policy is automatically deployed so that it now denies access to the file for that user—a system administrator is not required to manually update the policy to deny access to the user.

In light of this, the need for an EAC abstraction that can be used as the basis for reasoning about the correctness of EAC metapolicies and policies is imperative—evolved policies must continuously adhere to high-level requirements that a data source owner has mandated on his or her data source. In the example above, it is necessary to provide assurance that the first policy allowed access to that file, but after the user downloaded the file, the second policy denied access to that file after an automatic evolution. In this thesis, we aim to ensure that, initially, the abstraction is "fit for purpose", and then, subsequently, to provide a means of formally validating metapolicies.

This chapter describes the nature of metapolicies, highlights the advantages of EAC, and outlines examples of other high-level access control requirements that EAC can capture. The chapter concludes with four small model examples that illustrate how our EAC approach can be applied to real systems.

3.2 What is a MetaPolicy?

Legal and regulatory guidelines are typically written at a high level of abstraction—EAC aims to work closer to this level by directly capturing such guidelines in metapolicies and ensuring that relevant policy changes happen automatically. Some examples of such guidelines can include:

- "Allow Jim access to these files just once."
- "John can only update the payroll of employees given emergency authorisation."
- "Scientists can download no more than five gigabytes of data."
- "Once a doctor has accessed dataset A, he can no longer access other datasets."
- "If Eve exceeds the network bandwidth by a certain value, then limit her access."

We remind the reader that the motivation for the development of a formal model of EAC was due largely in part to the need for effective tools and technologies to support secure data management. In this thesis, our work captures the underpinnings of one such tool that can potentially represent higher order concepts (like the above guidelines) as logical statements that prescribe when policies should be updated—in essence, this illustrates the nature of our metapolicies.

Metapolicies can be a set of logical statements representing high-level requirements that a data source owner imposes on his or her data source. Such logical statements can describe the relationship between policies as they evolve, and, formally, can be represented as a directed graph (or state space), which theoretically captures the evolution of policies from an initial state. Consequently, this allows one to reason about the transition from one complex state to another on the basis of observed actions.

In this thesis, we concern ourselves with the investigation of this state space by developing a formal model, or abstraction, which facilitates the construction of small examples of metapolicy state spaces so that they can then be analysed. The examination of small examples of metapolicies in Alloy, especially, aligns with Jackson's "small scope hypothesis" [108], that a significant number of software faults can be discovered by testing for all inputs within some small scope. Of course, an EAC system that relied on only small examples of metapolicies would be useless. However, in [109], Jackson argues that "the point of the small scope hypothesis is that systems that fail on large instances almost always fail on small ones with similar properties, even if such small instances do not occur in practice. So by checking all small instances, we are effectively checking for large ones too."

In practice, policy writers are not exposed to formal methods; the purpose of these technologies is to underpin our approach, and drive the implementation of a suitable tool that policy writers can eventually use to capture, analyse and deploy EAC metapolicies and policies. While the development of such a tool is outside the scope of this thesis, we intermittently refer to its use so that our work can be placed in practical context with respect to providing assurance that the secure sharing of data is appropriate.

EXAMPLE 3.1. A typical scenario involving an EAC system and metapolicies can be envisioned as follows. David, a data source owner, approaches a policy writer, Mark, with requirements concerning access to his data source. David explains to Mark that, for now, he just wants to grant specific users the ability to access any resource but only once. Mark conceptually captures this requirement in a metapolicy using our potential tool.

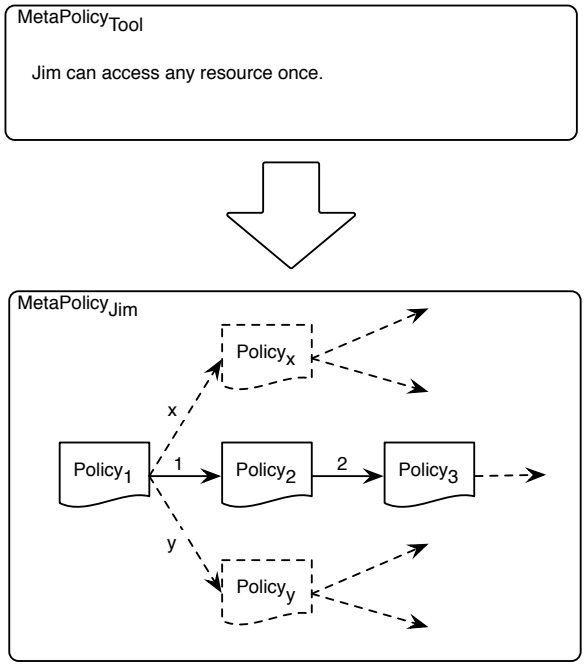


Figure 3.1: A metapolicy capturing policy evolution

Jim can access any resource once.

Using this tool, Mark can also check that this metapolicy possesses certain properties, and verify that all of its policies, during evolution, meet the metapolicy requirement. If the tool reports positive results—in effect, providing assurance to David that the secure management of his data is appropriate—Mark can then deploy this metapolicy in the EAC system with an initial policy, Policy_1.

Jim can access any resource.

Metapolicies are related to individual users or roles. Each user, such as Jim, has a state transition diagram associated with him or her, as shown in Figure 3.1, which illustrates the idea behind our EAC metapolicies. In the diagram, Jim's access is facilitated by the initial policy, which can evolve to other policies according to which resource he accesses, and guided by the requirement in his metapolicy. Furthermore, the autonomic nature of EAC minimises the need for Mark to update policies once he has constructed, analysed and deployed the metapolicy with an initial policy.

For instance, if Jim accesses resource 1, the EAC system is prompted to examine its access history. The audit reveals that Jim has accessed this resource once; an evolution is thus triggered from Policy_1 to Policy_2 blocking further access to resource 1 for Jim:

Jim can access any resource except resource 1.

If Jim then chooses to access resource 2, the policy evolves to Policy_3 for similar reasons.

Jim can access any resource except resource 1 and resource 2.

We see that transition operations (like accessing resources in this small example) can trigger evolutions from one state to another based on the *metapolicy* for that user. Transition 1 (accessing resource 1) has changed the policy's state from Policy_1 to Policy_2, whereas Transition 2 (accessing resource 2) has changed the policy's state from Policy_2 to Policy_3. These transition operations refer to a history of subject and resource accesses and *conditionally* change state because, in this example, Jim has accessed each resource (1 and 2) once.

We note that this is a simple, practical example of how a high-level requirement, embodied in a metapolicy, can be used to automatically evolve policies based on observed transactions—prior to using a suitable tool that can construct, analyse and deploy metapolicies and policies. In this thesis, we develop an EAC abstraction that forms the conceptual framework underlying the implementation of such a tool or tools.

3.3 Advantages

In this section, we outline two main advantages of our approach over existing means of access control models. EAC aims to be autonomous and centralised so that maintenance of policies is minimal, correct and secure.

3.3.1 Autonomous Control

As systems grow in size and complexity, those that can manage themselves given high-level objectives are ideal because they reduce the need for continuous administrator access [110]. To this end, an approach such as EAC has the potential to reduce the need for intervention by an authorised system administrator as policies are automatically created to reflect changes in the environmental context.

Typical access control policies can capture high-level requirements but with great difficulty. The relationships expressed require a manual translation down to the policy language to reflect any changes in the guidelines. Typical access control policies are also unaware of past transactions, and can only allow or deny access based on the current static policy, with intervention by a system administrator, who can be prone to error, to modify rules in such policies. EAC utilises audit information when making access control decisions, which allows the policies to automatically evolve based on changes in the system environment.

Possible use cases of EAC include a model of charging for data access such as, for example, in photo-sharing websites: individuals can pay to view a set of photos worth a particular value or accounts can be set up for fine-grained access allowing a limited number of views for certain users to specific photos. We can also apply this approach to online social networking utilities where the ability to view the profiles and albums of others can be charged credits or even awarded points. In other aspects, individuals can even listen to music from a data source at most once but unlock other music of the same genre after. In this mode, one would purchase a license which at a high level would dictate what resources can be accessed,

for how long, and the number of times. In effect, the license is a metapolicy in this charging model, which would automatically direct policy modifications at a lower level, ensuring that the modifications work within the boundaries prescribed by the license requirements.

As another example, consider the need to safeguard trading strategies when executed manually by traders. Trading strategies are guidelines for traders to follow and aid in making investment decisions. However, in reality, traders are subject to deviate from strategy, using tactics that diminish performance, or worse, are unethical or illegal. The set of rules that govern a trading strategy can be used as high-level requirements embodied in a metapolicy, that directs which trader is able to access (or make a trade) on which investment—automatically controlling the traders' tactics at a lower level. In a way, such an approach can prevent outlandish tactics from overstepping sensible boundaries.

Because of its autonomous nature, EAC can benefit several domains. Dependence on potentially error-prone system administrators overseeing access to massive amounts of sensitive data can be minimised, as data owners can trust that the mechanism operates automatically and securely as intended.

3.3.2 Central Authorisation

Another key benefit of EAC is that all information pertaining to authorisation resides in one place—as opposed to being split between databases and application logic. As previously noted, some high-level requirements cannot be easily captured by typical access control models such as RBAC and ACL. However, many can be captured in application logic but this leads to a distribution of authorisation information across different parts of a system, and updates to such a system can be complicated with potential for errors that might lead to security breaches. Therefore, in EAC, we wish authorisation information to reside in one place, as modifying a single access control policy is much simpler than updating several components.

3.4 Classes of Requirements

With the emergence of grid and cloud computing, new security concepts have prompted the need for defining suitable access control requirements [111]. Similarly, our EAC paradigm allows a variety of classes of access control requirements to be captured in metapolicies. However, these access control requirements can pertain to any conditional entity within the environment context, and can encompass factors ranging from the time of day to the size of the network bandwidth. Therefore, for the scope of this thesis, we collate some of these requirements into basic classes against which our contribution will be measured. We note that these classes of requirements are for motivational purposes only, and are intended to be neither prescriptive nor exhaustive.

Our classes have been selected from a number of projects across several sectors involving data storage and manipulation, and capture the needs of data source owners from a variety

of domains. The first two of these classes of requirements can be classified as binary requirements and counting requirements respectively. The next two classes involve additional information: the third concerns value; the fourth concerns data semantics. The fifth and sixth classes are more reactive requirements, while the next two are liveness and time-based classes of requirements. Our last class of requirements captures system factors like CPU load and network capacity.

1. *Binary.* A user can simply download or access a resource exactly once. (This requirement originated from a digital viewing application to support the NHS Breast Screening Programme.)

2. *Counting.* The user can only access a certain number of resources out of a total number. (This is a healthcare requirement and limits information flow so that patient confidentiality is preserved.)

3. *Subscription.* Based on a subscription fee, users can have access to any resource—some resources can be worth more than others. Alternatively, the user can have access to a certain worth of data. (This requirement is of special interest to commercial companies who offer their data via such subscription-based access.)

4. *Compartment.* A user can initially access resource A or resource B. However, once A has been accessed, the opportunity to access B is denied and vice versa (similar to the Chinese Wall Security Policy [37]). This is because a user who learns both the value of resource A and the value of resource B can possibly deduce other information, which can sometimes be highly confidential as occurs frequently in the financial sector—this prevents insider trading. (This requirement is derived from engagement with financial institutions.)

5. *Reject.* A user attempts to access restricted resources and after a certain number of times, is completely rejected from accessing all resources. (Extracted from the commercial sector, this is a requirement proposed by a large UK grocery for their online shopping facility.)

6. *Emergency.* An authorised user (like an administrator) can choose to override access rights for any other user—also known as the "emergency override" use case. (Derived from the healthcare sector, this requirement is essential in the event of an emergency when patient details are needed by the attending physician, who may not have access rights to that patient's details.)

7. *Liveness.* Access to a resource is denied if there has been no communication for an extended period of time. (This is a military requirement which prevents access from devices that have been lost in battle or fallen into enemy hands.)

8. *Period.* The user can access data in a set period. (This is an academic and research requirement, which enforces both fair and equal sharing of resources, and that resources do not get overly stretched.)

9. *System.* This class of requirements captures system factors like CPU load, network capacity, number of users and even weather conditions. (As an example within an access control context, if the network capacity were to exceed some threshold value, then users are denied access to resources.)

3.5 EAC Models

In this section, we present four simple case studies that drive the development of our EAC abstraction using these classes of requirements. In each model example, the assumption is that a data owner is concerned with providing access to, and enforcing high-level rules on, his or her data source using EAC. These high-level rules are instantiated from eight of the nine basic classes of requirements and specific to each of these examples. We note that the *System* class of requirements is, later on, modelled by triggering *external* events whenever a system factor limit (represented by an integer) has been crossed (for instance, if the network capacity for a particular connection has exceeded five units); this is a necessary simplification for the purpose of ensuring that our model is tractable.

The entire state space of each example is illustrated in Appendix A—it is this graphical state space that is captured by our metapolicies. It should be noted that while each model is limited in scope, they give us the opportunity to consider different classes of requirements in a straightforward and accessible fashion.

3.5.1 Images Example

In this example, we are provided with an online facility which allows users to pay for viewing images. There is an associated pricing model where some images are worth more than others:

Image	Price
i1	2
i2	2
i3	2
i4	4
i5	4
i6	4
i7	6

The owner of this online facility imposes the following high-level requirements on viewers of these images.

1. One can view any image at most once.
2. One can not have more than three accesses.
3. One can not view a set of images worth more than a value of 10.

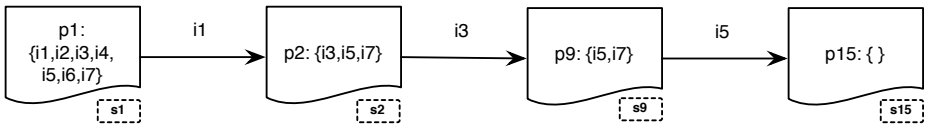


Figure 3.2: A possible evolution of the images example

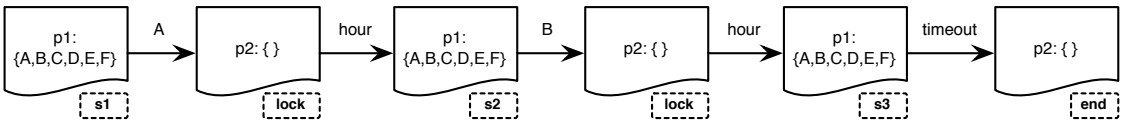


Figure 3.3: A possible evolution of the temporal example

4. One can view either only odd or only even images.

With EAC in mind, one possible set of access control evolutions is shown in Figure 3.2. The initial policy allows access to all images. After viewing image i1, a policy is created which allows access to images i3, i5 and i7 (only odd images). Viewing image i3 creates another new policy that allows access to images i5 and i7. Finally, viewing image i5 produces yet another policy which rejects all accesses (no more than three accesses). The entire state space for this example is illustrated in Appendix A.1.

We see in this example that the first rule is an instance of the first class of requirements of Section 3.4, the second rule that of the second class, and so on.

3.5.2 Temporal Example

With this example, we utilise EAC in a health facility which allows researchers to download archived datasets according to time constraints.

The rules for accessing these datasets must follow these guidelines:

1. A researcher can download only one dataset per hour.
2. Attempts to download further datasets within that hour result in a lockout.
3. Access to the facility is terminated if there has been no access for 2 hours.

Based on these three requirements, one possible evolution of access control policies is shown in Figure 3.3, and illustrates the access control policy at each point in time. The initial policy allows access to any one of the datasets. After being authenticated for access by the health facility, a researcher decides to download dataset A. The access control policy immediately evolves to deny access to all datasets for an hour, according to requirement 1. An external event is triggered after that hour which signals an evolution that subsequently re-allows access to all datasets. The researcher then downloads dataset B—the policy will again deny all access for another hour. Once more, an external event is triggered after that hour allowing access to all datasets again. At this point, the researcher has downloaded the

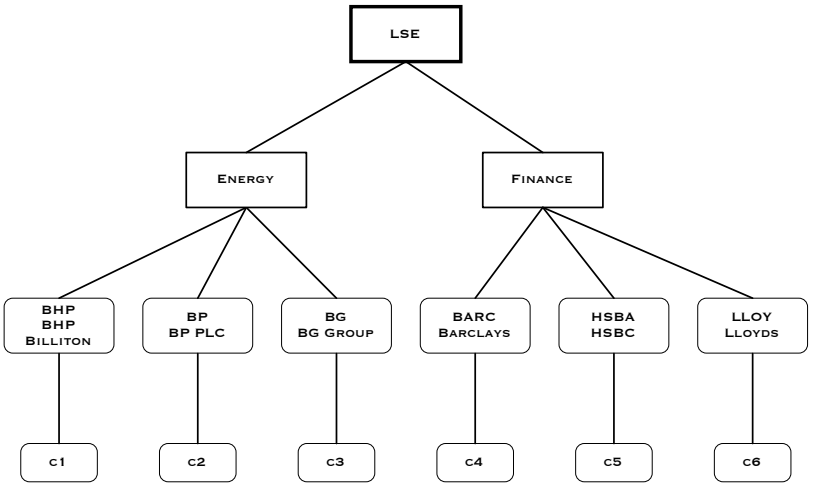


Figure 3.4: A hypothetical hierarchy of companies

necessary data, and allows time to pass without any download activity. Consequently, the policy denies access to all resources after 2 hours in compliance with requirement 3.

We observe that in this evolution trace, requirement 2 does not come into effect. However, the entire state space diagram in Appendix A.2 captures this scenario—if the researcher attempts to download any datasets while in a lock state, the policy also evolves to a permanent end state. We additionally note that the state space of this example continues indefinitely until the researcher has been rejected or timed-out.

These rules have all been derived from the classes of requirements of Section 3.4: rule 1 is an instance of the eighth class, rule 2 is an instance of the fifth class, while rule 3 is an instance of the seventh class.

3.5.3 Compartment Example

Here we apply our EAC theory to model the Chinese Wall security policy (formally known as the Brewer-Nash model [37]). The rationale behind the Chinese Wall security policy is that users are only allowed to access information which will not conflict with information that he or she already has in possession. A conflict arises when knowing some information in one area can possibly corrupt the intentions for an act in another area, which, in some instances, can lead to violations of legislation [112]. As an example, the Chinese Wall security policy can be most easily visualised as the code of practice that must be followed by a market analyst working for a financial institution providing corporate business services. Such an analyst must uphold the confidentiality of information provided to him by his firm's clients; this means he cannot advise corporations where he has insider knowledge of the plans, status or standing of a competitor. However, the analyst is free to advise corporations which are not in competition with each other, and also to draw on general market information.

In this example, we shall concentrate on two popular sectors of any stock market: Energy and Finance. All corporate information can be filed in reports in a hierarchical order with multiple levels of significance as shown in Figure 3.4.

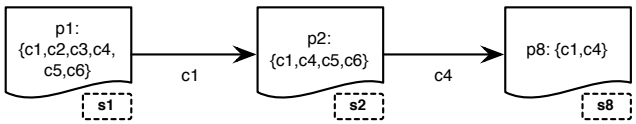


Figure 3.5: A possible evolution of the compartment example

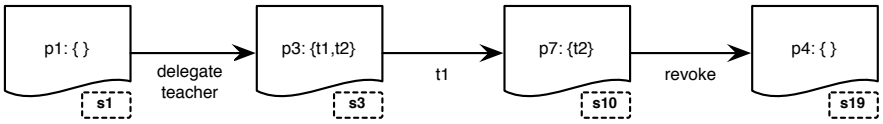


Figure 3.6: A possible evolution of the delegation example

At the lowest level, we consider individual items of information (c1 – c6), each concerning a single corporation. At the next highest level, we group all items of information which concern the same corporation into a confidential company report (BP). At the next highest level, we group all company reports whose corporations are in competition (Energy). Each such group is called a conflict of interest class, so that, for example, Energy and Finance are both conflict of interest classes.

The initial requirements for accessing company data are simple and in accordance with a (basic) Chinese Wall security policy:

- One can initially access any company report.
- Further accessed reports must not be in the same conflict of interest class as the reports already accessed.

For instance, the initial policy allows access to all reports c1 – c6 in Figure 3.5. Choosing to read report c1 denies access to other reports in c1's conflict class (c2 and c3) because knowing both sets of data can lead to misuse of such information (like insider trading).

Accessing report c4 next restricts the policy choices further since that means c5 and c6 are ruled out because they appear in the same conflict class as c4. The final policy only permits repeated access to reports c1 and c4.

This example's only requirements are wholly derived from the fourth requirement class. We choose to explore this class alone and in-depth because of its potential broad use. Its corresponding state space is illustrated in Appendix A.3.

3.5.4 Delegation Example

Delegation is the process whereby an active user on a system is able to grant some authorisations to another active user in order to perform some functions. In this example, we use EAC to model this delegation concept using the sixth requirement class.

SoftEng is an academic department that uses a computer system to access and store student (u1 and u2) and teacher (t1 and t2) data including grades, salaries and personal

Reference	Requirement	Origins & Motivation	Sample Example
1	Binary	Healthcare	Images; Delegation
2	Counting	Academic	Images
3	Subscription	Commercial	Images
4	Compartment	Financial	Images; Compartment
5	Reject	Commercial; Military	Temporal
6	Emergency	Healthcare	Delegation
7	Liveness	Military	Temporal
8	Period	Academic	Temporal

Table 3.1: The basic classes instantiated by examples

details. This sensitive data can only be fully accessed by the programme director, although, occasionally, he likes to delegate a subset of these permissions to his administrative assistant according to the following requirements.

- The administrative assistant can be delegated responsibility to read either students' records or teachers' records.
- Records can only be modified on a per-session basis provided authorisation has been delegated (logging out revokes permissions).
- Once a record has been accessed and modified, it can not be modified later on.
- The director can revoke delegated permissions at any time.

As illustrated in Figure 3.6, the initial policy for that assistant denies access to all records.

The director then delegates permission to read and/or update teacher records. Permissions are reverted to the initial policy (denying all access) when either the director has revoked permissions at a later time or the assistant has terminated the login session.

Three of the four rules imposed on the SoftEng data source are all derivatives of the sixth class of requirements (the third rule is an instance of the first class of requirements). The entire state space can be viewed in Appendix A.4—external events here also trigger state changes.

Table 3.1 summarises the mapping of the basic classes to our examples. At this point, we have identified four examples that can be captured by our EAC abstraction. The evolution of policies according to the respective requirements can be represented by a metapolicy state space, which enables the analysis of metapolicies as we see in the next chapter.

3.6 Summary

At the beginning of this chapter, we introduced a small, practical example showing how our EAC abstraction represents a conceptual framework that aids in the construction, analysis and automatic deployment of EAC metapolicies and policies. Ultimately, this framework forms the underlying theory of potential tools that can facilitate this end-to-end process.

Then, we proposed that EAC aims to satisfy the need for a centralised, autonomic and context-sensitive access control model in today's world as the complexity of access control requirements increases to accommodate emerging legal and ethical guidelines.

Next, we characterised several broad classes of requirements that drive the verification and validation of our EAC abstraction. These classes have been identified as motivating examples only and not a complete list.

Furthermore, four small EAC models have demonstrated how data source owners can impose such requirements using an EAC system. Small models like these four are used to propel development of our EAC abstraction. The images example has instantiated four of these requirements pertaining to an online image facility. The temporal example has dealt with the time-based classes of requirements, whereas our third example has only utilised the compartment class of requirements. The last small example has considered delegating emergency rights. In each case, the evolution of policies based on the respective requirements produces a state space—in the next chapter, we seek to formally capture these state spaces with metapolicies.

The next chapter explores the major contribution of this thesis—a formal executable model of our EAC abstraction.

4

EAC Abstraction

4.1 Introduction

In this chapter, we present the main contribution of this thesis—a formal abstraction that allows the construction, analysis and automatic deployment of EAC metapolicies and policies. Our abstraction allows the construction of small models of metapolicies so that we can demonstrate how its properties can be analysed, and how policies can be verified and deployed. Crucially, our abstraction offers a stable platform for subsequent development of applications, technologies and tools that support secure data management.

We use the Alloy [108] language to model our EAC abstraction.[1] Alloy has been termed as a lightweight formal method and its language is based on first-order logic similar to formal specification languages such as Z [113], VDM [72] and B [114]. However, unlike these languages, Alloy models are amenable to full automatic analysis using the associated tool, the Alloy Analyzer. The Analyzer is designed for analysing state machines with operations over complex states, which fits perfectly with our needs in this thesis. Moreover, Alloy has already been extensively employed in checking numerous examples of access control models [115, 116, 117, 118]. For instance, Alloy has been used to validate the integration of policies in [119], as well as to detect conflicting features of a model [120]. Additionally, Alloy has been used to formally model the building of state machines [121] as well as the analysis of graph transformation systems [122]. Alloy has even been utilised in modelling and analysing access control policies used by Amazon Web Services, presently one of the most popular cloud computing infrastructures [123]. Therefore, naturally, we attempt to use this tool in our research because of its proven use in modelling access control policies, coupled with its extensibility.

We begin this chapter with an introduction to the Alloy language and its associated tool, the Alloy Analyzer. We then proceed to formally define our EAC abstraction which includes rules, policies, metapolicies, and associated structures and algorithms that facilitate the evolution of state. Next, desirable properties of metapolicies are investigated, followed by the verification and deployment of policies. This chapter concludes with a small example demonstrating how our formal model aids in the construction of a small metapolicy example,

[1] See Appendix B for the original Z model.

the analysis of that metapolicy, the verification of policies as they evolve according to the requirements of that metapolicy, and the automatic deployment of those policies.

4.2 Alloy

Alloy is a declarative, structural language, that is fully analysable [107], meaning that the analysis is a form of constraint solving in which solutions are located within a finite space. Alloy is the smallest modelling notation that can express a useful range of structural properties—its notation is inspired by the declarative and structural nature of Z, but with a subset of features that is just essential for object and state modelling, like sets and relations. Although this can limit expressibility, it allows the language to be automatically analysed—an aspect influenced by the Symbolic Model Verifier (SMV) [124] and, to a lesser extent, Promela [125]. It is the integration of these features that makes Alloy practical in terms of declarative modelling and analysability of states.

Declarative models, such as those implemented in Alloy, describe a system's state and behaviour by stating constraints or properties. Declarative models do not explain how states are constructed, or how an execution obtains a new state from an old state. Rather, they provide constraints that define a correct state and how a new state is related to an old state. Declarative languages encourage incremental modelling—constructing a model incrementally with the Alloy Analyser, simulating and then checking, exposes errors much faster than programming and debugging. Additionally, the feedback from simulating motivates one to think about the important properties of the model.

In software engineering, conceptual models, like Fowler's "analysis patterns" [126], are typically much smaller than the systems they represent. Similarly, Alloy models are really *micromodels*—it is quite possible to model a system whose implementation could potentially contain thousands of lines of code in just 10 or 20 lines of Alloy [107]. The language itself is relatively small, yet powerful and flexible enough to model complex applications. Instead of modelling all details of a system, Alloy allows one to concentrate on more crucial aspects of the design. Small models can be developed to represent key features, without having to worry about the overall larger, more complex model.

4.2.1 Language and Logic

All structures in Alloy are constructed from *atoms* and *relations*. In essence, they represent the basic entities and the relationships between them. A *signature* declares a set of atoms using the **sig** keyword.

```
sig X {}
```

This set of atoms can also be extended to represent subsets. For example, set X1 is a subset of set X—extensions of a signature are mutually disjoint.

```
sig X1 extends X {}
```

An **abstract** signature has no elements and must be extended for use within the model.

abstract sig X {}
sig X1 **extends** X {}
sig X2 **extends** X {}

Extensions of an abstract signature partition that set. In the example above, we have X = X1 ∪ X2 and X1 ∩ X2 = {}.

In Alloy logic, all values are relations. Relations must be declared as fields in signatures.

sig X {a: b}

This signature declaration defines a relation a whose domain is X with range b.

a: X → b

Even functions are treated as relations—the Z modelling language also treats functions in this manner. However, Z is still significantly different from Alloy including its ability to distinguish between scalars, sets and tuples—in Alloy, a tuple is represented by a singleton relation, and a scalar by a singleton set.

Predicates, Functions, Facts and Assertions

In Alloy, constraints that always hold are embodied in a **fact**. These types of global constraints are assumptions on the model regardless of the order and number of facts. They can be labelled with a unique mnemonic name. In this thesis, a **fact** is used to describe how we transition from one state to the next.

fact Name { expression }

Another type of named constraint, **pred**, are predicates that only hold when invoked (unlike a **fact**). When modelling a system, these predicates can either be included or excluded, which serves to quickly analyse the behaviour of the model with and without that constraint. Predicates can have zero or more arguments, and expressions must be used for each argument. In this manner, a predicate can be used to embody an operation that describes a state transition by restricting the relationship between the current and next state. Here, predicates will be used to model the core evolution process.

pred Name [arguments] { expression }

The named expression, **fun**, are functions with zero or more declarations for arguments (similar to predicates), and a declaration for the result. However, functions encapsulate expressions for reuse. Whenever such functionality may be needed frequently, it is best to package the constraints as a function instead of a predicate. Functions are used to represent helper procedures that support our core evolution process.

fun Name [arguments]: result { expression }

An *assertion* is a constraint which follows from the facts of the model. Assertions are used to check models for flaws and will report a *counterexample* if some fact is not upheld. Assertions are also useful for checking the properties of a model.

assert Name { expression }

We shall see later on that assertions are used to check for the desirable properties of metapolicies, and that metapolicy requirements are maintained in each state.

Multiplicity

Alloy uses five keywords to express multiplicity in sets and relations, and to describe quantification and signature declarations. These five keywords are:

1. **set**: any number
2. **one**: exactly one
3. **lone**: zero or one
4. **some**: one or more
5. **all**: all

A straightforward example is given below.

firstName: **one** Name
middleNames: **set** Name
lastName: **some** Name

This states that firstName can only be a scalar in the set Name, that middleName is a subset (possibly empty subset) of the set Name, and that lastName is a nonempty subset of Name.

The following are examples of relation multiplicity.

addrA: Name → **one** Address
addrB: Name → **lone** Address

We see that addrA is an injective function whose domain is Name, whereas addrB is a function that is partial over the domain of Name.

With respect to quantification, we have the following example which says that all names are mapped in any address book:

all b: Book, n: Name | **some** a: Address | contains[b, n, a]

Finally, in signature declarations we have this example that states there can only be one Name atom.

one sig Name { }

Commands and Scope

To actually analyse a model, the **run** command instructs the Alloy Analyzer tool to search for a solution satisfying all the constraints of the model in the form of predicates and facts, which is known as searching for an *example*. The **check** command, on the other hand, instructs the tool to search for a *counterexample* to an assertion. In addition to these commands, a *scope* can be defined as a number which bounds the size of these examples or counterexamples. By default, if no scope is defined, each top-level signature is bound to three atoms. Furthermore, if an **abstract** signature has no explicit scope but its extensions have bounds, the considered scope is the sum of those of its extensions. A signature declared with the multiplicity of **one** has a scope of one atom.

run [predicate]
check [assertion] **for** [scope]

We use the **run** command to find an example where the system evolves a number of times (as dictated by the scope) based on the model constraints. The **check** command is used to verify the properties of our model.

4.2.2 Analysis

The Alloy language was designed to be analysed with its associated constraint solver, the Alloy Analyzer. The Analyzer aims to achieve an assignment of variables that makes all constraints true—as represented in the model with the Alloy language. Constraints are translated into Boolean formulae and then solved using a SAT solver.

As mentioned previously, an essential feature of the Analyzer is the declaration of a scope, which constrains the number of signature atoms in the model, and an exhaustive search within that scope for *instances*. This is because Alloy supports two types of analysis: simulation, in which the state space of a dynamic model is generated in an example, and checking, in which the properties of that model are verified by attempting to produce a counterexample.

Instances can be displayed graphically, textually or in tree form. Visualising an instance allows one to step through states in a sequence or view how a counterexample violates the model. In this thesis, we visualise counterexamples to examine how they violate the model—this aids in the debugging process. The Alloy Evaluator is a command line tool (tied to the Visualiser), which also aids in debugging. It allows the user to type in Alloy expressions and execute them for immediate feedback. Moreover, functions, predicates and expressions can be individually executed from a call stack, allowing one to trace the source of errors or investigate the state space.

The Alloy Analyzer is neither a theorem prover nor model checker, but a model *finder*, and finds solutions that satisfy constraints declared in the Alloy model. The Analyzer is designed to check state machines with operations that transition from one complex state to another. Model checkers, on the other hand, are designed to analyse state machines that are composed of other state machines running in parallel that have simple states. Furthermore,

the Analyzer is a "refuter" rather than a theorem prover in that it aims to prove assertions false by generating counterexamples. When theorem provers fail to prove a theorem, it can be difficult to ascertain whether the theorem is invalid, or if the proof strategy is faulty. If the Analyzer fails to find a counterexample, the assertion may still be invalid, but by selecting a scope that is sufficiently large, this becomes very unlikely.

Examples

When a model is analysed by running a predicate, the Analyzer searches for an instance of an analysis constraint—an assignment of values to variables that satisfy the constraints of the model. The analysis constraint is the constraints of predicates conjoined with both implicit and explicit facts of the model—finding an instance therefore means seeking an example in which both the predicates and facts of the model hold. Implicit facts arise from constraints associated with signatures, field declarations and arguments. Explicit facts are stated in **fact** statements. Variables that are assigned in an example include signature sets, field relations and predicate arguments [108].

Counterexamples

In contrast, when a model is analysed by checking an assertion, the analysis constraint is the negation of that assertion's constraint conjoined with the facts of the model. The instance found in this scenario is known as a counterexample—the facts hold, but the assertion fails to hold. We reiterate that Alloy does not aim to establish proof that an assertion holds, but to refute this claim.

Counterexamples are reported when assertions are invalid. Assertions are checked against a finite set of test cases (valid assignments of variables)—if an assertion does not hold for one of these test cases, this case is described as a counterexample. Thus, finding a counterexample is the quickest indication of an incorrect model. The absence of a counterexample, however, does not necessarily guarantee correctness as one may still exist in a larger scope. Nevertheless, the developers of the Alloy Analyzer have justified its use of working in a finite scope by citing Jackson's "small scope hypothesis" [108]: a significant number of software faults can be discovered by testing for all inputs within some small scope. Using several implementations of data structures, this hypothesis was investigated in [127] to reveal that exhaustive testing in small scopes can attain complete code coverage.

4.2.3 An Example

The example below (adapted from [108]) lists a small model for an address book that simply maps names to addresses . We define three top-level signatures: Name, Address and Book, which maps a Name to an Address. Our predicate, contains, takes three arguments and states that if we map the Name to an Address, then it should be contained in the relation addr i.e. a name must map to an address in that address book. The **fact** states that all names are

mapped to one or more addresses (denoted by the multiplicity keyword **some**). The lookup function returns an address given an address book and name.

```
sig Name, Address { }
sig Book { names: set Name, addr: Name → Address }

pred contains[ b: Book, n: Name, a: Address ] { n → a in b.addr }

fact everyNameMapped {
  all b: Book, n: Name |
    some a: Address | contains[ b, n, a ]
}

fun lookup[ b: Book, n: Name ] : set Address { b.addr[n] }

assert lookupResults { all b: Book, n: b.names | some lookup[b,n] }

pred show [ ] { }

run show for 4 but 1 Book

check lookupResults for 4 but 1 Book
```

An assertion is used to check that every name look up in the address book must produce some results—this technique of checking properties is valuable in verifying the correctness of the model. The **run** command attempts to find an example satisfying the constraints by running the dummy predicate show, but with an explicit scope set to four Name and Address atoms but just one Book atom. Running this command produces this output:

```
Executing "Run show for 4 but 1 Book"
   Solver=sat4j Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
   235 vars. 29 primary vars. 356 clauses. 154ms.
   Instance found. Predicate is consistent. 37ms.
```

The **check** command executes the lookupResults assertion and searches for any counterexamples, also using four Name atoms, four Address atoms and one Book atom. We know that counterexamples violate the constraints of the model, which can provide deeper insights into the model. Checking this assertion produces no counterexamples:

```
Executing "Check lookupResults for 4 but 1 Book"
   Solver=sat4j Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
   308 vars. 34 primary vars. 467 clauses. 34ms.
   No counterexample found. Assertion may be valid. 10ms.
```

We can now use Alloy to specify our EAC abstraction. A simple running example from the point of view of a policy writer using a tool will be identified as **POLICY WRITER**. We remind the reader that such a tool is the long-term goal of our work, much like the potential tool of Example 3.1, and its development will rely on the research in this thesis. For now, we assume its existence as a high-level interface that policy writers can use to directly access our executable EAC model at a lower level from this point on.

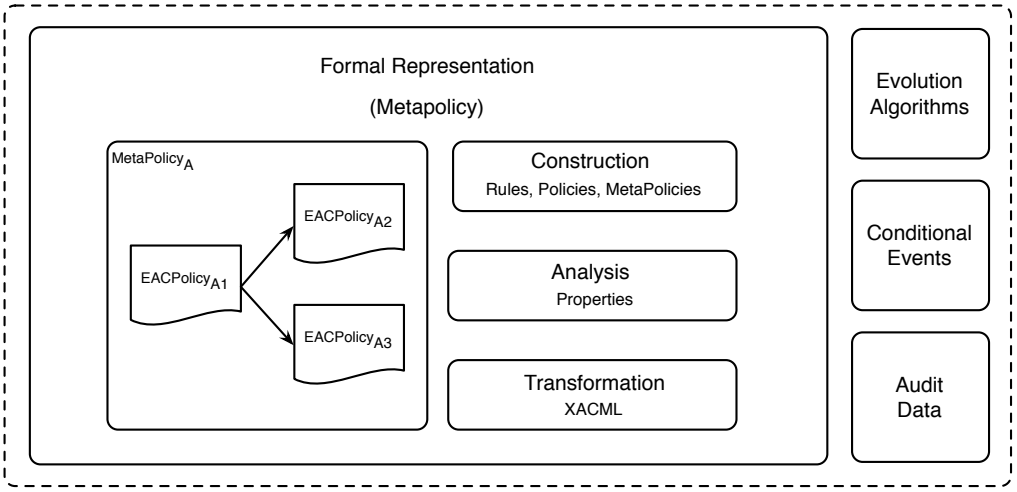


Figure 4.1: An overview of the EAC abstraction

4.3 Abstraction Overview

Figure 4.1 presents a high-level view of our abstraction and identifies the main components. Formally, metapolicies can be represented as directed graphs where nodes represent policy states and edges represent conditional events. As noted, such a representation facilitates model checking as a finite model can capture an infinite system. Each EAC model exhibits a metapolicy, which describes the relationship between policies; policies are then made up of rules. These basic structures are first defined and allow us to specify a small metapolicy example for that EAC model by graphing its state space. In effect, we assume the existence of small metapolicy examples so that we can demonstrate how they can be analysed and how their associated policies can be verified. Most importantly, this allows us to reason about the transition from one state to the next and the events that trigger such an evolution.

An audit data source is then specified, which sequentially records all observations including user accesses and system events. Conditional events, on the other hand, are finite sequences of observations or patterns that can reflect the metapolicy requirements. Finally, we define a framework of functions that essentially forms our evolution algorithm. The algorithm indicates when an evolution should occur by searching for conditional events in the audit data source—if such a pattern is found, the current state is automatically updated to another state according to the metapolicy, otherwise the current state remains as is.

Each EAC model generates a state space according to the classes of requirements and, to some extent, the number of resources. The metapolicy is constructed from this state space for each user or role, and then loaded into the framework of functions so that analysis and verification can take place. Analysis checks for certain key properties of metapolicies, while verification ensures that policies meet the requirements of that metapolicy. If requirements for that model change or the number of resources are modified, then the new metapolicy state space generated is re-mapped to a specification.

Finally, policies are transformed to XACML and deployed in suitable middleware (such as sif) so that we can validate the behaviour of our metapolicy examples.

4.4 Rules, Policies and Metapolicies

We begin the development of our EAC abstraction in this section. We first formalise the basic entities that allow us to construct small examples of metapolicies for particular EAC models. Metapolicies capture all the possible states that an initial policy can attain via evolution when triggered by some event; policies dictate which resources in a system that a subject can or can not access—in accordance with rules.

Policies consist of rules; rules refer to subjects, resources, effects and actions. Our policies support only positive authorisations, meaning that our approach to access control is *closed* [1]. For example, we can declare a policy rule that explicitly allows subject A to perform some action on resource B. If no rules are declared concerning a subject and resource, then by default, access is denied.

With this in mind, we can declare EACRule, a signature which defines which subject is permitted access to which resource and the subsequent action that must be taken.

```
abstract sig EACRule {
    rid: RuleID,
    subject: Subject,
    resource: Resource,
    effect: Effect,
    action: Action
}
```

An EAC policy is then comprised of an injective sequence[2] of rule identifiers. In Alloy, injective sequences can be modelled using the hasDups library function, which returns true if there are duplicate elements in a sequence. We therefore negate this constraint to ensure that the rules sequence has no duplicate RuleID atoms.

```
abstract sig EACPolicy {
    pid: PolicyID,
    rules: seq RuleID
} {
    !rules.hasDups
}
```

EXAMPLE 4.1. A rule that a subject, alice, is permitted to write to a resource, file, (an image, dataset, etc.) can be captured as the following.

```
one sig rule1 extends EACRule {} {
    rid = r1 and subject = alice and resource = file and effect = Permit and action = write
}
```

We can then define a policy referencing r1, as well as rules r2 and r3, as follows.

```
one sig policy1 extends EACPolicy {} {
    pid = pid1 and rules = {(0 → r1) + (1 → r2) + (2 → r3)}
}
```

[2] An *injective* sequence is one that does not include repeated elements.

Metapolicies describe the relationship between policies as they evolve based on high-level requirements. In effect, this creates a state space of policy evolutions. MetaPolicy, defined below, captures this state space of policy evolutions in a directed graph.

```
abstract sig Priority { mpc: MPConditionID, state: StateID }

abstract sig MetaPolicy {
    mid: MetaPolicyID,
    initialstate: StateID,
    rule: RuleID → lone EACRule,
    policy: PolicyID → lone EACPolicy,
    states: StateID → lone PolicyID,
    transitions: StateID → (seq Priority)
} {
    initialstate in dom[states]
    all s: ran[transitions] | s.state in dom[states]
    all r: dom[rule] | rule[r].rid = r
    all p: dom[policy] | policy[p].pid = p
    all i: ran[states] | i in dom[policy]
    all j: ran[policy] | all k: ran[j.rules] | k in dom[rule]
}

fact canonicalisation { no disj a,b:Priority | a.mpc=b.mpc && a.state=b.state }
```

Partial functions in Alloy are modelled with the **lone** multiplicity keyword representing the constraint that many StateID atoms can be mapped to zero or one PolicyID atoms. The rule, policy and states functions map a RuleID element to an EACRule element, PolicyID element to an EACPolicy element and a StateID element to a PolicyID element respectively. We note that the MPConditionID element is a unique identifier that is associated with a conditional event which can trigger policy transitions—for now, it suffices for us to consider conditional events as abstract entities. The transitions function captures all the possible policy transitions that can occur—the relation maps a starting state to a sequence of Priority atoms (Alloy can only model sequences of atoms), inherently embedding priority levels in the case that several transitions are triggered. Priority signatures consist of two fields—one is related to the next state in a transition, and the other is related to the MPConditionID that triggered the change of state. Moreover, according to [128], we also need to add a canonicalisation fact to ensure that Priority signatures with the same contents are represented by the same atom.

The first and second constraints on the MetaPolicy definition indicate that the initial state, as well as the states in the range of transitions, must be contained in the domain of states, ensuring that all states in the directed graph are connected. The third and fourth constraints guarantee that all rule identifiers in the domain of rule are each mapped to an EACRule element, and that all policy identifiers in the domain of policy are each mapped to an EACPolicy element respectively. The fifth constraint ensures that all policy identifiers in the range of states are contained in the domain of policy, while the final constraint states that all rule identifiers in each policy rules sequence is contained in the domain of rule.

The states function associates state identifiers with policy identifiers so as to avoid am-

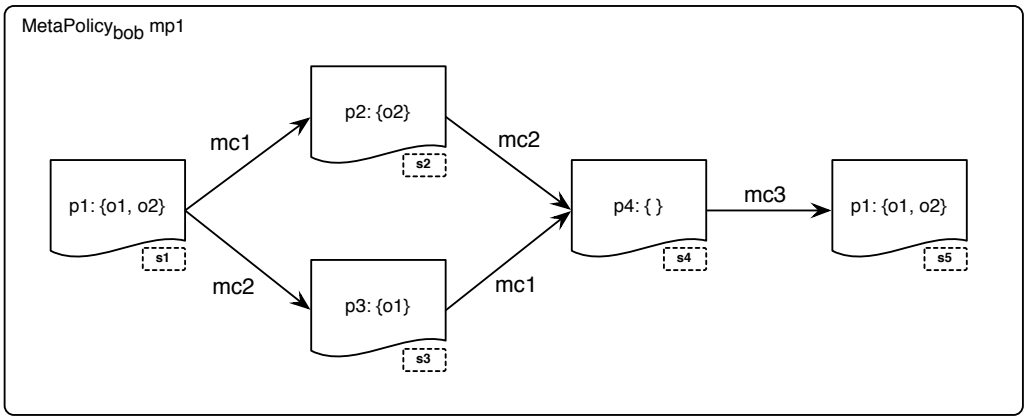


Figure 4.2: The association between policies and states

biguity in changing policies where cycles are possible, as it is possible to evolve to the same policy several times (as illustrated by p1 in Figure 4.2), but the fact that the system is in a different state is important, as this will have an impact on subsequent transitions because data recorded in an audit data source is used to determine when a change of state is necessary.

EXAMPLE 4.2. Figure 4.2 illustrates all the states that a system can attain as policies evolve according to some list of requirements. The construction of a metapolicy example requires mapping this diagram to rules, policies and states. We assume that there are two resources, o1 and o2, which can be read by a subject, bob. There are two rules in the initial policy allowing such access, with four possible policies thereafter. We note the assumption that if a resource is not explicitly permitted in a rule, then access is disallowed (closed policy).

```
one sig rule1 extends EACRule {} {
    rid = r1 and subject = bob and resource = o1 and effect = Permit and action = read
}

one sig rule2 extends EACRule {} {
    rid = r2 and subject = bob and resource = o2 and effect = Permit and action = read
}


one sig policy1 extends EACPolicy {} {
    pid = p1 and rules = {(0 → r1) + (1 → r2)}
}

one sig policy2 extends EACPolicy {} {
    pid = p2 and rules = {(0 → r2)}
}

one sig policy3 extends EACPolicy {} {
    pid = p3 and rules = {(0 → r1)}
}

one sig policy4 extends EACPolicy {} {
    pid = p4 and no rules}
}
```

The metapolicy declaration below combines the definitions above to capture the graphical representation of Figure 4.2.

```
one sig imagesrmp extends MetaPolicy {} {
  mid = mpid1
  initialstate = s1
  rule = {(r1 → rule1) + (r2 → rule2)}
  policy = {(p1 → policy1) + (p2 → policy2) + (p3 → policy3) + (p4 → policy4)}
  states = {(s1 → p1) + (s2 → p2) + (s3 → p3) + (s4 → p4) + (s5 → p1)}
  transitions = {(s1 → ((0 → ps1) + (1 → ps2))) + (s2 → (0 → ps3)) + (s3 → (0 → ps4)) +
                 (s4 → (0 → ps5))}
}

one sig ps1 extends Priority {} { mpc = mc1 and state = s2 }
one sig ps2 extends Priority {} { mpc = mc2 and state = s3 }
one sig ps3 extends Priority {} { mpc = mc2 and state = s4 }
one sig ps4 extends Priority {} { mpc = mc1 and state = s4 }
one sig ps5 extends Priority {} { mpc = mc3 and state = s5 }
```

POLICY WRITER In collaboration with data source owners, a policy writer can capture this example using a tool that can generate the state space given the class of requirements and number of resources. The tool produces this formal representation matching the state space.

In the next section, we extend our abstraction to incorporate the use of an audit data source that aids in the evolution of state.

4.5 Audit Data

In any access control system, audit logs record several types of events. However, these systems typically lack the ability to use that audit data intelligently in making access decisions. Utilising such information in access control models has the potential to, for example, allow a system to monitor the number of times a personal website profile has been accessed.

In EAC, we concentrate on recording two types of events in our audit logs. Internal events are those events which are directly observed by the access control mechanism. For example, when subjects attempt to access resources that are allowed by the current policy, the access control mechanism will log these access attempts. In our model, we can abstract away from details of this access and consider internal events consisting of just resource accesses.

External events are those events which occur outside the control of the access control system, but are still of interest to our audit data source. As one can imagine, these events can originate from a variety of sources, ranging from CPU load to network bandwidth. For now, it suffices to capture these external events in terms of a basic signature. We can thus define an Event as a super-type of external and internal events.

```
abstract sig Event {}

abstract sig Resource extends Event {}
abstract sig External extends Event {}
```

This now allows us to specify our audit data source, ADDB, as an injective sequence of Event atoms representing a timeline of observed events in the system.

```
sig ADDB {
    data: seq Event
} {
    !data.hasDups
}
```

4.6 Conditional Events

In EAC, states evolve in response to changes in the system environment. Such changes can include a resource access, a lapse in time, or even a decrease in network bandwidth. For example, whenever a subject accesses a resource, an event is logged which is then used to determine if the current state needs to be evolved. Such state changes bring about the automatic deployment of new policies that allow or deny access to resources accordingly.

Conditional events are non-empty "trigger" sequences of events, Trigger, that signal an evolution of system state. We associate an MPConditionID atom with a set of "trigger" sequences using the MPCondition signature. The set of "trigger" sequences can not be empty and each "trigger" sequence itself can not be empty. The condition function uses an MPConditionID atom to identify a set of "trigger" sequences that can cause a change of state.

```
abstract sig Trigger {
    data: seq Event
}

abstract sig MPCondition {
    mpid: MPConditionID,
    trigger: set Trigger
} {
    all k: trigger | #k.data != 0
    trigger != none
}

fun condition (c: MPConditionID) : set Trigger {
    mpid.c.trigger
}
```

EXAMPLE 4.3. A data source requirement mandates that a user can not access a resource, o1, more than three times. This can be represented by the following "trigger" sequence: ⟨o1,o1,o1⟩. Once this *pattern* is observed in ADDB, then we signal a change of state as the resource has been accessed three times. It is important to note that this "trigger" sequence can represent the data source requirement.

We can use the structures defined so far to construct an example metapolicy. The next section presents a framework of core functions that can be used to automatically evolve policies based on an example metapolicy and data recorded in the audit data source.

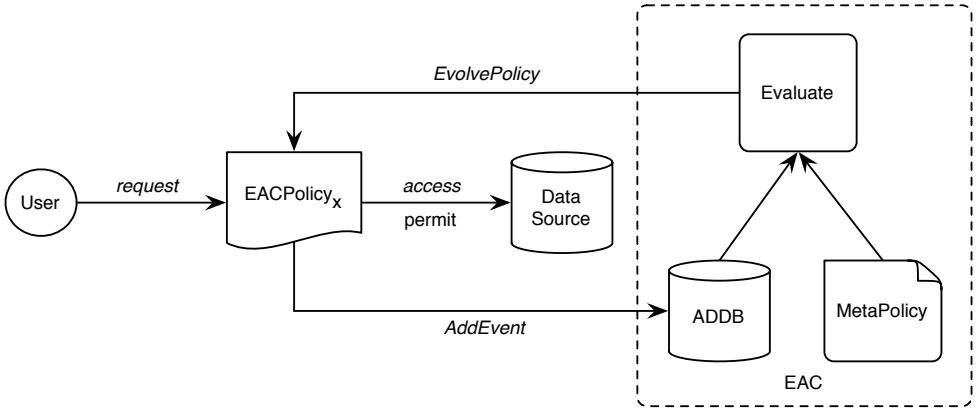


Figure 4.3: A reactive policy evolution

4.7 Evolving the Policy

Now that rules, policies, metapolicies, audit data sources and conditional events have been specified, this section serves to describe a framework of core EAC functions that can evolve policies based on observed events according to a metapolicy example. As shown in Figure 4.3, our approach establishes a constant mode of evolution within the system—once an event is observed, the system determines if an evolution of policy is necessary.

This is accomplished using our state-evolving function, evaluateMPC. This core function checks if the current sequence of accesses contains any "trigger" sequences. The function produces a True or False value each time an event is logged, which is an indication for an evolution.

```
pred evaluateMPC (c: MPConditionID, a: ADDB) {
    some t: condition[c] | subsequence[t.data, a.data]
}
```

The second function, subsequence, is recursive and used to determine if a "trigger" sequence is a contiguous or non-contiguous subsequence of the current sequence of accesses. We have modified the longest common subsequence (LCS) algorithm of [129]. Alloy can not model recursive operations so it is necessary to manually unroll this function a certain number of times as shown below. This limits the length of ADDB but we argue that this is a reasonable restriction since our EAC models are finitely small.

```
pred subsequence (t: seq Resource, a: seq Resource) {
        #a = 4 implies subsequence4 [t, a]
    else
        #a = 3 implies subsequence3 [t, a]
    else
        #a = 2 implies subsequence2 [t, a]
    else
        #a = 1 implies subsequence1 [t, a]
    else
        #a = 0 implies subsequence0 [t, a]
}
```

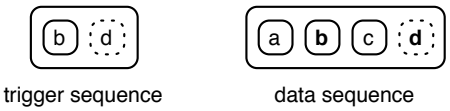


Figure 4.4: The last access compared in both sequences

```
pred subsequence4 (t: seq Resource, a: seq Resource) {
    #t != 0 implies
        (t.last = a.last implies subsequence3 [t.butlast, a.butlast]
        else
        subsequence3 [t, a.butlast])
}

pred subsequence3 (t: seq Resource, a: seq Resource) {
    #t != 0 implies
        (t.last = a.last implies subsequence2 [t.butlast, a.butlast]
        else
        subsequence2 [t, a.butlast])
}

pred subsequence2 (t: seq Resource, a: seq Resource) {
    #t != 0 implies
        (t.last = a.last implies subsequence1 [t.butlast, a.butlast]
        else
        subsequence1 [t, a.butlast])
}

pred subsequence1 (t: seq Resource, a: seq Resource) {
    #t != 0 implies
        (t.last = a.last implies subsequence0 [t.butlast, a.butlast]
        else
        subsequence0 [t, a.butlast])
}

pred subsequence0 (t: seq Resource, a: seq Resource) {
    #t = 0
}
```

EXAMPLE 4.4. Suppose we have a "trigger" sequence ⟨b,d⟩ and that the current sequence of data accesses is ⟨a,b,c,d⟩. Each time an event is logged, this function is executed—particularly, when the d resource has been logged, the function is able to determine that ⟨b,d⟩ is indeed a subsequence of ⟨a,b,c,d⟩. As shown in Figure 4.4, this function works by comparing the last entry of the "trigger" and data sequences, and recursively shortening either sequence accordingly. If the last entries match, then the "trigger" sequence is shortened; otherwise the data sequence is shortened. If the data sequence has been shortened until its size is zero, then the input "trigger" sequence has not been found. Instead, if the input "trigger" sequence has been shortened until its size is zero, then it forms a contiguous or non-contiguous subsequence of the data sequence. The function returns True, indicating a state evolution.

A typical system employing EAC can now be encompassed in the following definition. This signature captures the system state during evolution.

```
sig System {
    addb: ADDB,
    metapolicy: MetaPolicy,
    current: StateID
} {
    current in dom[metapolicy.states]
}
```

When a metapolicy example is constructed and loaded into the framework, the observation predicate selects an event (internal or external) as input to the addEventEvolve predicate.

```
pred observation (s, s': System) {
    let p = getPolicy [s.metapolicy, s.current] |
        some e: (accessibleResources[s.metapolicy, p] + External) |
            addEventEvolve[e, s, s']
}
```

The addEventEvolve predicate is a core function in our abstraction. When an event is added, the evaluateMPC function ascertains if the current sequence of accesses contains a "trigger" sequence. The associated MPConditionID with the highest priority in the transitions sequence of the metapolicy example is then used to select the next state. We recall that an MPConditionID atom and a next state StateID atom are both packaged in the first and second fields of Priority atoms. The current state is then set to that StateID, meaning that an automatic switch to another policy has occurred.

```
pred addEventEvolve (e: Event, s, s': System) {
    s'.addb.data = s.addb.data.add[e]
    s'.metapolicy = s.metapolicy
    one c: MPConditionID, i: dom[s.metapolicy.transitions[s.current]] |
      (evaluateMPC [c, s'.addb] and
          c = first[s.metapolicy.transitions[s.current][i]] and
              (no j: seq/Int | 1 <= j and j <= (i - 1) and
                  evaluateMPC [first[s.metapolicy.transitions[s.current][j]], s'.addb]) and
                      s'.current = second[s.metapolicy.transitions[s.current][i]])
}
```

The next four operations are helper functions that support evolution in our model. The first returns the policy that is mapped to a StateID from our metapolicy states function.

```
fun getPolicy (m: MetaPolicy, s: StateID) : EACPolicy {
    pid.(m.states[s])
}
```

The second function returns the resources that a user can access for that policy.

```
fun accessibleResources (m: MetaPolicy, p: Policy) : set Resource {
    (m.rule[p.rules.elems]).resource
}
```

Our last two helper functions return the first element of a tuple and the second element of a tuple respectively. We use these functions to reference the first and second fields of Priority tuples.

```
fun first (p: Priority) : MPConditionID {
    p.mpc
}

fun second (p: Priority) : StateID {
    p.state
}
```

Our framework uses the *Trace* pattern [130], and facilitates investigation of metapolicy examples. This pattern orders on an atom by constructing a linked list using the util/ordering library in Alloy, which supports the ability to reference the next, previous, first and last state of that atom. In our framework, we order on System atoms as its signature captures the system state during evolution. We then define the following fact, which chains System atoms one after the other as evolution proceeds (via the observation predicate). An example solution is found when all our framework constraints are satisfied to form a trace—all the example solutions found are combined to form the state space of that metapolicy instance.

```
open util/ordering[System]

fact transitions {
    init[first[]]
    all s: System − last[] | let s' = next[s] | observation [s, s']
}
```

This now brings us to the end of our EAC abstraction. At this stage, we can construct small examples of metapolicies, which can then be loaded into this framework of functions so that its state space can be analysed. In the next section, we identify some desirable properties that metapolicies should possess and discuss how they can be checked.

4.8 MetaPolicy Properties

In this section, we discuss how metapolicies can be checked for a number of "healthiness" properties. These are properties of the metapolicy state space, and include the expectation of definite next states (*determinism*), the reachability of states (*connectedness*), and the need to restrict resources for a particular category of requirements as evolution progresses (*restriction*). Such properties can enforce correct behaviour concerning the evolution of policies and, thus, motivate the construction of reliable metapolicies.

We note that we can only check for these properties, not prove. In Alloy, checking assertions means that the Alloy Analyzer tool seeks to refute the assertion claim instead of prove the claim. The assertion is checked against all the possible assignments of variables for that scope—if the assertion does not hold for one case, that case is described as a counterexample, and, in this context, means that the metapolicy does not possess that

property. Our checks can only provide assurance that a metapolicy possesses some property as a counterexample can exist in a greater scope (but such a situation is unlikely).

4.8.1 Determinism

Deterministic metapolicies are driven by the needs of implementation, and it is important that metapolicies produce the same results every time, provided that the inputs and starting state are the same. Non-determinism reflects unpredictability, which is an unattractive proposition—especially for an autonomic, security-critical system. We can check for determinism in metapolicies using the following assertion.

```
assert determinism {
    all s: System | {
        (all i: dom[s.metapolicy.transitions] |
            all disj j, k: dom[s.metapolicy.transitions[i]] |
                first[s.metapolicy.transitions[i][j]] != first[s.metapolicy.transitions[i][k]])
                }
}
```

This assertion ascertains that in the sequence of Priority atoms associated with a starting state, the first field of each Priority atom must be a unique MPConditionID. This means that for every starting state, the transitions to a next state must each be triggered by a distinct conditional event. Even though the addEventEvolve operation already selects only one Priority atom from the sequence, checks for this property reinforces our notion of determinism.

4.8.2 Connectedness

The connected property might be characterised as the fact that in the evolution of policies, all states described in the metapolicy are reachable from the initial state. Reachability is advantageous here because we want to eliminate the presence of "orphan" states in the graph. This is to ensure we do not waste precious resources in considering unreachable states. Therefore, it is important that all states are connected to that initial state.

Alloy uses the transitive closure of a relation to represent reachability. The transitive closure of a binary relation, r, is the smallest relation that contains r and is transitive. The relation represents the paths that are one step long, its square the paths that are two paths long, and so on. The closure relates one atom to another when they are connected by a path of any length (except for zero) [109]. In our model, the transitions relation of the MetaPolicy signature is a relation that is of arity 4, and so calculating the transitive closure is difficult. Consequently, we can check for this property in metapolicy examples with the following assertion.

```
assert connectedness {
    all s: System | {
        (dom[s.metapolicy.states] − s.metapolicy.initial) in ran[s.metapolicy.transitions].state
        }
}
```

The assertion checks that all states, except the initial state, is a subset of or equal to the next states that appear in the range of the transitions function. We recall that the range of the transitions relation is a sequence of Priority atoms—a next state is then the second element of these ordered pairs. In essence, this assertion provides assurance that all states are connected to the initial state as described in the metapolicy.

4.8.3 Restriction

Refinement, in the general case, is an incremental process which converts specification to implementation. In program refinement, we start with an abstract specification of a system, and continually transform this into a concrete program executable via a sequence of correctness preserving transformations [131, 132]. While implementation is outside the scope of this thesis, we keep the general principle of refinement in mind when reasoning about this property.

Restriction is a property of metapolicies in which successive policies, during evolution, contain rules allowing access to a smaller set of resources. The property is loosely related to the refinement process described above from the perspective of a domain reduction such that the set of accessible resources in any state is a proper subset of the preceding state—just as the level of uncertainty diminishes after a sequence of refinement steps [131].

Checking metapolicies for the restriction property may be appropriate (but not necessary) if we want to provide assurance that the next subsequent policy is, according to some classes of requirements, a restriction or subset of the current policy. This property can be checked with the following assertion.

```
assert restriction {
  all s: System − last[] | let s' = next[s] | {
      observation [s, s'] implies
          (accessibleResources [s'.metapolicy, (getPolicy [s'.metapolicy, s'.current])] in
              accessibleResources [s.metapolicy, (getPolicy [s.metapolicy, s.current])])
              }
}
```

This assertion ensures that the set of accessible resources in the next state is a subset of the set of accessible resources in the previous state. This check also provides a security test for policy writers who may be interested in knowing if successive policies, during evolution, increase the restriction of access to resources.

POLICY WRITER Having constructed the formal representation of a metapolicy and its associated structures, the policy writer can use a suitable tool to run analysis for these properties. The tool can indicate whether or not these properties hold by reporting any counterexamples found or lack thereof, allowing the policy writer to investigate further and make appropriate amendments if necessary.

In the next section, we discuss how our framework of functions can also verify the correctness of policies.

4.9 Policy Properties

We use our EAC abstraction to also verify the notion of *consistency*—while it is desirable that a metapolicy be deterministic and connected (and in some cases, restricted), its policies must also be consistent with the metapolicy requirements. Consistency describes the idea that in all states, some invariant must hold—these invariants are expressions of the data source requirements captured in a metapolicy. Consistency properties pertain to particular instantiations of our EAC abstraction as requirements vary from model to model. For example, we recall that the images example has four data source requirements, captured in a metapolicy, that are enforced on users of the system. These data source requirements are our invariants that must be maintained in every state of the evolution.

In Example 4.3, we demonstrated how such requirements create data patterns in the audit data source—this is the key idea behind our consistency checks—we use such data patterns in the audit data source to verify that policies meet the requirements captured in the metapolicy. Verification of policy properties thus involves checking that in each state, the relationship between the current policy and the sequence of accesses logged in the audit data source, is consistent with the requirements captured in the metapolicy. Such checks ensure that there is no deviation from intended behaviour when policies evolve.

As noted earlier, since policy properties vary from model to model, in Alloy, we can use assertion patterns to check for consistency. When we apply these assertion patterns to an example, variables must be tailored to capture that requirement.

4.9.1 Binary

We can check consistency here with the following assertion pattern. We remember that an instance of the *binary* class of requirements dictates that a resource can only be accessed once. Therefore, if a resource is accessible via the current policy, then it can not also be recorded in the audit data source (implying that the resource has been accessed).

```
assert binary {
    all s: System | {
        all r: accessibleResources[s.metapolicy, (getPolicy[s.metapolicy, s.current])] |
            r !in s.addb.data.elems
            }
}
```

4.9.2 Counting

With respect to an instance of the *counting* class of requirements, only a certain number, c, of resources can be accessed, which means that the number of entries in the audit data source must be less than or equal to that number c. Once this number has been attained, then the current policy must deny access to all resources. We can check consistency here using the following assertion pattern.

```
assert counting {
    all s: System | {
        #s.addb.data = c implies
            no accessibleResources[s.metapolicy, (getPolicy[s.metapolicy, s.current])]
            }
}
```

4.9.3 Subscription

Access to a specific worth of data indicates that the sum of the associated value of the elements already in the audit data source and the value of the resources in the current policy must be less than or equal to that worth, *w*, in a *subscription* based requirement. The sum keyword in Alloy represents the integer value obtained by summing the values of the elements logged in ADDB. The elements are associated with values as specified in domain specific relations in Domain.

```
assert subscription {
    all s: System | {
        all r: accessibleResources[s.metapolicy, (getPolicy[s.metapolicy, s.current])] |
            ((sum n: s.addb.data.elems | Domain.values[n])) + Domain.values[r] =< w
            }
}
```

4.9.4 Compartment

The ability to only access resources in an isolated set suggests that in the audit data source, only elements from that set should be present. This also means that, in the current policy, each accessible resource should also belong to that same set. Accordingly, we can check that all elements logged in ADDB, and all resources in the current policy belong to one set, A, or the other, B. Elements are associated with groups via domain specific relations in Domain.

```
assert compartment {
    all s: System | {
        all r: accessibleResources[s.metapolicy, (getPolicy[s.metapolicy, s.current])] |
            ((all n: s.addb.data.elems | Domain.groups[n] = A) and Domain.groups[r] = A)
            or
            ((all n: s.addb.data.elems | Domain.groups[n] = B) and Domain.groups[r] = B)
            }
}
```

4.9.5 Reject

Instances of this requirement involve the user attempting to access restricted resources in which he or she is denied all access after a number of rejections. Because our framework has been modelled based on only successful access attempts (EAC assumes a closed approach), then by symmetry we can extrapolate that such a requirement can be checked if we

assume that our framework can only handle unsuccessful attempts, then after *n* unsuccessful attempts will the current policy refuse all accesses (similar to the *counting* requirement).

```
assert reject {
    all s: System | {
        #s.addb.data = n implies
            no accessibleResources[s.metapolicy, (getPolicy[s.metapolicy, s.current])]
            }
}
```

4.9.6 Emergency

An authorised user (like an administrator) can choose to override access rights for any user. In our framework, we can model the ability to override access as an external event—checking here will then involve ensuring that when the external event is triggered, the current policy is consistent with requirements. The nature of this requirement is very similar to that of *delegation*—the ability to grant permission to another user. For example, if an attending physician needs immediate access to a file of an individual who is not a patient of that physician, then this implies that once a trigger is received, the policy evolves to allow access. Using the following assertion pattern, we can check that if the last entry in the audit data source is that trigger, then the previous policy has no accessible resources while the current policy now has accessible resources.

```
assert emergency {
  all s: System − last[] | let s' = next[s] | {
      observation [s, s'] implies
          (s'.addb.data.last = emergency implies
              no accessibleResources[s.metapolicy, (getPolicy[s.metapolicy, s.current])]
              and
              #accessibleResources[s.metapolicy, (getPolicy[s.metapolicy, s'.current])] >= 0)
              }
}
```

4.9.7 Liveness

In this requirement, we prevent further access after a period of inactivity. Time events are external to our access control model. When a set length of time has expired, an external event is triggered which signals that the policy should be closed to further access. In Alloy, we can perform this check with this assertion pattern.

```
assert liveness {
  all s: System − last[] | let s' = next[s] | {
      observation [s, s'] implies
            (s'.addb.data.last = liveness implies
                no accessibleResources[s.metapolicy, (getPolicy[s.metapolicy, s'.current])])
                }
}
```

4.9.8 Period

The *period* requirement involves the triggering of an external event once a set time period has expired to re-allow access to resources. We can check consistency here using this pattern: we compare the last access to the current policy; if the last access is a period external event, then the previous policy must have denied access to all resources. The current policy then allows access to some resource.

```
assert period {
  all s: System − last[] | let s' = next[s] | {
      observation [s, s'] implies
          (s'.addb.data.last = period implies
              no accessibleResources[s.metapolicy, (getPolicy[s.metapolicy, s.current])]
              and
              #accessibleResources[s.metapolicy, (getPolicy[s.metapolicy, s'.current])] >= 0)
              }
}
```

4.9.9 System

For this requirement, we employ a similar assertion pattern and check that once an external event occurs, then the policy evolves—for example, if the network capacity has been exceeded, an external event, network_exceed is triggered. When this occurs, then no more accesses can occur, and the current policy evolves to block all access. In Alloy, we can check for this consistency property with the following assertion pattern.

```
assert system {
  all s: System − last[] | let s' = next[s] | {
       observation [s, s'] implies
            (s'.addb.data.last = network_exceed implies
                no accessibleResources[s.metapolicy, (getPolicy[s.metapolicy, s'.current])])
                }
}
```

4.9.10 Failure Cases

The assertion patterns just presented demonstrate how we can analyse metapolicies and verify policies. It is possible that the construction of a metapolicy will lead to an analysis or verification failure—this identifies potential issues in that metapolicy example. A failure case produces counterexamples that a user can further explore using our model.

For example, in one evolution trace of policies directed by a metapolicy, the restriction property can fail if one of the high-level rules is an instantiation of the *period* class of requirements. In this requirement, the number of accessible resources decreases and increases as evolution proceeds—this violates the restriction check producing the following typical result:

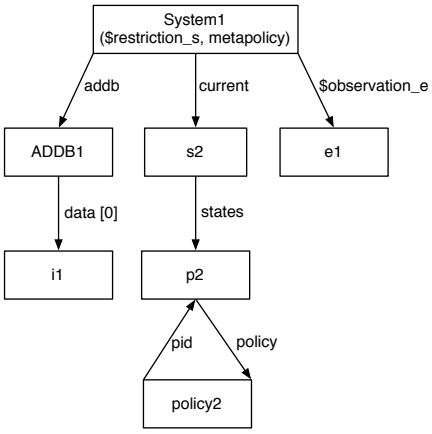


Figure 4.5: A failure case: System 1

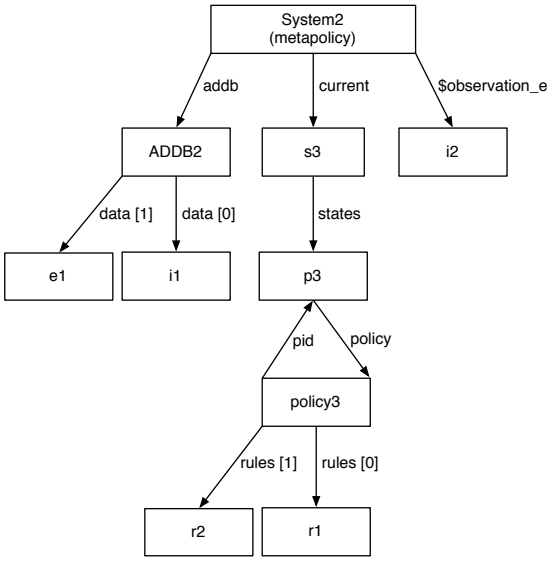


Figure 4.6: A failure case: System 2

Executing "Check restriction **for** 5 **int**, 9 **seq**, **exactly** 5 System, 5 ADDB"
Solver=sat4j Bitwidth=5 MaxSeq=9 SkolemDepth=1 Symmetry=20
205586 vars. 8789 primary vars. 653351 clauses. 2858049ms.
Counterexample found. Assertion is invalid. 462ms.

The Visualisation tool of Alloy can also be used to investigate failure cases. This tool illustrates the assigned variables of the counterexample trace for the restriction assertion as shown in the figures above. Figure 4.5 shows that in some System state, s2, the current policy as mapped by the states relation is policy2 with no rules. The next observation to take place is the external event, e1, while the internal event, i1 has already been recorded. In Figure 4.6, System evolves to another state, s3, triggered by e1—in this state, the current policy has two rules, r1 and r2. This violates the restriction assertion as the number of accessible resources in the next state (two) is larger than the number of accessible resources in the previous state (zero). We also note that the next observation to take place is the internal event, i2, while the events i1 and e1 have already been logged. The "$" identifies a "skolem" in our model, and is used to prevent conflicts with actual Alloy signatures, fields or functions that are top-level expressions. Since "s" and "e" are quantified variables inside restriction and observation respectively, they are not visible at the top level. Using the Visualisation tool, particular signatures can be projected so that they are removed from the diagram making it easier to identify issues.

POLICY WRITER Once the formal representation of the metapolicy example is generated, the policy writer can now apply the appropriate checking clauses to match the class of requirements for that metapolicy. The tool can, perhaps, translate each logical statement of the metapolicy into the corresponding assertion, and then verify that all constructed policies meet these requirements. If the tool reports counterexamples during the verification process, the policy writer can then perform the necessary amendments. Otherwise, having analysed the metapolicy, and verified its associated policies, they can now be considered fit for deployment, as we explore in the next section.

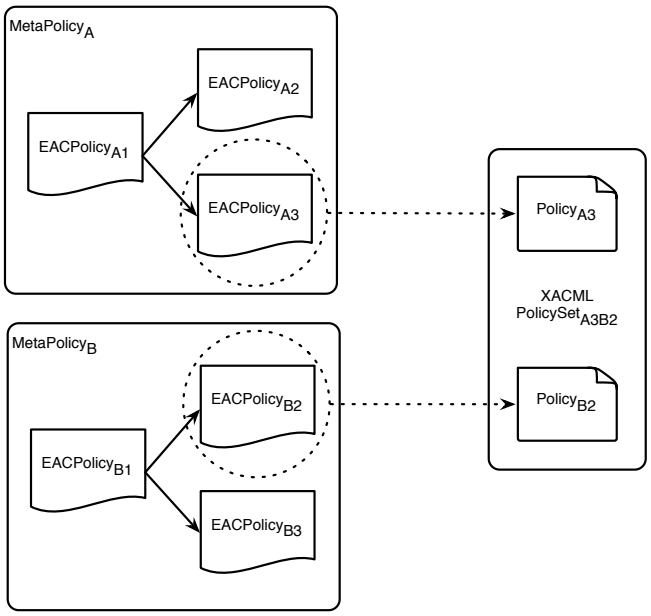


Figure 4.7: The translation process

4.10 Transformation

Having demonstrated how metapolicies and policies can be constructed, analysed and verified, we seek to transform our version of an EAC policy to machine-readable XACML so that it can be deployed in real access control systems whenever a change of state occurs.

4.10.1 Why XACML?

We choose to translate EAC policies to XACML for several reasons. First, XACML is a standard that has been reviewed by a wide community of experts and users, minimising the need to design a bespoke language. Second, it offers a powerful framework for creating policies while providing an expressive language that supports a diverse collection of data types, functions and combining algorithms that can be easily extended. Third, XACML is sufficiently generic to be deployed in any environment, making policy management easier as one policy can be constructed to be used by several applications. Fourth, XACML can be utilised in distributed contexts which means that a policy can refer to other policies in remote locations, again making policy management simple as groups of authorised individuals can manage separate areas of the overall policy (XACML is capable of combining results from different policies into a single decision). Fifth, XACML has already been utilised in the GIMI project for fine-grained access control, which provides us with a test-bed for our research.

4.10.2 Translation

A high-level view of the translation process is illustrated in Figure 4.7 above. We see that each EAC policy associated with a metapolicy is translated to a XACML policy, which then makes up a policy set. A policy set can comprise several policies—in our model, one user

corresponds to one policy in a sole root policy set, thereby supporting multiple users. The equivalent machine-readable XACML that is produced can be deployed in access control systems. In effect, the translation allows the deployment of EAC policies in real systems putting our theory into practice. As states evolve, the XACML policy set (containing a policy for each user) is re-generated each time.

In this thesis, the transformation can be mapped directly from Alloy to XACML in the following manner. An example rule in our model can be modelled as rule1 below.

```
one sig rule1 extends EACRule {} {
    rid = r1 and subject = A and resource = o1 and effect = Permit and action = read
}
```

In XACML, an EAC rule can be implemented as the following. We note the use of *targets*, which are structures that represent a tuple of values for a subject, resource, action, and an environment. If all the values of a *target* match that of an access *request*, then its associated policy set, policy or rule is applied to that request. In line 1 we observe the rid and effect of our EAC rule. Line 6 maps our subject, whereas line 16 maps the resource, and line 26 maps our action.

```
1  <Rule RuleId="r1" Effect="Permit">
2    <Target>
3      <Subjects>
4        <Subject>
5          <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
6            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">A</AttributeValue>
7            <SubjectAttributeDesignator
8              DataType="http://www.w3.org/2001/XMLSchema#string"
9              AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
10         </SubjectMatch>
11       </Subject>
12     </Subjects>
13     <Resources>
14       <Resource>
15         <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
16           <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">o1</AttributeValue>
17           <ResourceAttributeDesignator
18             DataType="http://www.w3.org/2001/XMLSchema#string"
19             AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
20         </ResourceMatch>
21       </Resource>
22     </Resources>
23     <Actions>
24       <Action>
25         <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
26           <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">read</AttributeValue>
27           <ActionAttributeDesignator
28             DataType="http://www.w3.org/2001/XMLSchema#string"
29             AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
30         </ActionMatch>
31       </Action>
32     </Actions>
33   </Target>
34 </Rule>
```

EAC rules are compiled into an EAC policy as a sequence of rule identifiers concerning one user as defined in an example below.

```
one sig policy1 extends EACPolicy {} {
    pid = p1 and rules = {(0 → r1)}
}
```

Our policy, policy1, with rule1 defined above can be mapped to the equivalent XACML

policy below. In line 1, we note the pid of this policy. Also on this line, we notice the use of the deny-overrides rule-combining algorithm, which returns a *Deny* result if no rule is successfully evaluated, which is in accordance with our closed approach to access control. The associated target with this policy filters requests pertaining to only subject A (line 6). The sequence of rules concerning this subject is then listed starting at line 21.

```
1  <Policy PolicyId="p1" RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:deny-overrides">
2    <Target>
3      <Subjects>
4        <Subject>
5          <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
6            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">A</AttributeValue>
7            <SubjectAttributeDesignator
8              DataType="http://www.w3.org/2001/XMLSchema#string"
9              AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
10         </SubjectMatch>
11       </Subject>
12     </Subjects>
13     <Resources>
14       <AnyResource/>
15     </Resources>
16     <Actions>
17       <AnyAction/>
18     </Actions>
19   </Target>
20
21   <Rule RuleId="r1" Effect="Permit">
22     <Target>
23       <Subjects>
24         <Subject>
25           <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
26             <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">A</AttributeValue>
27             <SubjectAttributeDesignator
28               DataType="http://www.w3.org/2001/XMLSchema#string"
29               AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
30           </SubjectMatch>
31         </Subject>
32       </Subjects>
33       <Resources>
34         <Resource>
35           <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
36             <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">o1</AttributeValue>
37             <ResourceAttributeDesignator
38               DataType="http://www.w3.org/2001/XMLSchema#string"
39               AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
40           </ResourceMatch>
41         </Resource>
42       </Resources>
43       <Actions>
44         <Action>
45           <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
46             <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">read</AttributeValue>
47             <ActionAttributeDesignator
48               DataType="http://www.w3.org/2001/XMLSchema#string"
49               AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
50           </ActionMatch>
51         </Action>
52       </Actions>
53     </Target>
54   </Rule>
55 </Policy>
```

The final XACML policy set that can be deployed in real systems is listed below. The target associated with this policy set (lines 7 to 17) allows all access requests to be matched. Each of the policies listed in a policy set is associated with one user. In our example below, the only policy will be matched against requests pertaining to user A.

As indicated in our final chapter, future work will prove and automate this transformation. In the next section, we use the images example of Chapter 3 to exemplify our end-to-end process of constructing, analysing and deploying EAC metapolicies and policies.

```
<?xml version="1.0" encoding="UTF-8"?>
<PolicySet xmlns="urn:oasis:names:tc:xacml:1.0:policy"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:oasis:names:tc:xacml:1.0:policy cs-xacml-schema-policy-01.xsd"
  PolicyCombiningAlgId="urn:oasis:names:tc:xacml:1.0:policy-combining-algorithm:deny-overrides"
  PolicySetId="ps1">
  <Target>
    <Subjects>
      <AnySubject/>
    </Subjects>
    <Resources>
      <AnyResource/>
    </Resources>
    <Actions>
      <AnyAction/>
    </Actions>
  </Target>

  <Policy PolicyId="p1" RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:deny-overrides">
    <Target>
      <Subjects>
        <Subject>
          <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">A</AttributeValue>
            <SubjectAttributeDesignator
              DataType="http://www.w3.org/2001/XMLSchema#string"
              AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
          </SubjectMatch>
        </Subject>
      </Subjects>
      <Resources>
        <AnyResource/>
      </Resources>
      <Actions>
        <AnyAction/>
      </Actions>
    </Target>

    <Rule RuleId="r1" Effect="Permit">
      <Target>
        <Subjects>
          <Subject>
            <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
              <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">A</AttributeValue>
              <SubjectAttributeDesignator
                DataType="http://www.w3.org/2001/XMLSchema#string"
                AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
            </SubjectMatch>
          </Subject>
        </Subjects>
        <Resources>
          <Resource>
            <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
              <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">o1</AttributeValue>
              <ResourceAttributeDesignator
                DataType="http://www.w3.org/2001/XMLSchema#string"
                AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
            </ResourceMatch>
          </Resource>
        </Resources>
        <Actions>
          <Action>
            <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
              <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">read</AttributeValue>
              <ActionAttributeDesignator
                DataType="http://www.w3.org/2001/XMLSchema#string"
                AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
            </ActionMatch>
          </Action>
        </Actions>
      </Target>
    </Rule>
  </Policy>
</PolicySet>
```

POLICY WRITER Once the metapolicy and associated policies have been deemed suitable for deployment, the policy writer can use the same tool to do so. In our approach, we have only deployed EAC policies so as to validate the behaviour of metapolicies.

4.11 The Images Example

In this section, we present a small EAC model which uses our abstraction to construct and analyse a metapolicy, and then verify and deploy policies. The images example introduced in Chapter 3 is used to drive this demonstration. We recall that, in this example, we are provided with an online facility which allows users to pay for viewing images. There is an associated pricing model where some images are worth more than others:

Image	Price
i1	2
i2	2
i3	2
i4	4
i5	4
i6	4
i7	6

The data source owner of this online facility imposes the following high-level requirements on viewers of these images.

1. One can view any image at most once.
2. One can not have more than three accesses.
3. One can not view a set of images worth more than a value of 10.
4. One can view either only odd or only even images.

Conceptually, these requirements can be captured using a suitable tool at one (high) level, which decomposes the requirements into a state space at a lower level. The state space generated is illustrated in Figure 4.8 (reproduced from Appendix A.1).

At this lower level, we can now use our EAC abstraction to provide assurance that the evolution of policies meets these requirements. We begin by instantiating our abstraction for this example—the entire Alloy model is listed in Appendix C.8; here we only refer to snippets of the code to aid in our discussion.

4.11.1 Construction

The basic signatures are first defined as extensions of abstract signatures. There are seven resources with 15 states and policies.

```
one sig user extends Subject {}
one sig i1, i2, i3, i4, i5, i6, i7 extends Resource {}
one sig view extends Action {}
one sig empty extends External {}
```

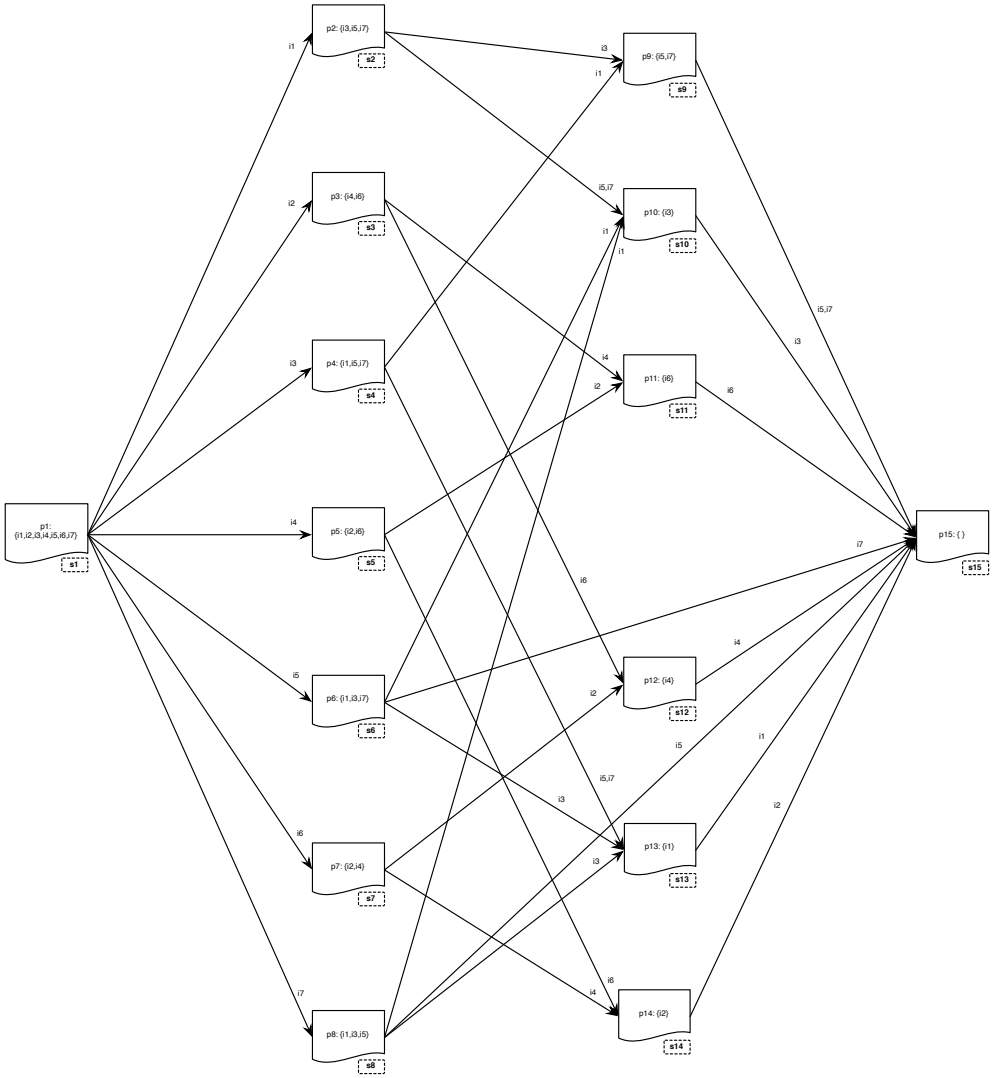


Figure 4.8: The images example state space

one sig mpid1 **extends** MetaPolicyID {}
one sig mc1, mc2, mc3, mc4, mc5, mc6, mc7 **extends** MPConditionID {}
one sig p1, p2, p3, p4, p5, p6, p7 **extends** PolicyID {}
one sig p8, p9, p10, p11, p12, p13, p14, p15 **extends** PolicyID {}
one sig s1, s2, s3, s4, s5, s6, s7 **extends** StateID {}
one sig s8, s9, s10, s11, s12, s13, s14, s15 **extends** StateID {}
one sig r1, r2, r3, r4, r5, r6, r7 **extends** RuleID {}

The metapolicy itself is encoded with the following signature, which represents the state space of Figure 4.8 and calls on other signatures that we explain below.

```
one sig imagesmp extends MetaPolicy {} {
  mid = mpid1
  initial = s1
  rule = {(r1 → rule1) + (r2 → rule2) + (r3 → rule3) + (r4 → rule4) +
          (r5 → rule5) + (r6 → rule6) + (r7 → rule7)}
  policy = {(p1 → policy1) + (p2 → policy2) + (p3 → policy3) + (p4 → policy4) +
            (p5 → policy5) + (p6 → policy6) + (p7 → policy7) + (p8 → policy8) +
```

```
            (p9 → policy9) + (p10 → policy10) + (p11 → policy11) +
            (p12 → policy12) + (p13 → policy13) + (p14 → policy14) +
            (p15 → policy15)}
  states = {(s1 → p1) + (s2 → p2) + (s3 → p3) + (s4 → p4) + (s5 → p5) +
            (s6 → p6) + (s7 → p7) + (s8 → p8) + (s9 → p9) + (s10 → p10) +
            (s11 → p11) + (s12 → p12) + (s13 → p13) + (s14 → p14) +
            (s15 → p15)}
  transitions = {(s1 → ((0 → ps1) + (1 → ps2) + (2 → ps3) + (3 → ps4) +
                        (4 → ps5) + (5 → ps6) + (6 → ps7))) +
                (s2 → ((0 → ps8) + (1 → ps9) + (2 → ps10))) +
                (s3 → ((0 → ps11) + (1 → ps12))) +
                (s4 → ((0 → ps13) + (1 → ps14) + (2 → ps15))) +
                (s5 → ((0 → ps16) + (1 → ps17))) +
                (s6 → ((0 → ps18) + (1 → ps19) + (2 → ps20))) +
                (s7 → ((0 → ps21) + (1 → ps22))) +
                (s8 → ((0 → ps18) + (1 → ps19) + (2 → ps20))) +
                (s9 → ((0 → ps23) + (1 → ps20))) +
                (s10 → (0 → ps24)) +
                (s11 → (0 → ps25)) +
                (s12 → (0 → ps26)) +
                (s13 → (0 → ps27)) +
                (s14 → (0 → ps28))}
}
```

This is an Alloy instance of the MetaPolicy signature that we specified earlier in this chapter. There are 28 Priority pairs in this example metapolicy that resemble the following definition. Priority pairs hold unique pairs of an MPConditionID and StateID atom, and each represents the transition to a next state.

```
one sig ps1 extends Priority {} {mpc = mc1 and state = s2}
```

Seven rules are defined, one for each resource, that allow the ability to view that image. For example, the first rule can be declared in this way.

```
one sig rule1 extends EACRule {} {
    rid = r1 and subject = user and resource = i1 and effect = Permit and action = view
}
```

Policies are defined as sequences of rule identifiers. For instance, the initial policy is a sequence of all seven rules. We observe that our rules and policies here are instances of EACRule and EACPolicy.

```
one sig policy1 extends EACPolicy {} {
    pid = p1 and rules = {(0 → r1) + (1 → r2) + (2 → r3) + (3 → r4) +
                          (4 → r5) + (5 → r6) + (6 → r7)}
}
```

"Trigger" sequences are extensions of the Trigger abstract signature, and are sequences of events. The condition function maps a MPConditionID to a set of "trigger" sequences, and are used to determine when an evolution is necessary. We define seven such trigger sequences for each MPConditionID.

```
one sig trig1 extends Trigger {} {
    data = {(0 → i1)}
}
```

An MPCondition is a conditional event and placeholder for a MPConditionID and a Trigger—we have defined seven signatures extended from MPCondition, which aid in associating a MPConditionID with a set of Trigger sequences.

```
one sig mp1 extends MPCondition {} {
    mpid = mc1 and trigger = {trig1}
}
```

The initial ADDB contains an empty data sequence—as evolution progresses, events are logged in chronological order.

```
one sig imagesaddb extends ADDB {} {
    no data
}
```

The following signature definitions are domain-specific. The definitions associate resources with an integer value (worth), and a group. These definitions are specific to this example and changes from (metapolicy) instance to instance. We see that each image has an increasing worth and belongs to either of two groups.

```
abstract sig Class {}

one sig Odd, Even extends Class {}

one sig Domain {
    values: Resource → Int,
    groups: Resource → Class
} {
    values = {i1 → 2 + i2 → 2 + i3 → 2 + i4 → 4 + i5 → 4 + i6 → 4 + i7 → 6}
    groups = {i1 → Odd + i2 → Even + i3 → Odd + i4 → Even + i5 → Odd +
              i6 → Even + i7 → Odd}
}
```

At this point, we have constructed our metapolicy instance for this example, and can now load this model into our EAC framework of functions to check for properties and verify consistency.

4.11.2 Analysis

Once the model is loaded, we can check assertions with the following command, setting the scope accordingly. The scope tells the Analyzer how many System atoms it should use in the analysis as each System atom represents the system state as evolution progresses.

```
check assertion for exactly 4 System, 4 ADDB, 4 int, 7 seq
```

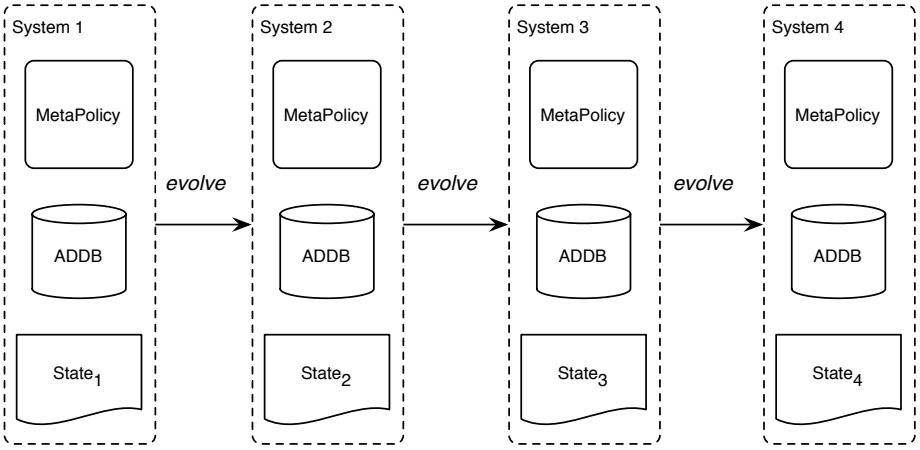


Figure 4.9: A visualisation of the scope

The Analyzer is instructed to use four System and ADDB atoms (in our abstraction, these two structures, as well as the current state, are modified during evolution—the metapolicy does not change), which implies three iterations of evolutions as shown in Figure 4.9. This number of evolutions allows us to capture the entire state space of Figure 4.8. We also set the maximum integer bitwidth to 4, since this metapolicy instance uses integers within the range -8 to +7. The sequence length is set to 7 since the initial policy is a sequence of seven rule identifiers (these are Alloy specific settings).

MetaPolicy Properties

In this section, we use the assertions of Section 4.8 to check for desirable properties in our small metapolicy example. Such properties, as discussed, can enforce the correct evolution of policies, providing assurance that the metapolicy behaves as intended. Running the first assertion to check for *determinism* in our model produces the following results.

```
assert determinism {
    all s: System | {
        (all i: dom[s.metapolicy.transitions] |
            all disj j, k: dom[s.metapolicy.transitions[i]] |
                first[s.metapolicy.transitions[i][j]] != first[s.metapolicy.transitions[i][k]])
                }
}

check determinism for exactly 4 System, 4 ADDB, 4 int, 7 seq

Executing "Check determinism for 4 int, 7 seq, exactly 4 System, 4 ADDB"
   Solver=sat4j Bitwidth=4 MaxSeq=7 SkolemDepth=1 Symmetry=20
   115639 vars. 6138 primary vars. 336259 clauses. 308100ms.
   No counterexample found. Assertion may be valid. 173ms.
```

We can make additional checks for determinism by ensuring that all priority pairs are unique for each next state.

```
assert uniquepriority {
    all s: System | {
        all disj i,j: ran[s.metapolicy.transitions[s.current]] | i != j
        }
}

check uniquepriority for exactly 4 System, 4 ADDB, 4 int, 7 seq

Executing "Check uniquepriority for 4 int, 7 seq, exactly 4 System, 4 ADDB"
   Solver=sat4j Bitwidth=4 MaxSeq=7 SkolemDepth=1 Symmetry=20
   114930 vars. 6165 primary vars. 324578 clauses. 310230ms.
   No counterexample found. Assertion may be valid. 701ms.
```

The result of finding no counterexamples means that none of the constraints of the model were violated as the metapolicy directed evolutions—we conclude that this metapolicy instance possesses the determinism property for that particular scope.

Next, we make checks for the *connectedness* property. Results show that this metapolicy instance is connected.

```
assert connectedness {
    all s: System | {
        (dom[s.metapolicy.states] − s.metapolicy.initial) in ran[s.metapolicy.transitions].state
        }
}

check connectedness for exactly 4 System, 4 ADDB, 4 int, 7 seq

Executing "Check connectedness for 4 int, 7 seq, exactly 4 System, 4 ADDB"
   Solver=sat4j Bitwidth=4 MaxSeq=7 SkolemDepth=1 Symmetry=20
   112321 vars. 6109 primary vars. 319366 clauses. 316892ms.
   No counterexample found. Assertion may be valid. 107ms.
```

Our *restriction* property is also met by this example metapolicy instance.

```
assert restriction {
  all s: System − last[] | let s' = next[s] | {
      observation [s, s'] implies
          (accessibleResources [s'.metapolicy, (getPolicy [s'.metapolicy, s'.current])] in
              accessibleResources [s.metapolicy, (getPolicy [s.metapolicy, s.current])])
              }
}

check restriction for exactly 4 System, 4 ADDB, 4 int, 7 seq

Executing "Check restriction for 4 int, 7 seq, exactly 4 System, 4 ADDB"
   Solver=sat4j Bitwidth=4 MaxSeq=7 SkolemDepth=1 Symmetry=20
   111898 vars. 6109 primary vars. 318252 clauses. 313880ms.
   No counterexample found. Assertion may be valid. 233ms.
```

Policy Properties

We progress to the verification of policy properties by using our assertion patterns of Section 4.9. The four requirements of this example can be verified using the *binary*, *counting*, *subscription*, and *compartment* assertion patterns. The last three of these patterns need to be tailored to capture the variables of the requirements. Nevertheless, we would check for *binary* consistency in our example using the following assertion (the first requirement).

```
assert binary {
    all s: System | {
        all r: accessibleResources[s.metapolicy, (getPolicy[s.metapolicy, s.current])] |
            r !in s.addb.data.elems
            }
}

check binary for exactly 4 System, 4 ADDB, 4 int, 7 seq

Executing "Check binary for 4 int, 7 seq, exactly 4 System, 4 ADDB"
   Solver=sat4j Bitwidth=4 MaxSeq=7 SkolemDepth=1 Symmetry=20
   110514 vars. 6116 primary vars. 315795 clauses. 312859ms.
   No counterexample found. Assertion may be valid. 192ms.
```

Next, we need to modify our assertion pattern for the second *counting* requirement of this metapolicy instance—the number of entries logged in ADDB must be equal to three (making c equal to three).

```
assert counting {
    all s: System | {
        #s.addb.data = 3 implies
            no accessibleResources[s.metapolicy, (getPolicy[s.metapolicy, s.current])]
            }
}

check counting for exactly 4 System, 4 ADDB, 4 int, 7 seq

Executing "Check counting for 4 int, 7 seq, exactly 4 System, 4 ADDB"
   Solver=sat4j Bitwidth=4 MaxSeq=7 SkolemDepth=1 Symmetry=20
   110822 vars. 6109 primary vars. 317714 clauses. 315637ms.
   No counterexample found. Assertion may be valid. 273ms.
```

Similarly, the *subscription* requirement can be checked if we set the worth, w, to 10 value points. The bitwidth has been increased to 5 for this check to encompass the integer value of 10 (otherwise the Analyzer throws an error).

```
assert subscription {
    all s: System | {
        all r: accessibleResources[s.metapolicy, (getPolicy[s.metapolicy, s.current])] |
            ((sum n: s.addb.data.elems | Domain.values[n])) + Domain.values[r] =< 10
            }
}
```

check subscription **for exactly** 4 System, 4 ADDB, 5 **int**, 7 **seq**

Executing "Check subscription **for** 5 **int**, 7 **seq**, **exactly** 4 System, 4 ADDB"
Solver=sat4j Bitwidth=5 MaxSeq=7 SkolemDepth=1 Symmetry=20
113996 vars. 6228 primary vars. 330625 clauses. 339914ms.
No counterexample found. Assertion may be valid. 284ms.

The final consistency check for the fourth requirement of this example metapolicy instance can be performed in the following manner. Our resources can belong to either of two conflict classes, Odd or Even. We modify our assertion pattern to account for this.

```
assert compartment {
  all s: System | {
    all r: accessibleResources[s.metapolicy, (getPolicy[s.metapolicy, s.current])] |
        ((all n: s.addb.data.elems | Domain.groups[n] = Odd) and Domain.groups[r] = Odd)
        or
        ((all n: s.addb.data.elems | Domain.groups[n] = Even) and Domain.groups[r] = Even)
        }
}

check compartment for exactly 4 System, 4 ADDB, 4 int, 7 seq
```

Executing "Check compartment **for** 4 **int**, 7 **seq**, **exactly** 4 System, 4 ADDB"
Solver=sat4j Bitwidth=4 MaxSeq=7 SkolemDepth=1 Symmetry=20
110529 vars. 6116 primary vars. 315897 clauses. 310655ms.
No counterexample found. Assertion may be valid. 356ms.

Having applied our property and consistency verification techniques to check an example metapolicy and associated policies, we have provided some assurance that the metapolicy of the images example is deterministic, connected and restricted, and that policies consistently meet its four requirements. A suitable tool, at a higher level, would be able to return these results to the policy writer, signalling that this metapolicy and associated policies can be safely deployed in an access control system configured for its use. However, in this thesis, and most importantly, we have demonstrated how, in general, our EAC framework might be used to analyse metapolicies and verify policies.

4.11.3 Transformation

We know that our metapolicy and associated policies for this example have met our criteria for deployment, and so, in this section, we present how the initial EAC policy can be translated to a machine-readable XACML policy. Each time an evolution occurs, this process is repeated.

The initial policy of the images example has been constructed in Section 4.11.1. This policy references seven rules for the subject user as shown below.

```
one sig policy1 extends EACPolicy {} {
    pid = p1 and rules = {(0 → r1) + (1 → r2) + (2 → r3) + (3 → r4) +
                          (4 → r5) + (5 → r6) + (6 → r7)}
}
```

Using the transformation process of Section 4.10, the equivalent machine-readable XACML policy is now listed.

```
<?xml version="1.0" encoding="UTF-8"?>
<PolicySet xmlns="urn:oasis:names:tc:xacml:1.0:policy"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:oasis:names:tc:xacml:1.0:policy cs-xacml-schema-policy-01.xsd"
  PolicyCombiningAlgId="urn:oasis:names:tc:xacml:1.0:policy-combining-algorithm:deny-overrides"
  PolicySetId="imagesxacml1">
  <Target>
    <Subjects>
      <AnySubject/>
    </Subjects>
    <Resources>
      <AnyResource/>
    </Resources>
    <Actions>
      <AnyAction/>
    </Actions>
  </Target>

  <Policy PolicyId="p1" RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:deny-overrides">
    <Target>
      <Subjects>
        <Subject>
          <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">user</AttributeValue>
            <SubjectAttributeDesignator
              DataType="http://www.w3.org/2001/XMLSchema#string"
              AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
          </SubjectMatch>
        </Subject>
      </Subjects>
      <Resources>
        <AnyResource/>
      </Resources>
      <Actions>
        <AnyAction/>
      </Actions>
    </Target>

    <Rule RuleId="r1" Effect="Permit">
      <Target>
        <Subjects>
          <Subject>
            <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">user</AttributeValue>
              <SubjectAttributeDesignator
                DataType="http://www.w3.org/2001/XMLSchema#string"
                      AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
            </SubjectMatch>
          </Subject>
        </Subjects>
        <Resources>
          <Resource>
            <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
              <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">i1</AttributeValue>
              <ResourceAttributeDesignator
                DataType="http://www.w3.org/2001/XMLSchema#string"
                AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
            </ResourceMatch>
          </Resource>
        </Resources>
        <Actions>
          <Action>
            <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">view</AttributeValue>
              <ActionAttributeDesignator
                DataType="http://www.w3.org/2001/XMLSchema#string"
                AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
            </ActionMatch>
          </Action>
        </Actions>
      </Target>
    </Rule>

    <Rule RuleId="r2" Effect="Permit">
      <Target>
        <Subjects>
          <Subject>
            <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
```

```
79              <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">user</AttributeValue>
80                <SubjectAttributeDesignator
81                  DataType="http://www.w3.org/2001/XMLSchema#string"
82                  AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
83              </SubjectMatch>
84            </Subject>
85          </Subjects>
86          <Resources>
87            <Resource>
88              <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
89                <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">i2</AttributeValue>
90                <ResourceAttributeDesignator
91                  DataType="http://www.w3.org/2001/XMLSchema#string"
92                  AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
93              </ResourceMatch>
94            </Resource>
95          </Resources>
96          <Actions>
97            <Action>
98              <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
99              <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">view</AttributeValue>
100               <ActionAttributeDesignator
101                 DataType="http://www.w3.org/2001/XMLSchema#string"
102                 AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
103             </ActionMatch>
104           </Action>
105         </Actions>
106       </Target>
107   </Rule>
108
109   <Rule RuleId="r3" Effect="Permit">
110     <Target>
111       <Subjects>
112         <Subject>
113           <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
114           <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">user</AttributeValue>
115             <SubjectAttributeDesignator
116               DataType="http://www.w3.org/2001/XMLSchema#string"
117               AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
118           </SubjectMatch>
119         </Subject>
120       </Subjects>
121       <Resources>
122         <Resource>
123           <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
124             <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">i3</AttributeValue>
125             <ResourceAttributeDesignator
126               DataType="http://www.w3.org/2001/XMLSchema#string"
127                     AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
128           </ResourceMatch>
129         </Resource>
130       </Resources>
131       <Actions>
132         <Action>
133           <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
134           <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">view</AttributeValue>
135             <ActionAttributeDesignator
136               DataType="http://www.w3.org/2001/XMLSchema#string"
137               AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
138           </ActionMatch>
139         </Action>
140       </Actions>
141     </Target>
142   </Rule>
143
144   <Rule RuleId="r4" Effect="Permit">
145     <Target>
146       <Subjects>
147         <Subject>
148           <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
149           <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">user</AttributeValue>
150             <SubjectAttributeDesignator
151               DataType="http://www.w3.org/2001/XMLSchema#string"
152               AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
153           </SubjectMatch>
154         </Subject>
155       </Subjects>
156       <Resources>
157         <Resource>
158           <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
159             <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">i4</AttributeValue>
160             <ResourceAttributeDesignator
```

```
161                 DataType="http://www.w3.org/2001/XMLSchema#string"
162                 AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
163               </ResourceMatch>
164             </Resource>
165           </Resources>
166           <Actions>
167             <Action>
168               <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
169               <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">view</AttributeValue>
170                 <ActionAttributeDesignator
171                   DataType="http://www.w3.org/2001/XMLSchema#string"
172                   AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
173               </ActionMatch>
174             </Action>
175           </Actions>
176         </Target>
177       </Rule>
178
179       <Rule RuleId="r5" Effect="Permit">
180         <Target>
181           <Subjects>
182             <Subject>
183               <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
184               <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">user</AttributeValue>
185                 <SubjectAttributeDesignator
186                   DataType="http://www.w3.org/2001/XMLSchema#string"
187                   AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
188               </SubjectMatch>
189             </Subject>
190           </Subjects>
191           <Resources>
192             <Resource>
193               <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
194                 <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">i5</AttributeValue>
195                 <ResourceAttributeDesignator
196                   DataType="http://www.w3.org/2001/XMLSchema#string"
197                   AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
198               </ResourceMatch>
199             </Resource>
200           </Resources>
201           <Actions>
202             <Action>
203               <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
204               <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">view</AttributeValue>
205                 <ActionAttributeDesignator
206                   DataType="http://www.w3.org/2001/XMLSchema#string"
207                   AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
208               </ActionMatch>
209             </Action>
210           </Actions>
211         </Target>
212       </Rule>
213
214       <Rule RuleId="r6" Effect="Permit">
215         <Target>
216           <Subjects>
217             <Subject>
218               <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
219               <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">user</AttributeValue>
220                 <SubjectAttributeDesignator
221                   DataType="http://www.w3.org/2001/XMLSchema#string"
222                   AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
223               </SubjectMatch>
224             </Subject>
225           </Subjects>
226           <Resources>
227             <Resource>
228               <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
229                 <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">i6</AttributeValue>
230                 <ResourceAttributeDesignator
231                   DataType="http://www.w3.org/2001/XMLSchema#string"
232                   AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
233               </ResourceMatch>
234             </Resource>
235           </Resources>
236           <Actions>
237             <Action>
238               <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
239               <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">view</AttributeValue>
240                 <ActionAttributeDesignator
241                   DataType="http://www.w3.org/2001/XMLSchema#string"
242                   AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
```

```
243             </ActionMatch>
244           </Action>
245         </Actions>
246       </Target>
247     </Rule>
248
249     <Rule RuleId="r7" Effect="Permit">
250       <Target>
251         <Subjects>
252           <Subject>
253             <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
254             <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">user</AttributeValue>
255               <SubjectAttributeDesignator
256                 DataType="http://www.w3.org/2001/XMLSchema#string"
257                 AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
258             </SubjectMatch>
259           </Subject>
260         </Subjects>
261         <Resources>
262           <Resource>
263             <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
264               <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">i7</AttributeValue>
265               <ResourceAttributeDesignator
266                 DataType="http://www.w3.org/2001/XMLSchema#string"
267                 AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
268             </ResourceMatch>
269           </Resource>
270         </Resources>
271         <Actions>
272           <Action>
273             <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
274             <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">view</AttributeValue>
275               <ActionAttributeDesignator
276                 DataType="http://www.w3.org/2001/XMLSchema#string"
277                 AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
278             </ActionMatch>
279           </Action>
280         </Actions>
281       </Target>
282     </Rule>
283   </Policy>
284 </PolicySet>
```

In the next chapter, we use a more complex case study to demonstrate how policies like this are automatically generated and deployed. We notice that XACML policies, even for this small example, can be unwieldy and cumbersome—a situation that can easily introduce errors if each one was manually updated. Not only would our approach automatically update and generate these XACML policies, but also provide assurance that the currently deployed policy meets the requirements of the data source owner.

Next, we consider the performance of the Alloy Analyzer in the analysis and verification of metapolicies and policies.

4.12 Performance

The Alloy Analyzer converts any model written in the Alloy language to a Boolean formula in conjunctive normal form (CNF). The Analyzer then aims to achieve an assignment of variables that makes this formula true. With respect to performance, there are two distinct benchmarks here—the time taken to convert the model to CNF, and the time taken to find a satisfying assignment. As mentioned previously, an essential feature of the Analyzer is the declaration of a scope, which constrains the number of signature atoms in the model. In general, this scope (the number of atoms for top-level signatures) is the most important factor which affects the overall analysis time. The CNF generation time is affected by the

complexity of the predicates, whereas the CNF solving time is affected by the constrained part of the state space—in turn the size of this state space (number of variables) is proportional to the number of signatures, fields and atoms.

In our results, we observe that the Analyzer (build 4.1.10) takes some time (approximately five minutes) to build up the state space (generate the CNF), but less than a second to find no counterexamples (solve the CNF)—each check has been executed on an Apple iMac running Mac OS X 10.5.8 using a 2.4 GHz Intel Core 2 Duo with 2 GB (667 MHz) DDR2 SDRAM. We see that it takes about 300 seconds to build up an entire state space of approximately 6000 primary variables with 150 atoms.

4.13 Summary

This chapter provides the major contribution of our thesis and has presented several facets about the EAC abstraction. The Alloy language and associated tool have first been introduced as a lightweight formal method that can analyse models within a bounded scope. Throughout, a simple example from the point of view of an access control policy writer has been used to visualise how the complex constructions are created using a suitable tool.

Using Alloy, we defined basic access control components such as rules, policies, and metapolicies providing corresponding examples to clarify our stance. Then, we specified our audit data source which logs internal and external events in a sequence. We also declared the notion of conditional events—"trigger" sequences of events that cause state transitions, which can reflect the nature of high-level requirements. With this in mind, we then described how metapolicies automatically direct the evolution of EAC policies. An evolution is signalled when a "trigger" sequence forms a contiguous or non-contiguous sequence in the audit data source. This prompts the current EAC policy to transition to the next state according to the metapolicy. Effectively, we are searching for data patterns in the audit data source to indicate when states should evolve.

Furthermore, we suggested a model of analysis and verification by first checking that metapolicies possess certain desirable properties, and second, checking that the current state of the system—the relationship between policies and the sequences of logged accesses—is consistent with the requirements of that metapolicy.

Next, we introduced a model of transformation and deployment which allows us to translate our version of policies to a machine-readable version that can be deployed in real access control systems.

Finally, the example of the online image facility has been used to show how our formal abstraction supports the construction, analysis and deployment of small models of EAC metapolicies and policies. Ultimately, we use this work as a stable basis in the development of suitable tools and technologies that support the secure management of data by bridging the gap between high-level requirements and low-level implementations of policies that meet those requirements. In the next chapter, we now aim to validate the contribution of this thesis using a more significant case study.

5

A Case Study: University Admissions Service

5.1 Introduction

In this chapter, we use a sufficiently complex case study to validate the major contribution of this thesis. Our aim is to provide some level of assurance that data sharing is appropriate using our formal EAC model to construct, analyse and deploy metapolicies and policies. The case study focuses on applying our methodology to a common scenario involving the sharing of admissions data at a local university. Additionally, we have implemented an application using the Java programming language, which demonstrates how we can deploy verified metapolicies and policies so that users can safely access the admissions data based on particular requirements.

We open this chapter with a description of the data source, the requirements, and its users. Then, we use our methodology to construct metapolicies for each user so that we can check each one for key properties and consistency before translating policies to XACML. Verified metapolicies are then integrated into our application while policies are deployed to the sif middleware. Next, we discuss the results obtained when users request access to protected resources based on certain requirements using our application. Finally, we compare our results to other related models, and identify main differences.

5.2 Context

In this section, we present the background of our case study. Three admissions officers of a university would like access to a data source containing sensitive student admissions information. The information concerns typical university admissions data such as a student's name, address, degree type, course, supervisor and so on. Ultimately, the university owns this information, but bequeaths this responsibility to a trusted individual, who reports to a data protection officer (ensures the protection of information) and a freedom of information officer (guarantees access to information), both of whom have authority throughout the institution. Given the confidential nature of the data, the data source owner would like to use a secure system which can capture high-level requirements or rules on how each of these officers can access this data. We consider the information contained in our data source, and then the requirements imposed on the three admissions officers.

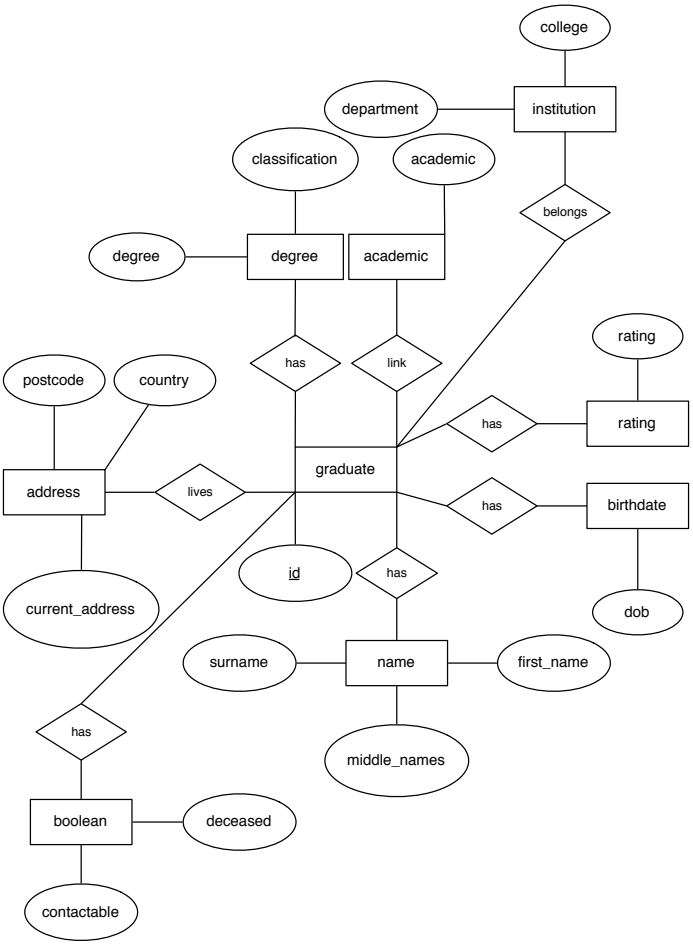


Figure 5.1: The ER model of Graduate data

5.2.1 Student Admissions Data Source

The data used in this case study originates from the Data Security (DAS)[1] postgraduate course offered by the Software Engineering Programme at the University of Oxford.[2] The entity-relationship model using Chen's notation [133] is used to describe the conceptual structure of the data source containing our student admissions information. In this model, entities, which can be thought of as nouns, are illustrated using rectangles. A relationship captures how entities are related, illustrated using diamonds, while attributes are illustrated using ovals. We shall assume that the logical types have been defined (and will be resolved at implementation), and that the value of *student_id* is unique for each student in the database. We also note that primary keys have been underlined.

The *Graduates* alumni table captures information on those students who have graduated. The *contactable* attribute denotes whether or not the graduate wants to be contacted, whereas *link* identifies the member of academic staff who can make contact with that graduate. The *rating* is a numeric value (between 1 and 5 inclusive, with 5 being the highest) indicating how willing a graduate is able to donate to the university.

[1] http://www.cs.ox.ac.uk/softeng/subjects/DAS.html
[2] http://www.cs.ox.ac.uk/softeng/

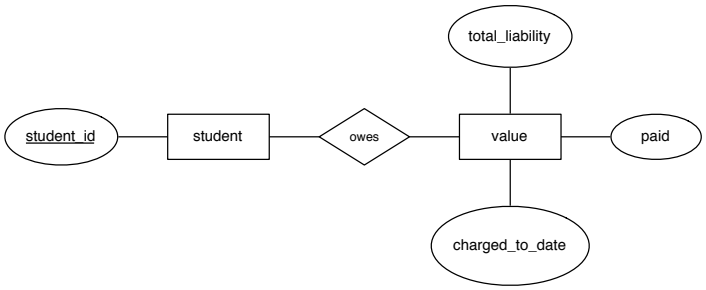


Figure 5.2: The ER model of Fees data

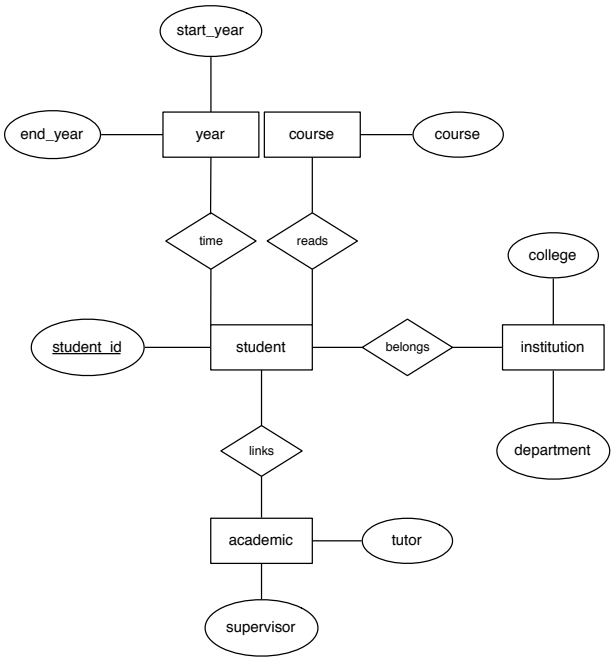


Figure 5.3: The ER model of Courses data

The *Fees* table represents students' financial information. The *total_liability* of fees is dependent on the student's status—EU students are charged at a different rate from overseas students, and some students are being funded by a scholarship, award or grant.

The *Course_details* relation contains basic information about students' courses. Each course is associated with a department. Additionally, every student has a supervisor, who is associated with that student's department, and a tutor, who is linked to that student's college.

As an example, if at any instant in time, a current policy allows a user to view the data associated with *student_id* 17245, the query will return the following row of information for that student.

student_id	start_year	end_year	course	department	college	supervisor	tutor
17245	2008	2011	Biology	Life Sciences	St Bobs	Mrs Herd	Mr Flock

In this data source, we note that current student records are mapped to 5-digit identification numbers; graduate records are mapped to 4-digit identification numbers.

5.2.2 Requirements and Users

Alice, our first admissions officer and user, would like access to *graduate* data so that she can contact alumni concerning donations to the university. Bob, on the other hand, would like constant access to *fees* data so that he can monitor the payment progress for certain students, and flag when those students have not paid in time. Finally, Charlie would like general access to *student* data so that he can update records when necessary. The trusted data source owner, David, wishes to impose the following requirements on the officers—each user is associated with three requirements (drawn from our nine classes of motivating requirements of Chapter 3). The data protection and freedom of information officers frequently liaise with David to ensure that these requirements are always preserved.

- Binary: Alice can access any graduate record but only once.
- Period: Alice can only access two graduate records per hour.
- Subscription: Alice can not access more than 10 rating points worth of graduate data.
- Reject: If Bob has been denied access four times, then ban all his future accesses.
- Liveness: If Bob has no activity for five days, then ban all his future accesses.
- System: Bob can not access any student record if the network bandwidth exceeds 50 points.
- Counting: Charlie can not access more than three student records.
- Compartment: Charlie can access any student record S, but can only access records of S's college after.
- Emergency: The data source owner, David, can override Charlie's current policy at any time.

5.3 EAC Model

For the sake of brevity, in this section, we examine only Alice's requirements in the demonstration of constructing, analysing and deploying EAC metapolicies and policies. We will consider Bob and Charlie again in Section 5.4

Alice can only access alumni data in the student admissions data source, with access being in accordance with the following requirements.

- Binary: Alice can access any graduate record but only once.
- Period: Alice can only access two graduate records per hour.
- Subscription: Alice can not access more than 10 rating points worth of graduate data.

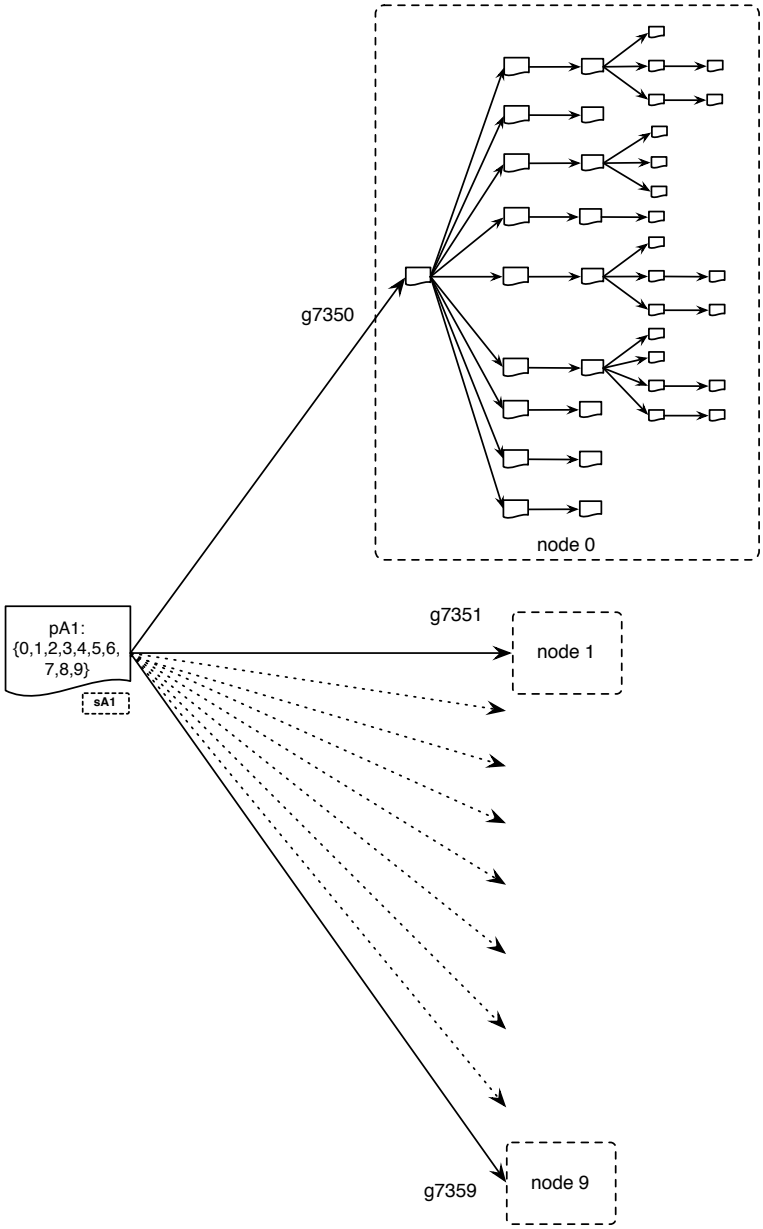


Figure 5.4: The state diagram of Alice's metapolicy

Alice's state diagram is potentially very large (as depicted in Figure 5.4 with 10 resources). A metapolicy of this size will be difficult to analyse using the Alloy tool (we discuss this limitation and its impact on scalability in the next chapter). We therefore address this issue by decomposing Alice's state diagram into 10 nodes, and analyse each node separately. In the forthcoming description, we concentrate on just node 0 as pictured in Figure 5.5 overleaf. Node 0 encompasses an initial access of resource g7350 from state 1, the initial policy, and the successive state graph from that point.

The state diagrams of Bob and Charlie are small enough to be fully analysable without the need to decompose either of them into nodes. Their respective metapolicies are analysed in a straighforward fashion.

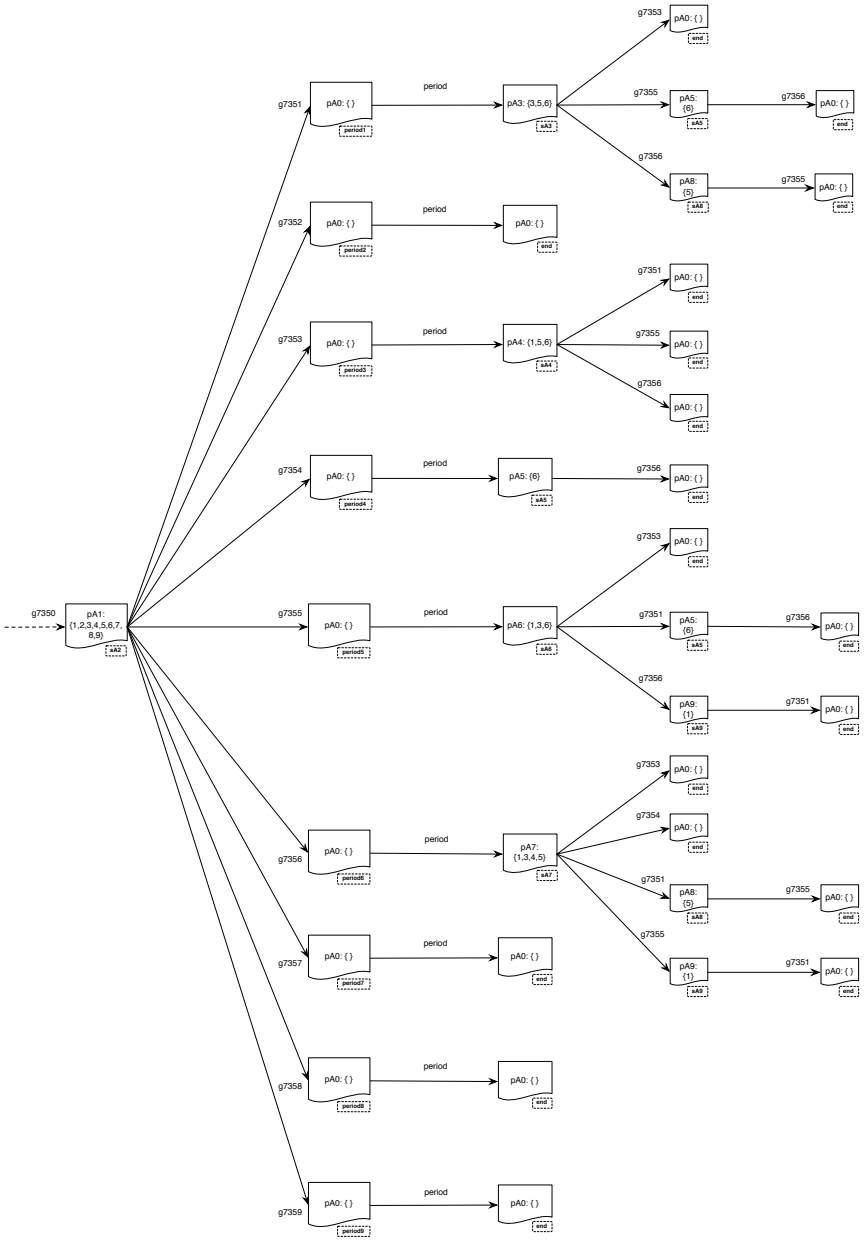


Figure 5.5: The state diagram of Alice's node 0

5.3.1 Construction

Our EAC model is used to first encode Alice's metapolicy—we do this for every node of her metapolicy but, for the sake of brevity, we only present the construction and analysis of node 0 as pictured in Figure 5.5. The Alloy model for this node is listed in Appendix D.3.

one sig alice **extends** Subject {}
one sig g7350, g7351, g7352, g7353, g7354 **extends** Resource {}
one sig g7355, g7356, g7357, g7358, g7359 **extends** Resource {}
one sig rA0, rA1, rA2, rA3, rA4, rA5, rA6, rA7, rA8, rA9 **extends** RuleID {}
one sig access **extends** Action {}
one sig period **extends** External {}
one sig node0 **extends** MetaPolicyID {}
one sig mcA0, mcA1, mcA2, mcA3, mcA4, mcA5 **extends** MPConditionID {}

```
one sig mcA6, mcA7, mcA8, mcA9, mcA10 extends MPConditionID {}
one sig pA0, pA2, pA3, pA4, pA5, pA6, pA7, pA8, pA9 extends PolicyID {}
one sig sA2, sA3, sA4, sA5, sA6, sA7, sA8, sA9, end extends StateID {}
one sig period1, period2, period3, period4 extends StateID {}
one sig period5, period6, period7, period8, period9 extends StateID {}

one sig csamp extends MetaPolicy {} {
  mid = node0
  initialstate = sA2
  rule = {(rA0 → ruleA0) + (rA1 → ruleA1) + (rA2 → ruleA2) + (rA3 → ruleA3) +
          (rA4 → ruleA4) + (rA5 → ruleA5) + (rA6 → ruleA6) + (rA7 → ruleA7) +
          (rA8 → ruleA8) + (rA9 → ruleA9)}
  policy = {(pA0 → policyempty) + (pA2 → policyA2) + (pA3 → policyA3) +
            (pA4 → policyA4) + (pA5 → policyA5) + (pA6 → policyA6) +
            (pA7 → policyA7) + (pA8 → policyA8) + (pA9 → policyA9)}
  states = {(sA2 → pA2) + (sA3 → pA3) + (sA4 → pA4) + (sA5 → pA5) + (sA6 → pA6) +
            (sA7 → pA7) + (sA8 → pA8) + (sA9 → pA9) + (end → pA0) +
            (period1 → pA0) + (period2 → pA0) + (period3 → pA0) + (period4 → pA0) +
            (period5 → pA0) + (period6 → pA0) + (period7 → pA0) + (period8 → pA0) +
            (period9 → pA0)}
  transitions = {(sA2 → ((0 → psA1) + (1 → psA2) + (2 → psA3) + (3 → psA4) +
                         (4 → psA5) + (5 → psA6) + (6 → psA7) + (7 → psA8) + (8 → psA9))) +
                 (sA3 → ((0 → psA11) + (1 → psA12) + (2 → psA13))) +
                 (sA4 → ((0 → psA15) + (1 → psA16) + (2 → psA17))) +
                 (sA5 → (0 → psA15)) +
                 (sA6 → ((0 → psA11) + (1 → psA20) + (2 → psA21))) +
                 (sA7 → ((0 → psA11) + (1 → psA23) + (2 → psA24) + (3 → psA25))) +
                 (sA8 → (0 → psA16)) +
                 (sA9 → (0 → psA17)) +
                 (period1 → (0 → psA10)) +
                 (period2 → (0 → psA26)) +
                 (period3 → (0 → psA14)) +
                 (period4 → (0 → psA18)) +
                 (period5 → (0 → psA19)) +
                 (period6 → (0 → psA22)) +
                 (period7 → (0 → psA26)) +
                 (period8 → (0 → psA26)) +
                 (period9 → (0 → psA26))}
}
```

5.3.2 Analysis

In this section, we apply our assertion patterns for checking metapolicy properties and policy consistency. Again, for the sake of brevity, we only present the analysis results of Alice's node 0—the list of results for all three users is presented in Appendix D.4. Concerning Alice's node 0, we note that the integer bitwidth is set to 5 since this metapolicy instance uses integers in the range -32 to +31. The sequence length is set to 9 as the initial policy is a sequence of nine rules. Finally, the number of System and ADDB signatures must be set to 5 to fully embody the representation of Figure 5.5.

MetaPolicy Properties

First, we consider properties of the metapolicy—checking for determinism and connectedness in our model produces no counterexamples.

```
assert determinism {
all s: System | {
    (all i: dom[s.metapolicy.transitions] |
        all disj j, k: dom[s.metapolicy.transitions[i]] |
            first[s.metapolicy.transitions[i][j]] != first[s.metapolicy.transitions[i][k]])
            }
}

check determinism for exactly 5 System, 5 ADDB, 5 int, 9 seq

Executing "Check determinism for 5 int, 9 seq, exactly 5 System, 5 ADDB"
   Solver=sat4j Bitwidth=5 MaxSeq=9 SkolemDepth=1 Symmetry=20
   212210 vars. 8825 primary vars. 681413 clauses. 2957243ms.
   No counterexample found. Assertion may be valid. 387ms.

assert connectedness {
    all s: System | {
        (dom[s.metapolicy.states] − s.metapolicy.initial) in ran[s.metapolicy.transitions].state
        }
}

check connectedness for exactly 5 System, 5 ADDB, 5 int, 9 seq

Executing "Check connectedness for 5 int, 9 seq, exactly 5 System, 5 ADDB"
   Solver=sat4j Bitwidth=5 MaxSeq=9 SkolemDepth=1 Symmetry=20
   207258 vars. 8789 primary vars. 656989 clauses. 2956395ms.
   No counterexample found. Assertion may be valid. 394ms.
```

We note, however, a failure case when checking for our restriction property—a counterexample is found.

```
assert restriction {
  all s: System − last[] | let s' = next[s] | {
    observation [s, s'] implies
      (accessibleResources [s'.metapolicy, (getPolicy [s'.metapolicy, s'.current])] in
        accessibleResources [s.metapolicy, (getPolicy [s.metapolicy, s.current])])
        }
}

check restriction for exactly 5 System, 5 ADDB, 5 int, 9 seq

Executing "Check restriction for 5 int, 9 seq, exactly 5 System, 5 ADDB"
   Solver=sat4j Bitwidth=5 MaxSeq=9 SkolemDepth=1 Symmetry=20
   205586 vars. 8789 primary vars. 653351 clauses. 2858049ms.
   Counterexample found. Assertion is invalid. 462ms.
```

However, this result is expected: when checking our restriction property, we ascertain that the accessible resources of the next policy is a subset of the previous policy. Due to the nature of Alice's second *period* requirement, the number of accessible resources decreases to zero (after two resources have been accessed within the hour), but then increases above zero (after an hour has elapsed) in the same trace as illustrated in Figure 5.5.

Policy Properties

Consistency checking here involves a modification of our assertion patterns to reflect the conditional nature of Alice's three requirements. For example, we would check for *binary* consistency using the following assertion pattern.

```
assert binary {
    all s: System | {
      all r: accessibleResources[s.metapolicy, (getPolicy[s.metapolicy, s.current])] |
        r !in s.addb.data.elems
        }
}

check binary for exactly 5 System, 5 ADDB, 5 int, 9 seq

Executing "Check binary for 5 int, 9 seq, exactly 5 System, 5 ADDB"
   Solver=sat4j Bitwidth=5 MaxSeq=9 SkolemDepth=1 Symmetry=20
   204514 vars. 8799 primary vars. 651461 clauses. 2853156ms.
   No counterexample found. Assertion may be valid. 560ms.
```

In verifying the consistency of Alice's *period* requirement, we check that whenever the period external event is logged, the number of accessible resources in the previous policy is zero while the number of accessible resources in the next policy is greater than zero. This means that the previous policy has blocked access until the external event, period, has signalled that an hour has expired thus re-allowing access to resources. We use the following assertion pattern to check for this situation.

```
assert period {
    all s: System - last[] | let s' = next[s] | {
      observation [s, s'] implies
        (s'.addb.data.last = period implies
           no accessibleResources[s.metapolicy, (getPolicy[s.metapolicy, s.current])]
           and
           #accessibleResources[s.metapolicy, (getPolicy[s.metapolicy, s'.current])] >= 0)
           }
}

check period for exactly 5 System, 5 ADDB, 5 int, 9 seq

Executing "Check period for 5 int, 9 seq, exactly 5 System, 5 ADDB"
   Solver=sat4j Bitwidth=5 MaxSeq=9 SkolemDepth=1 Symmetry=20
   231765 vars. 8800 primary vars. 750717 clauses. 3575365ms.
   No counterexample found. Assertion may be valid. 689ms.
```

	Metapolicy Properties			Policy Consistency					
Node	**Deterministic**	**Connected**	**Restricted**	**Binary**	**Period**	**Subscription**	**States**	**Atoms**	**Average Time (min)**
0	Yes	Yes	No	Yes	Yes	Yes	18	176	48.7
1	Yes	Yes	No	Yes	Yes	Yes	22	200	51.0
2	Yes	Yes	No	Yes	Yes	Yes	18	176	48.7
3	Yes	Yes	No	Yes	Yes	Yes	20	186	49.9
4	Yes	Yes	No	Yes	Yes	Yes	19	185	49.4
5	Yes	Yes	No	Yes	Yes	Yes	22	200	51.0
6	Yes	Yes	No	Yes	Yes	Yes	24	215	52.2
7	Yes	Yes	No	Yes	Yes	Yes	18	176	48.7
8	Yes	Yes	No	Yes	Yes	Yes	18	176	48.7
9	Yes	Yes	No	Yes	Yes	Yes	18	176	48.7

Table 5.1: The analysis results of Alice's metapolicy

The *subscription* requirement can be checked if we verify that the cumulative rating value does not exceed five as evolution proceeds (for node 0, we assume that record g7350, rated at five points, has already been viewed so that five points is yet to be accessed).

```
assert subscription {
    all s: System | {
      all r: accessibleResources[s.metapolicy, (getPolicy[s.metapolicy, s.current])] |
        ((sum n: s.addb.data.elems | Domain.values[n])) + Domain.values[r] =< 5
        }
}

check subscription for exactly 5 System, 5 ADDB, 5 int, 9 seq

Executing "Check subscription for 5 int, 9 seq, exactly 5 System, 5 ADDB"
   Solver=sat4j Bitwidth=5 MaxSeq=9 SkolemDepth=1 Symmetry=20
   208564 vars. 8799 primary vars. 668068 clauses. 2861171ms.
   No counterexample found. Assertion may be valid. 687ms.
```

In Table 5.1, we show the results of checking the remaining nodes of Alice's metapolicy. We observe that the number of atoms used to encode a metapolicy is proportional to the number of states in that metapolicy. The metapolicies of Bob and Charlie are simpler to analyse as the total number of states are 2 and 11 respectively, which do not require node decomposition. These results are all listed in Appendix D.4. In the next section, we put theory to practice and translate these verified EAC policies to XACML.

5.3.3 Transformation

In this section, we show how we can transform each user's EAC policy into XACML to make up a policy set. Since this transformation occurs every time a state is updated, we assume that at some point in the evolution of policies, that we have the following context of rules and policies for each user.

one sig ruleA1 **extends** EACRule {} {
    rid = rA1 **and** subject = alice **and** resource = g7351 **and** effect = Permit **and** action = access
}

one sig ruleA3 **extends** EACRule {} {
    rid = rA3 **and** subject = alice **and** resource = g7353 **and** effect = Permit **and** action = access
}

one sig ruleA6 **extends** EACRule {} {
    rid = rA6 **and** subject = alice **and** resource = g7356 **and** effect = Permit **and** action = access
}

one sig ruleC0 **extends** EACRule {} {
    rid = rC0 **and** subject = charlie **and** resource = s39465 **and** effect = Permit **and** action = access
}

one sig ruleC9 **extends** EACRule {} {
    rid = rC9 **and** subject = charlie **and** resource = s40566 **and** effect = Permit **and** action = access
}

one sig policyA6 **extends** EACPolicy {} {
    pid = pA6 **and** rules = {(0 → rA1) + (1 → rA3) + (2 → rA6)}
}

one sig policyB3 **extends** EACPolicy {} {
    pid = pA0 **and no** rules
}

one sig policyC2 **extends** EACPolicy {} {
    pid = pC2 **and** rules = {(0 → rC0) + (1 → rC9)}
}

Using our transformation map, three EAC policies (one for each user) can be compiled into an XACML policy set and deployed in an access control system. This policy set is listed below—the policy pertaining to Alice is listed in lines 19 to 143, that of Bob in lines 145 to 164 (we notice that the absence of rules here denies all requests in a *deny-biased* mechanism), and that of Charlie in lines 166 to 255.

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <PolicySet xmlns="urn:oasis:names:tc:xacml:1.0:policy"
 3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 4   xsi:schemaLocation="urn:oasis:names:tc:xacml:1.0:policy cs-xacml-schema-policy-01.xsd"
 5   PolicyCombiningAlgId="urn:oasis:names:tc:xacml:1.0:policy-combining-algorithm:deny-overrides"
 6   PolicySetId="csxacml632">
 7   <Target>
 8     <Subjects>
 9       <AnySubject/>
10     </Subjects>
11     <Resources>
12       <AnyResource/>
13     </Resources>
14     <Actions>
15       <AnyAction/>
16     </Actions>
17   </Target>
18
19   <Policy PolicyId="pa6" RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:deny-overrides">
20     <Target>
21       <Subjects>
```

```
22          <Subject>
23            <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
24              <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">alice</AttributeValue>
25              <SubjectAttributeDesignator
26                DataType="http://www.w3.org/2001/XMLSchema#string"
27                AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
28            </SubjectMatch>
29          </Subject>
30        </Subjects>
31        <Resources>
32          <AnyResource/>
33        </Resources>
34        <Actions>
35          <AnyAction/>
36        </Actions>
37      </Target>
38
39      <Rule RuleId="ra1" Effect="Permit">
40        <Target>
41          <Subjects>
42            <Subject>
43              <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
44              <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">alice</AttributeValue>
45                <SubjectAttributeDesignator
46                  DataType="http://www.w3.org/2001/XMLSchema#string"
47                  AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
48              </SubjectMatch>
49            </Subject>
50          </Subjects>
51          <Resources>
52            <Resource>
53              <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
54                <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">g7351</AttributeValue>
55                <ResourceAttributeDesignator
56                  DataType="http://www.w3.org/2001/XMLSchema#string"
57                  AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
58              </ResourceMatch>
59            </Resource>
60          </Resources>
61          <Actions>
62            <Action>
63              <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
64                <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">access</AttributeValue>
65                <ActionAttributeDesignator
66                  DataType="http://www.w3.org/2001/XMLSchema#string"
67                  AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
68              </ActionMatch>
69            </Action>
70          </Actions>
71        </Target>
72      </Rule>
73
74      <Rule RuleId="ra3" Effect="Permit">
75        <Target>
76          <Subjects>
77            <Subject>
78              <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
79              <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">alice</AttributeValue>
80                <SubjectAttributeDesignator
81                  DataType="http://www.w3.org/2001/XMLSchema#string"
82                  AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
83              </SubjectMatch>
84            </Subject>
85          </Subjects>
86          <Resources>
87            <Resource>
88              <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
89                <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">g7353</AttributeValue>
90                <ResourceAttributeDesignator
91                  DataType="http://www.w3.org/2001/XMLSchema#string"
92                  AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
93              </ResourceMatch>
94            </Resource>
95          </Resources>
96          <Actions>
97            <Action>
98              <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
99                <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">access</AttributeValue>
100               <ActionAttributeDesignator
101                 DataType="http://www.w3.org/2001/XMLSchema#string"
102                 AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
103           </ActionMatch>
```

```
104           </Action>
105         </Actions>
106       </Target>
107     </Rule>
108
109     <Rule RuleId="ra6" Effect="Permit">
110       <Target>
111         <Subjects>
112           <Subject>
113             <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
114             <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">alice</AttributeValue>
115               <SubjectAttributeDesignator
116                 DataType="http://www.w3.org/2001/XMLSchema#string"
117                 AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
118             </SubjectMatch>
119           </Subject>
120         </Subjects>
121         <Resources>
122           <Resource>
123             <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
124               <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">g7356</AttributeValue>
125               <ResourceAttributeDesignator
126                 DataType="http://www.w3.org/2001/XMLSchema#string"
127                 AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
128             </ResourceMatch>
129           </Resource>
130         </Resources>
131         <Actions>
132           <Action>
133             <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
134               <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">access</AttributeValue>
135               <ActionAttributeDesignator
136                 DataType="http://www.w3.org/2001/XMLSchema#string"
137                 AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
138             </ActionMatch>
139           </Action>
140         </Actions>
141       </Target>
142     </Rule>
143   </Policy>
144
145   <Policy PolicyId="pb3" RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:deny-overrides">
146     <Target>
147       <Subjects>
148         <Subject>
149           <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
150             <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">bob</AttributeValue>
151             <SubjectAttributeDesignator
152               DataType="http://www.w3.org/2001/XMLSchema#string"
153               AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
154           </SubjectMatch>
155         </Subject>
156       </Subjects>
157       <Resources>
158         <AnyResource/>
159       </Resources>
160       <Actions>
161         <AnyAction/>
162       </Actions>
163     </Target>
164   </Policy>
165
166   <Policy PolicyId="pc2" RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:deny-overrides">
167     <Target>
168       <Subjects>
169         <Subject>
170           <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
171           <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">charlie</AttributeValue>
172             <SubjectAttributeDesignator
173               DataType="http://www.w3.org/2001/XMLSchema#string"
174               AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
175           </SubjectMatch>
176         </Subject>
177       </Subjects>
178       <Resources>
179         <AnyResource/>
180       </Resources>
181       <Actions>
182         <AnyAction/>
183       </Actions>
184     </Target>
185
```

```
186     <Rule RuleId="rc0" Effect="Permit">
187       <Target>
188         <Subjects>
189           <Subject>
190             <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
191               <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">charlie</AttributeValue>
192               <SubjectAttributeDesignator
193                 DataType="http://www.w3.org/2001/XMLSchema#string"
194                 AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
195             </SubjectMatch>
196           </Subject>
197         </Subjects>
198         <Resources>
199           <Resource>
200             <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
201               <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">s39465</AttributeValue>
202               <ResourceAttributeDesignator
203                 DataType="http://www.w3.org/2001/XMLSchema#string"
204                 AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
205             </ResourceMatch>
206           </Resource>
207         </Resources>
208         <Actions>
209           <Action>
210             <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
211               <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">access</AttributeValue>
212               <ActionAttributeDesignator
213                 DataType="http://www.w3.org/2001/XMLSchema#string"
214                 AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
215             </ActionMatch>
216           </Action>
217         </Actions>
218       </Target>
219     </Rule>
220
221     <Rule RuleId="rc9" Effect="Permit">
222       <Target>
223         <Subjects>
224           <Subject>
225             <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
226               <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">charlie</AttributeValue>
227               <SubjectAttributeDesignator
228                 DataType="http://www.w3.org/2001/XMLSchema#string"
229                 AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
230             </SubjectMatch>
231           </Subject>
232         </Subjects>
233         <Resources>
234           <Resource>
235             <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
236               <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">s40566</AttributeValue>
237               <ResourceAttributeDesignator
238                 DataType="http://www.w3.org/2001/XMLSchema#string"
239                 AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
240             </ResourceMatch>
241           </Resource>
242         </Resources>
243         <Actions>
244           <Action>
245             <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
246               <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">access</AttributeValue>
247               <ActionAttributeDesignator
248                 DataType="http://www.w3.org/2001/XMLSchema#string"
249                 AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
250             </ActionMatch>
251           </Action>
252         </Actions>
253       </Target>
254     </Rule>
255   </Policy>
256 </PolicySet>
```

As we have now checked all the metapolicy states upfront for desirable properties and consistency, we can now deploy respective metapolicies and associated policies. In the next section, we see how a policy set for our three users is automatically generated and deployed whenever an evolution occurs according to the metapolicy.

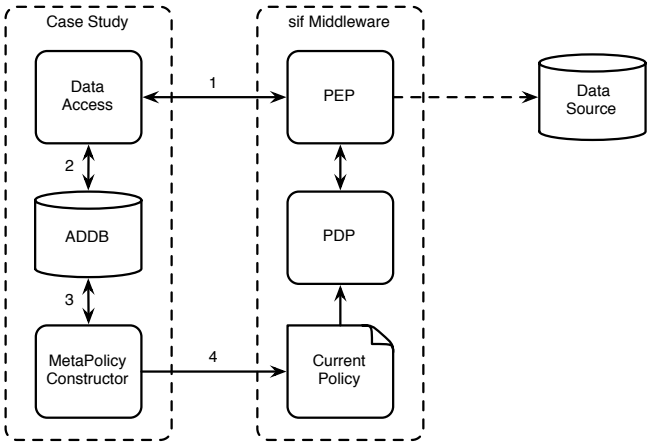


Figure 5.6: The application data-flow diagram

5.4 Application

In this section, we describe how a Java application has been implemented to validate the work of this thesis. In our approach, we deploy an instance of the sif middleware, and create the following three Java modules.

- A Data Access executable (DA) which allows users to send requests for, and view responses of data in the Student Admissions Data Source (DS).
- A State Database (ADDB) which logs all such requests as events.
- A MetaPolicy Constructor component (MPC) which captures high-level requirements, determines when the state changes, and then automatically generates policies.

5.4.1 Data-Flow

This system has the following data-flow as illustrated in Figure 5.6.

1. A user, via the DA, sends an access request for data in the DS. The sif middleware intercepts this request and consults the current policy to decide if that user can access that data.
2. This request, including the corresponding decision, is logged as an event in ADDB.
3. The MPC recognises when an event has been added, and computes if the state needs to be changed based on events logged in ADDB, and its supplied requirements.
4. If the MPC signals a state change, then a new XACML policy is generated and replaced in the sif middleware.

In an access control mechanism configured for XACML (such as sif), the Policy Enforcement Point (PEP) is the sole location at which requests are allowed or denied, whereas the Policy Decision Point (PDP) is the module which determines whether or not access to a resource is granted using the current policy.

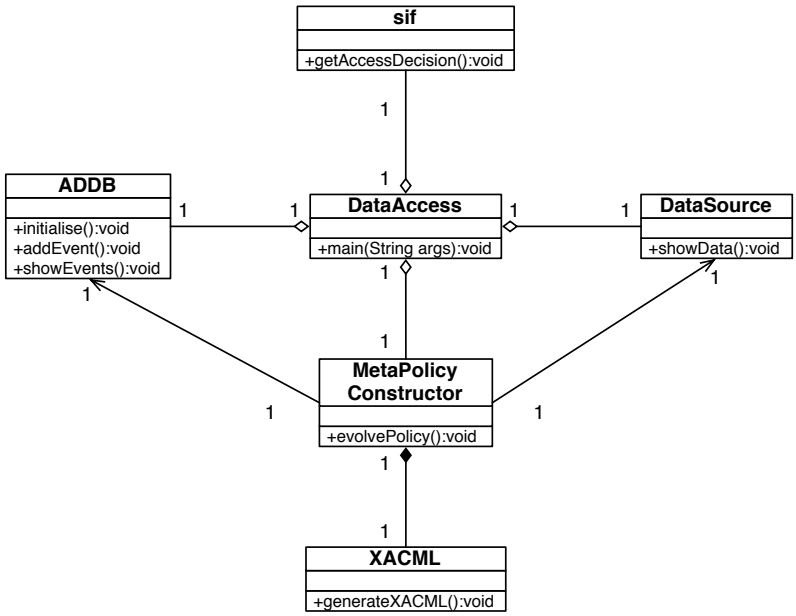


Figure 5.7: The application UML model

5.4.2 Implementation

The DA is a simple, command-line Java application which, when executed, creates and initialises the other classes. The DA then prompts for a user, a resource and an action, which allows the application to constantly create requests from these arguments. These requests are then forwarded to the middleware, which uses the current policy to determine the results. Once the DA receives the response (the result of a request from the middleware), the user is permitted or denied access to the resource. Additionally, it creates an event from this response, which is then logged in the *data* table in ADDB.

When an event has been logged, this triggers the *evolvePolicy* operation in the MPC. This operation then searches ADDB for any "trigger" sequences that may be present. We can use a suitable tool that can also act as an interface to the MPC to deploy our verified metapolicies and policies. For now, we assume that the logical statements representing metapolicies can be translated to source code functions in the MPC. For example, Bob's first requirement, which is an instance of our *reject* class of requirements, is implemented as a Java function which interfaces with ADDB. The function queries ADDB for specific sequence patterns and signals a change of state if one is found according to the metapolicy.

If the MPC signals a change of state, then a verified EAC policy is translated to XACML for that user. The generation involves an internal construction of an EAC policy representation using the Java *ArrayList* data structure. The *ArrayList* data structure captures the sequence of EAC rule identifiers in the formal definition of an EAC policy. Once this EAC policy representation has been built up, the *generateXACML* function (in the XACML class) is called. This function generates an XACML policy set from an EAC policy representation. The XACML policy set is then automatically loaded into the sif middleware, replacing the previous policy and allowing the DA to accept another request. We provide a UML model of this architecture in Figure 5.7.

5.4.3 Development Tools

Our application has been developed with the Java programming language version 1.5 on an Apple iMac running Mac OS X 10.5.8 using a 2.4 GHz Intel Core 2 Duo with 2 GB (667 MHz) DDR2 SDRAM. We have used the Eclipse Enterprise Edition IDE[3] version 3.5 (Galileo) to implement, test and debug our application. The following plug-ins to Eclipse have been configured for our use.

- Apache Derby[4] Core (version 10.3.3) and UI (version 1.1.1) plug-ins have been used to manage our state database, ADDB, with one table named *data* (as formalised in Section 4.5).
- Subversion[5] (version 1.6) plug-in has been utilised to interface with our source control repository.

The Oxygen XML editor[6] (version 8.2) has also aided in testing our main XACML generator function (*XACML.generateXACML()*) by ensuring correct XACML policies have been generated, due to its native XML validation and well-formedness checks.

The development process has created 1,500 lines of code embodied in five main classes.

5.4.4 Results

After using our EAC abstraction to construct, analyse and translate verified metapolicies and policies, we demonstrate how policies evolve based on the nine data source requirements from each user's perspective. To support our discussion in this section, we shall interleave a narrative explaining the results with the corresponding Java console output. We shall also group results based on user—for instance, we show the step-by-step results concerning user Charlie when he progressively attempts to request data from the data source according to his three requirements.

Initialisation

When we execute the DA application, the following console output is produced. The classes in our application are instantiated and initialised: the initial policy is generated, and the state database is cleared. The DA then waits for input, and prompts for a user (Alice, Bob, Charlie or Admin), a resource (a *student_id* mapped to a row in the DS) and an action (for now, we assume once a successful *access* has been demonstrated, the row of data concerning the requested resource is listed). As mentioned previously, we assume that verified metapolicies and policies have been deployed by a suitable tool which integrates with our MPC component.

[3] http://www.eclipse.org/
[4] http://db.apache.org/derby/
[5] http://subversion.apache.org/
[6] http://www.oxygenxml.com/

```
Middleware started...
MetaPolicy initialised with 9 requirements...
Initial policy generated for 3 users: Alice, Bob, Charlie...
ADDB emptied...

Submit Request
User:
```

User Alice

Alice's three data source requirements are listed here again to remind the reader.

1. *Alice can access any graduate record but only once.*
2. *Alice can only access two graduate records per hour.*
3. *Alice can not access more than 10 rating points worth of graduate data.*

We demonstrate that Alice can indeed access a graduate record only once. In the console printout below, we observe that having accessed graduate record g7350, Alice is denied further access to that resource.

```
Submit Request
User:alice
Resource:7350
Action:access

Result is Permit!

Data Source:
```

id	first_name	middle_names	surname	dob	current_address	postcode
7350	Robert	James	Beagle	1933-12-09	23 East Street, Nottingham	NG2 7JP

```
FLAG BINARY: Evolve policy to deny access to resource 7350 for user alice.

ADDB:
```

subject	resource	result	timestamp
alice	7350	Permit	1.290287937692E12

```
Submit Request
User:alice
Resource:7350
Action:access

Result is Deny!

Data Source:

ADDB:
```

subject	resource	result	timestamp
alice	7350	Permit	1.290287937692E12
alice	7350	Deny	1.290287946278E12

The second ADDB log shows that she was first allowed access to resource g7350 but then denied. For the purposes of presentation, we truncate the row of graduate data returned from the DS because of its lengthy schema.

In validating that the second requirement holds for Alice, we continue from above and show that having successfully accessed two graduate records (resources g7350 and g7355) within the hour, further requests are denied (resource g7356). When that hour has timed out, Alice is once again able to access graduate records—previously she was denied access to resource g7356.

Submit Request
User:alice
Resource:7355
Action:access

Result is Permit!

Data Source:

id	first_name	middle_names	surname	dob	current_address	postcode
7355	Mary	null	Smith	1961-05-04	23 West Street, Nottingham	NG1 7JQ

FLAG BINARY: Evolve policy to deny access to resource 7355 for user alice.

FLAG SUBSCRIPTION: Evolve policy to deny access to resources as current worth of 7 has been accessed for user alice.

FLAG PERIOD: Evolve policy to deny access to all resources for alice.

ADDB:

subject	resource	result	timestamp
alice	7350	Permit	1.290287937692E12
alice	7350	Deny	1.290287946278E12
alice	7355	Permit	1.290287955005E12

Submit Request
User:alice
Resource:7356
Action:access

Result is Deny!

Data Source:

ADDB:

subject	resource	result	timestamp
alice	7350	Permit	1.290287937692E12
alice	7350	Deny	1.290287946278E12
alice	7355	Permit	1.290287955005E12
alice	7356	Deny	1.290287965343E12

FLAG PERIOD: Evolve policy to allow access to resources for alice as 1 hour has expired.

Each graduate record is associated with a particular "rating" (an integer between 1 and 5 inclusive). As Alice accesses records, the policy evolves so that the cumulative sum of records already accessed, and those available for access, do not exceed 10 points. We see that Alice first accesses resource g7350 rated at five points. Next, she accesses resource g7355 (rated at two points), and the policy evolves to restrict access to resources worth over three points. She then accesses resource g7356, which takes the cumulative sum up to eight. At this point, she can only access resource g7351.

As we see below, the policy evolves to deny all access once the full worth of 10 points has been accessed. We additionally observe that the *period* flag has been signalled as, again, two resources (g7356 and g7351) have been accessed in another hour period. However, because Alice has satisfied her *subscription* requirement, the policy still denies all access until it can be reset to an initial state—after some time, Alice attempts to access resource g7359 but is rejected. We also note that a *period* external event is logged in the audit data source.

Submit Request
User:alice
Resource:7356
Action:access

Result is Permit!

Data Source:

id	first_name	middle_names	surname	dob	current_address	postcode
7356	Barry	John	Slaymaker	1951-06-08	5 Keble Road, Oxford	OX1 3QD

FLAG BINARY: Evolve policy to deny access to resource 7356 for user alice.

FLAG SUBSCRIPTION: Evolve policy to deny access to resources as current worth of 8 has been accessed for user alice.

ADDB:

subject	resource	result	timestamp
alice	7350	Permit	1.290287937692E12
alice	7350	Deny	1.290287946278E12
alice	7355	Permit	1.290287955005E12
alice	7356	Deny	1.290287965343E12
external	alice	period	1.290291537692E12
alice	7356	Permit	1.290291548192E12

Submit Request
User:alice
Resource:7351
Action:access

Result is Permit!

Data Source:

id	first_name	middle_names	surname	dob	current_address	postcode
7351	Tommy	TomTom	Thomson	1971-12-06	Sun Villa, St Johns	null

FLAG BINARY: Evolve policy to deny access to resource 7351 for user alice.

FLAG SUBSCRIPTION: Evolve policy to deny access to all resources as full worth of 10 has been accessed for user alice.

FLAG PERIOD: Evolve policy to deny access to all resources for alice.

ADDB:

subject	resource	result	timestamp
alice	7350	Permit	1.290287937692E12
alice	7350	Deny	1.290287946278E12
alice	7355	Permit	1.290287955005E12
alice	7356	Deny	1.290287965343E12
external	alice	period	1.290291537692E12
alice	7356	Permit	1.290291548192E12
alice	7351	Permit	1.290291559492E12

Submit Request
User:alice
Resource:7359
Action:access

Result is Deny!

Data Source:

ADDB:

subject	resource	result	timestamp
alice	7350	Permit	1.290287937692E12
alice	7350	Deny	1.290287946278E12
alice	7355	Permit	1.290287955005E12
alice	7356	Deny	1.290287965343E12
external	alice	period	1.290291537692E12
alice	7356	Permit	1.290291548192E12
alice	7351	Permit	1.290291559492E12
external	alice	period	1.290295148192E12
alice	7359	Deny	1.290295158792E12

User Bob

The data source requirements imposed on Bob are as follows.

1. *If Bob has been denied access four times, then ban all his future accesses.*
2. *Bob can not access any student record if the network bandwidth exceeds 50 points.*
3. *If Bob has no activity for five days, then ban all his future accesses.*

In this section, we demonstrate the validation of these requirements. Bob can access current student records in our Student Admissions Data Source, but his attempts to access *Graduate* data are rejected.

As we see in the following console output, Bob tries to access *Graduate* data and is

completely locked out after his fourth rejection. In the interim, we notice that an external system event is triggered, and closes Bob's policy to all accesses—this occurs when the network bandwidth exceeds fifty points. We see that once the bandwidth drops below this threshold value, the policy evolves to allow normal access again according to Bob's other requirements. However, when this happens, Bob is able to view only one more student record before he triggers his rejection limit flag.

Submit Request
User:bob
Resource:7357
Action:access

Result is Deny!

Data Source:

ADDB:

subject	resource	result	timestamp
bob	7357	Deny	1.290522866598E12

FLAG SYSTEM: Evolve policy to deny access to all resources for user bob - network bandwidth exceeded.

Submit Request
User:bob
Resource:17245
Action:access

Result is Deny!

Data Source:

ADDB:

subject	resource	result	timestamp
bob	7357	Deny	1.290522866598E12
external	network	over	1.290522871589E12
bob	17245	Deny	1.290522886396E12

Submit Request
User:bob
Resource:7358
Action:access

Result is Deny!

Data Source:

ADDB:

subject	resource	result	timestamp
bob	7357	Deny	1.290522866598E12
external	network	over	1.290522871589E12
bob	17245	Deny	1.290522886396E12
bob	7358	Deny	1.290522911594E12

FLAG SYSTEM: Evolve policy to allow access to resources for bob - network bandwidth below limit.

Submit Request
User:bob
Resource:17245
Action:access

Result is Permit!

Data Source:

student_id	total_liability	charged_to_date	paid
17245	3000	3000	1500

ADDB:

subject	resource	result	timestamp
bob	7357	Deny	1.290522866598E12
external	network	over	1.290522871589E12
bob	17245	Deny	1.290522886396E12
bob	7358	Deny	1.290522911594E12
external	network	under	1.290522931589E12
bob	17245	Permit	1.290522951589E12

Submit Request
User:bob
Resource:7359
Action:access

Result is Deny!

Data Source:

FLAG REJECT: Evolve policy to deny all further access to resources for user bob - rejection limit reached.

ADDB:

subject	resource	result	timestamp
bob	7357	Deny	1.290522866598E12
external	network	over	1.290522871589E12
bob	17245	Deny	1.290522886396E12
bob	7358	Deny	1.290522911594E12
external	network	under	1.290522931589E12
bob	17245	Permit	1.290522951589E12
bob	7359	Deny	1.290522971589E12

For Bob's last requirement, David resets his policy and we show that once he has accessed student record, s47777, with no further activity, then the policy evolves to block all access after his set period of five days has expired.

Submit Request
User:bob
Resource:47777
Action:access

Result is Permit!

Data Source:

student_id	total_liability	charged_to_date	paid
47777	9000	6000	1000

ADDB:

subject	resource	result	timestamp
bob	47777	Permit	1.290522983852E12

FLAG LIVENESS: Evolve policy to deny all further access to resources for user bob.

Submit Request
User:bob
Resource:47777
Action:access

Result is Deny!

Data Source:

ADDB:

subject	resource	result	timestamp
bob	47777	Permit	1.290522983852E12
external	bob	exit	1.290954983852E12
bob	47777	Deny	1.290954994852E12

User Charlie

Charlie's data source requirements are as follows.

1. *Charlie can not access more than three student records.*
2. *Charlie can access any student record S, but can only access records of S's college after.*
3. *The data source owner, David, can override Charlie's current policy at any time.*

We show the results of all Charlie's imposed requirements in the following console output. Charlie initially accesses student record s38475, a member of Dukes college. The policy evolves to allow access to student records of only Dukes college. This now enables him to access student records s39465 and s40566 until his policy is locked down after the third access (due to his instance of the *counting* requirement).

However, Charlie needs immediate access to student record s72341, and contacts David, the data source owner and administrator, for emergency rights. David logs in and overrides

Charlie's policy, allowing him to access the record. Once accessed, David rescinds all access permissions, and Charlie's policy reverts to its previous state—a closed policy.

Submit Request
User:charlie
Resource:38475
Action:access

Result is Permit!

Data Source:

student_id	start_year	end_year	course	department	college	supervisor	tutor
38475	2009	2012	Anc History	History	Dukes	Dr Smith	Professor Stark

FLAG COMPARTMENT: Evolve policy to only allow access to resources that belong to Dukes for user charlie.

ADDB:

subject	resource	result	timestamp
charlie	38475	Permit	1.290595757728E12

Submit Request
User:charlie
Resource:39465
Action:access

Result is Permit!

Data Source:

student_id	start_year	end_year	course	department	college	supervisor	tutor
39465	2010	2013	Anc History	History	Dukes	Dr Kennedy	Dr Thomas

ADDB:

subject	resource	result	timestamp
charlie	38475	Permit	1.290595757728E12
charlie	39465	Permit	1.290595771525E12

Submit Request
User:charlie
Resource:40566
Action:access

Result is Permit!

Data Source:

student_id	start_year	end_year	course	department	college	supervisor	tutor
40566	2010	2013	Computer Science	Computing	Dukes	Mr Parser	Dr Chip

FLAG COUNTING: Evolve policy to deny all further access to resources for user charlie.

ADDB:

subject	resource	result	timestamp
charlie	38475	Permit	1.290595757728E12
charlie	39465	Permit	1.290595771525E12
charlie	40566	Permit	1.290596155267E12

Submit Request
User:charlie
Resource:72341
Action:access

Result is Deny!

Data Source:

ADDB:

subject	resource	result	timestamp
charlie	38475	Permit	1.290595757728E12
charlie	39465	Permit	1.290595771525E12
charlie	40566	Permit	1.290596155267E12
charlie	72341	Deny	1.290596200831E12

Submit Request
User:admin
Resource:charlie
Action:unlock

Emergency Override...

FLAG EMERGENCY: Evolve policy to allow access to all resources for user charlie.

Submit Request
User:charlie
Resource:72341
Action:view

Result is Permit!

Data Source:

student_id	start_year	end_year	course	department	college	supervisor	tutor
72341	2009	2012	Modern History	History	St Bobs	Dr Smith	Mr Marr

ADDB:

subject	resource	result	timestamp
charlie	38475	Permit	1.290595757728E12
charlie	39465	Permit	1.290595771525E12
charlie	40566	Permit	1.290596155267E12
charlie	72341	Deny	1.290596200831E12
admin	charlie	unlock	1.290596294525E12
charlie	72341	Permit	1.290596324828E12

Submit Request
User:admin
Resource:charlie
Action:lock

Emergency Override...

FLAG EMERGENCY: Evolve and revert to current policy for user charlie.

FLAG COUNTING: Evolve policy to deny all further access to resources for user charlie.

Submit Request
User:charlie
Resource:72341
Action:access

Result is Deny!

Data Source:

ADDB:

subject	resource	result	timestamp
charlie	38475	Permit	1.290595757728E12
charlie	39465	Permit	1.290595771525E12
charlie	40566	Permit	1.290596155267E12
charlie	72341	Deny	1.290596200831E12
admin	charlie	unlock	1.290596294525E12
charlie	72341	Permit	1.290596324828E12
admin	charlie	lock	1.290597675693E12
charlie	72341	Deny	1.290597859584E12

5.5 Model Comparison

In this section, we compare our executable EAC model to other models that use similar tools for analysis and verification. We present a subset of the publications of Sections 2.5 and 4.1.

Since our approach is novel compared to related work in our immediate field of research, we omit comparisons concerning the use of a metapolicy and audit data, and the ability to automatically evolve policies, as none of the models presented can offer all three of these features. The table below, beginning with our abstraction, is ordered by descending correlation between our approach and the related work.

Work	Model	Scalable	Notes
Sieunarine	EAC	No	Our EAC abstraction is an executable formal model using the Alloy language, which specifies the use of a metapolicy and audit data source to automatically evolve policies guided by requirements captured in the metapolicy, and based on user and external events. Metapolicies are formally represented as state spaces that can be checked upfront. However, state-space explosion only allows models with tens of complex states—nevertheless, we argue that this is sufficient for the goals of this thesis.

Work	Model	Scalable	Notes
Power et al. [6]	RBAC	Yes	The authors have developed a policy language based on Alloy, and associated tool, Gauge, which captures and verifies self-modifying access control policies respectively. The tool is scalable since it can check a policy before deployment unlike our approach where we check all policies upfront. The Gauge tool is described in greater detail in the next chapter.
Piessens et al. [7]	RBAC	Yes	Access Interface – View Connector approach ensures that the application logic that is used to capture requirements is robust by using aspect oriented software development (AOSD). While this makes the approach scalable, requirements are still hard-coded over various applications that want to connect to an authorisation engine, which makes automated reasoning about the policy difficult, as well as maintenance.
Røstad et al. [8]	RBAC	N/A	While the authors have identified a need to bridge the gap between high-level requirements and low-level implementations, no prototype exists, only a model of capturing requirements is conceptualised particularly in the healthcare field. However, they have identified that particular roles induce patterns in the audit data which can be used to assert medical guidelines.
Bharadwaj et al. [9]	MAC	N/A	The authors describe an architecture and mathematical framework using semiring-based constraint logic programming (SCLP) to capture requirements at different levels that can be combined to compose access control policies. There is neither deployment of policies at a low level nor is there a discussion about computation issues related to the constraint solver.
Jaco et al. [10]	RBAC	Yes	Interceptor chains are embedded within the application code and can be programmed to capture requirements in medical information systems. While expressiveness is greater, again, requirements are hard-coded into application code making reconfigurability impossible. Access control is not specified at a high level of abstraction.

Work	Model	Scalable	Notes
Bryans et al. [75]	DAC	N/A	An executable formal model using VDM is described with the goal of assisting the policy developer by providing immediate feedback on design decisions. Using VDM has allowed the use of VDMTools to validate policies based on testing as well as a natural translation to XACML. Scalability of the tools used is not discussed.
Zhang et al. [13]	DAC	No	A framework uses the RW formalism and tool to capture and verify policies respectively. The framework is used to detect errors in policies of existing access control systems, employing the use of an algorithm to evaluate such policies. The evaluation involves the assessment of authorised and unauthorised access to data using two modes. The tool that implements the algorithm uses binary decision diagrams (BDD) to achieve good performance. Nevertheless, the implementation still suffers from the state explosion problem although the authors have also cited Jackson's "small-scope" hypothesis.
Fisler et al. [14]	RBAC	N/A	The authors describe how they use the Margrave software suite for analysing policies by translating XACML policies to multi-terminal binary decision diagrams (MTBDD) to answer queries. The tool can verify that a policy meets a certain property, and can also check two policies for differences to verify properties of the change. While this approach also suffers from the state explosion problem, the authors argue that MTBDDs provide a scalable and flexible approach for analysis of policies.
Qunoo et al. [64]	DAC	N/A	The X-Policy modelling language is described, which is used to model the dynamic permissions of web-based collaborative systems like EasyChair. Using X-Policy, the authors reason about the security properties of such systems which allows the ability to identify certain weaknesses. However, no tools have been specified that can automate the analysis of these systems, although the authors have cited the need to develop suitable tools for this purpose.

Work	Model	Scalable	Notes
Hassan et al. [115]	RBAC	No	Alloy is used to detect the inconsistencies of policies expressed in XACML. Assertions are used to determine if such policies are compliant with the system described. The authors have used small Alloy examples to demonstrate their stance but large real-world examples are not scalable.
Zao et al. [116]	RBAC	N/A	The use of Alloy to analyse access control policies in this publication is one of the very first applications of this model finder. The authors use Alloy to model a RBAC schema framework so that properties, consistency, and implementation correctness can be verified. However, the performance of the Analyzer is not discussed.
Hughes et al. [117]	DAC	Yes	In this approach, XACML policies are verified via translation to a formal model, which can be reduced to a normal form that facilitates analysis using a SAT solver. The authors have achieved scalability as they have progressed from using Alloy to directly using a SAT solver.
Mankai et al. [118]	RBAC	N/A	A logical model of XACML is translated to the Alloy language so that possible conflicts within sets of XACML access control policies can be visualised and detected. The Alloy model of access control policies can then be analysed using a set of predicates and assertions which represent the properties to be verified. The authors have identified that while the examples encountered have been tractable, the scalability of the method presented is not fully explored.
Ray et al. [120]	RBAC	N/A	A spatio-temporal RBAC model for pervasive computing applications that can take environmental factors into account has been formalised, and then converted to the Alloy language for analysis. Analysis has shown that potential conflicts may occur in this model, and so an investigation into new techniques is planned. Moreover, the scalability of this approach is not discussed.

Table 5.2: A comparison of related work

5.6 Summary

In this chapter, we illustrated our work by means of a case study. The case study has also allowed us to demonstrate that our work can be useful in practice, as we have applied our formal approach to an example based on a real-world scenario. Three university admissions officers require access to student and graduate records contained in a central university data source. Their access has been mediated by high-level requirements drawn from our classes of motivating requirements, imposed by a data source owner, and captured by a metapolicy component.

With this in mind, this chapter began with a description of the logical structure of our admissions data source, and a list of the requirements imposed on each admissions officer. Next, we applied our formal analysis techniques to first construct the metapolicies for each user. Second, we checked each metapolicy for key properties, and then verified that policies consistently adhered to users' high-level requirements during evolution. This analysis has also allowed us to test the limits of scalability using the Alloy Analyzer, which we address in the next chapter. Third, our XACML translation process has been used to convert these verified EAC policies to XACML policies.

We then introduced a Java application that can query the admissions data source via the sif access control system (sif acts as a broker between our application and the admissions data source). We developed this application as a proof of concept which closely reflects the operations of our EAC abstraction—"trigger" sequences signal an evolution in the system which automatically translates, generates, and deploys verified policies. The high-level requirements of each verified metapolicy are captured in a metapolicy component that directs the evolution of these EAC policies according to user requests and system events. Results of this deployment have shown that high-level data source requirements can be embodied in a metapolicy that can drive the evolution of policies for a finite number of states.

Finally, we presented how our end-to-end approach of constructing, analysing and deploying metapolicies and policies differs from related work in our sphere of research.

Essentially, our aim is to provide assurance that our research—having reasoned about, verified and validated the use of EAC metapolicies and policies—is fit for deployment in a real system that protects sensitive data. We consider these goals more in-depth in our final chapter.

6

Discussion and Conclusion

6.1 Contributions

In this thesis, we presented work undertaken in developing a formal abstraction of evolving access control. In Chapter 1, we motivated our contribution by explaining that there is need for tools and technologies for the secure and appropriate sharing of data specifically because there exists a conceptual gap between legal and ethical guidelines written at a high level and access control policies implemented at a low level. We then introduced our intent to address this issue by developing a formal model of EAC which provides assurance that policies that are automatically deployed meet the high-level requirements embodied in a metapolicy. Chapter 2 has positioned the background for our work in this area by explaining basic access control concepts, presenting immediate and related research, and outlining our methodology. Chapter 3 is dedicated to EAC and explains what are metapolicies, its advantages, classes of requirements that can be captured by metapolicies, and four small examples that drive the development of our EAC abstraction. In Chapter 4, the Alloy language has been utilised to capture the EAC abstraction and its properties. Consequently, this has allowed us to, first, construct small examples of metapolicies and reason about the manner in which metapolicies can capture high-level requirements, and, thereafter, direct policy evolutions accordingly. Second, the Alloy Analyzer tool has enabled the analysis of metapolicies to ascertain particular properties, and the verification of policies to check that each evolved policy consistently meets the high-level requirements of the metapolicy. Third, we provided a means of policy deployment by translating EAC policies to XACML. Finally, in Chapter 5, we validated our work by applying our approach to a case study, which also includes a proof of concept application. The results show that, for a finite number of states, EAC policies can evolve guided by high-level requirements captured in a metapolicy, and based on internal user access and external system events.

6.2 Limitations and Drawbacks

One of the more challenging issues encountered during our research involved the scalability of the Alloy Analyzer. Scalability is limited by the ability to analyse large state spaces with this tool. The Analyzer is designed to find small instances that satisfy the constraints given

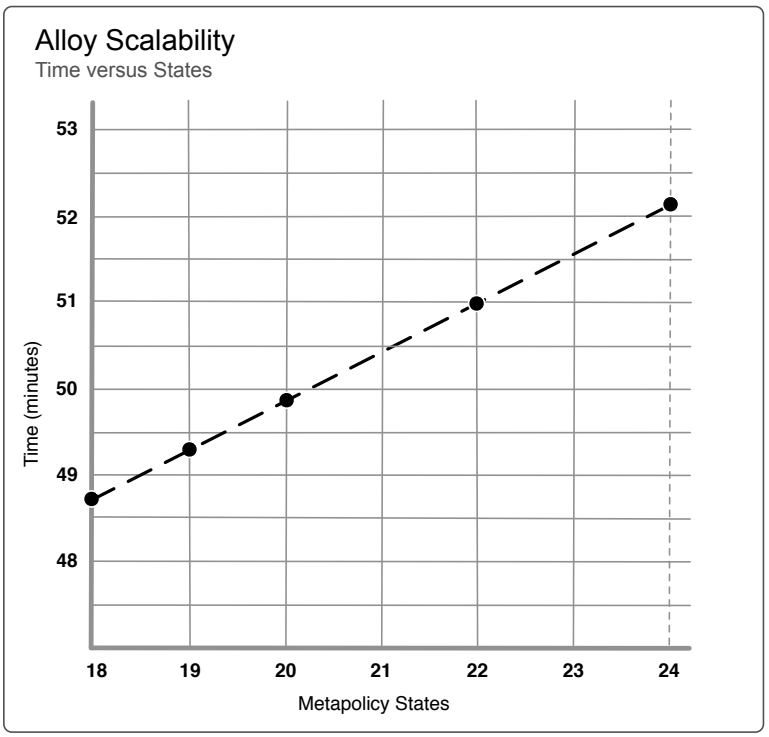


Figure 6.1: The scalability of the Alloy Analyzer

in the model, and is most suitable for scopes that possess only a finite number of states—this ensures that the model-finding problem is decidable. However, this has the immediate drawback of limiting the practicality of using the Analyzer.

As mentioned previously, the Alloy Analyzer works by first converting the Alloy model to a Boolean formula in conjunctive normal form (CNF). A SAT solver is then used to find an instance which satisfies that formula. In Alloy, it is crucial that the number of top-level signatures are constrained in a scope—larger scopes take more time to solve. Moreover, larger scopes can have a state space so enormous that the Analyzer will refuse to continue. For example, if we were to set the scope on an Alloy model to use an integer bitwidth of 12, then that means there are 4096 (2^{12}) atoms in the **Int** signature alone (a signature is a set of atoms, and the Alloy Analyzer uses the total number of atoms in each signature of the model to compute the CNF for that model). Furthermore, the existence of a ternary relation in that model also means that its state space will be over 68,719,476,736 (4096^3) atoms due to the **Int** scope alone. This value exceeds the range of integer values that can be stored in 32 bits (-2,147,483,648 through 2,147,483,647 for 32-bit signed integer). Consequently, this is where the limits of scalability are tested as the Analyzer will report the following error.

A type error has occurred:
Translation capacity exceeded.
In this scope, universe contains 4110 atoms and relations of arity 3 cannot be represented.

The time associated with solving our Alloy models of metapolicies increases with the number of states in that metapolicy. This is because the number of atoms used to encode and analyse a metapolicy's state space is proportional to the number of states in that metapolicy.

Additionally, our Alloy model contains a relation that is of arity 4—the transitions relation declared in the MetaPolicy signature. This relation is of arity 4 because of the manner in which Alloy treats sequences—the type of the sequence in this relation is a mapping from **Int** atoms to Priority atoms.

transitions → StateID → **Int** → Priority

From our calculations above, a relation of arity 4 in our Alloy models suggests that we can not go beyond 216 atoms in total as this produces a state space that is 2,176,782,336 (216^4) atoms, and, hence, larger than the positive 32-bit integer value (2^{31} - 1). In the analysis of our case study, we have circumvented this situation by decomposing metapolicies into nodes so that they can be analysed separately. Alice's metapolicy, for example, consists of 10 nodes which range from 176 atoms to encode 18 states (nodes 0, 2, 7, 8, 9) to 215 atoms to encode 24 states (node 6). After running repeated tests, the graph of Figure 6.1 indicates that the running time is proportional to the number of metapolicy states and hence number of atoms. We conclude that, in terms of scalability, the Alloy Analyzer tool is not suitable for cases where the metapolicy state count exceeds 24, as the associated number of atoms in the Alloy model to support analysis will exceed the threshold value of 216. Since Alloy 4.0 runs atop the Kodkod relational model finder [134], the designers of Alloy have suggested directly using the Kodkod engine to deal with encoding example data (like metapolicies) although this may still be unsuitable.[1] As we move forward, future solvers will have to handle larger example data, which will improve on our scalability issues. In the next section, we investigate how one such solver can accomplish this task but is currently under development.

Nevertheless, we can analyse systems whose data source owners would like to use guidelines similar to our classes of requirements. The number of states in a metapolicy is related to the class of requirements captured and, to some extent, the number of resources. We recall that the *binary* class of requirements concerns the sole access of a resource. Therefore, if a system that we attempt to model has a large number of resources, then the state space of that metapolicy will be equally just as large since the system changes state each time a resource is accessed. For example, a *binary* instance requirement with seven resources (the images example), produces a state space with 15 states that can be analysed by our abstraction. However, a *binary* instance requirement with ten resources (the case study) produces a state space that is over the limit of 24 states (hence the need for decomposition into nodes). Nevertheless, other requirements can produce small analysable state spaces regardless of the number of resources—in our case study, the metapolicy of user Bob is just two states, while that of Charlie is 11 states, neither of which capture the *binary* requirement.

Our formal executable model of EAC is used to establish a proof-of-concept for our approach akin to the formal executable models of Table 5.2. In this respect, Alloy has worked, and, as we move forward, the language, our framework of functions and how we may check for properties and deploy policies will be essential in the design and development of tools that provide assurance that the secure sharing of data is appropriate.

[1] http://alloy.mit.edu/community/node/376

6.3 Future Work

In this section, we focus on three main areas in which additional research can extend the work of this thesis.

6.3.1 Gauge

Having tested the scalability of the Alloy Analyzer tool in this thesis, we concluded that other means of handling a larger number of states is necessary. Consequently, work has begun in this area with the development of the Gauge tool [6], a prototype evaluator which evaluates predicates and expressions written in the Alloy language. Gauge is in early stages of development and designed to handle large instances of real-world data.

As explained earlier, the Alloy Analyzer is restricted by the number of atoms used in the Alloy model as tuples in Kodkod are represented as positive 32-bit integers.[2] In our model, the MetaPolicy structure contains the transitions relation with an arity of 4, which imposes a limit of 216 total atoms. The Gauge tool enforces no such limit on instance size except for the memory needed to execute the model.

Because Alloy is typeless, the representation of certain types such as integers is limited, as in this case, the bitwidth can be restricted. This is a reasonable restriction when checking for counterexamples—however, real-world data is very likely to surpass this bitwidth making it almost intractable to model instances. Gauge is able to bypass these limitations by casting between Alloy atoms and native types when necessary.

Gauge can also dynamically extend instances to evaluate expressions outside the scope of the Alloy instance being evaluated such as time and dates. This can not be done using the Alloy Analyzer as it explores a finite universe and expects that all values necessary to evaluate expressions are present in the model.

Recursion using predicates and functions in Alloy is not possible. We encountered this issue when modelling the subsequence function, and had to unroll this function into several subroutines (Appendix C.7). The Gauge tool allows recursive predicates and functions.

Finally, although in the early stages of development, Gauge appears to be faster in evaluating model instances than the Alloy Analyzer. Importantly, there is the potential to check each new state—or policy—prior to deployment, rather than check all states upfront.

6.3.2 Metapolicy Language and Tool

In Example 3.1, we described a potential tool that can act as an interface to an EAC mechanism which policy writers can use to capture metapolicy requirements, check properties, and deploy policies accordingly. The aim of this thesis, however, is to develop an EAC model which serves as the theory behind such a tool and technology, and provides the underpinnings of its implementation. Thus, a formal representation of a metapolicy is constructed so that it

[2] http://alloy.mit.edu/community/node/551

can be analysed and verified for correctness before deployment. In moving forward, program refinement may be useful as we aim to convert our specification to an implementation. We recall that in program refinement, we start with an abstract specification of a system, and incrementally transform this into a concrete program executable via a sequence of correctness preserving transformations [131, 132]. Furthermore, refinement calculus is a formalised approach for such stepwise refinement because the intent is to ensure that the low-level behaviour correctly meets its high-level specification at every step [135]. The result of such a process forms the implementation behind a potential tool that can be used to ascertain the correctness of metapolicies and policies before deployment.

Ultimately, we want to provide policy creators with such a tool that allows them to easily capture high-level requirements such as legal and regulatory guidelines (written at a high level of abstraction) in an expressive and manageable representation without loss of meaning (for example, a domain-specific metapolicy language). As we move forward with this research, development of tool support for a domain-specific metapolicy language for the construction of metapolicies that can then be analysed before deployment of policies would be ideal (perhaps based on the Alloy language). Such a methodology can then be employed across a broad range of applications including social networking sites, medical information systems and commercial cloud-based services without having to rely on the constant supervision of administrators. Furthermore, with a suitably designed domain-specific metapolicy language, we can identify particular external events that should be logged to be factored into EAC decisions. In essence, with such tools, we can provide some assurance that higher-level data access requirements, which, as we noted, are becoming increasingly complex as a consequence of legislation, standards and policy guidelines, are truly captured at a lower-level with EAC policies.

Power et al. in [136] have developed a similar tool for constructing RBAC policies, which are dynamically checked for conformance using the Alloy language and Gauge. This work has been undertaken within the Service-Oriented Federated Authorization (SOFA) project.[3] The sif middleware has been further extended as part of SOFA to accommodate such tools and technologies, which aid in the construction of access control policies with the aim of securely networking disparate data sources without the dependence on a sole authorisation mechanism. Similarly, we seek to integrate our metapolicy language and tool with the sif middleware.

6.3.3 Automated Transformations

We believe that there is some need for a more accessible, conceptual model on top of Alloy. Originally, our EAC abstraction was specified using the Z formal language (see Appendix B). Z is a powerful, mathematical language that allows the specification of systems at a very high level, as documented in [137]. This has allowed the ability to reason about software modules, protocols and algorithms at an abstract level producing Z specifications that can act as blueprints for further design. However, Z has a limited number of stable tools for undertaking the kind of analysis that is necessary in this thesis. A Z typechecker and parser

[3] http://www.cs.ox.ac.uk/projects/SOFA/

[138, 139] can both be utilised to verify that specifications are syntactically correct, but semantic errors cannot be identified using these tools. Z theorem provers [140, 141] can also be employed to investigate interesting properties, but the use of these tools can be quite complicated rendering them inaccessible to many. Z animators also exist [142, 143, 144], but none of them have been sufficiently developed to match the capabilities of Alloy. As a result, we surmise that there should be a complete and automated translation from Z to Alloy as current manual translations have the potential to be error-prone. Research in this area has already commenced, as Malik et al. in [128] aim to make the automatic analysis and visualisation of Alloy accessible to users of Z. They have established a subset of Z that can be translated to Alloy, and plan to formally prove this translation. Furthermore, Malik et al. are currently developing a Z to Alloy translation tool under the Community Z Tools (CZT) project [139]. This tool can already specify simple constructs and schemas but is yet to handle complex Z sets and relations (sets of sets). However, translation tools from Z to other formal notations exist. In [145, 146], Z has been translated to the Symbolic Analysis Laboratory (SAL) language [147] so as to leverage its suite of verification tools. Moreover, in [148], Z specifications have also been translated to the B method [114], and validated using the ProB tool [149]. Nevertheless, as Malik et al. have speculated in [128], the visual feedback supported by the Alloy Analyzer is sufficiently invaluable so as to warrant another translation from Z.

Another transformation that should be proved and automated is the translation from our version of EAC policies to XACML. In [117], XACML policies have been translated to Alloy for verification of properties, but, in the context of our thesis, we need proof that the reverse translation holds. We have formally specified this translation using Z in Appendix B.7, but future research will prove and automate this translation, perhaps using the Alloy language once the Z to Alloy translation has been completed.

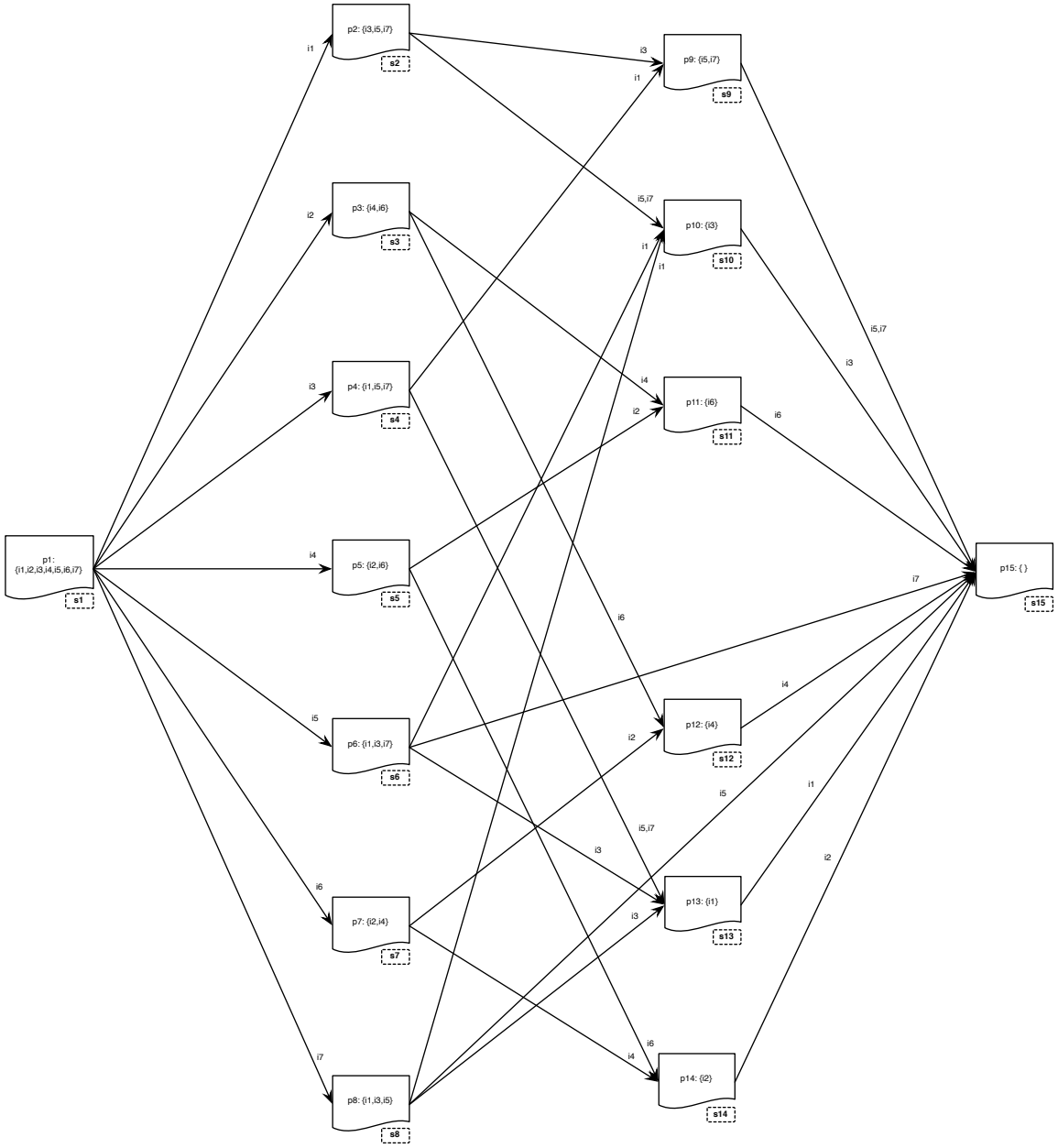
6.4 Summary

In this thesis, we presented our argument for a formal model of EAC. At the beginning, we cited that the motivation for the development of EAC was due largely in part to the need for effective tools to support secure data management. Then, having laid the foundation for our research, and with the use of formal methods and associated tools, we set out to establish confidence that our EAC model functions as intended. Our research methodology involved the use of the Alloy language and tool to construct and analyse metapolicies, and verify and deploy policies. Moreover, our model was used to develop algorithms that trigger a change of state and automatically evolve policies based on conditional events. Finally, a small yet sufficiently complex case study was employed to validate our approach.

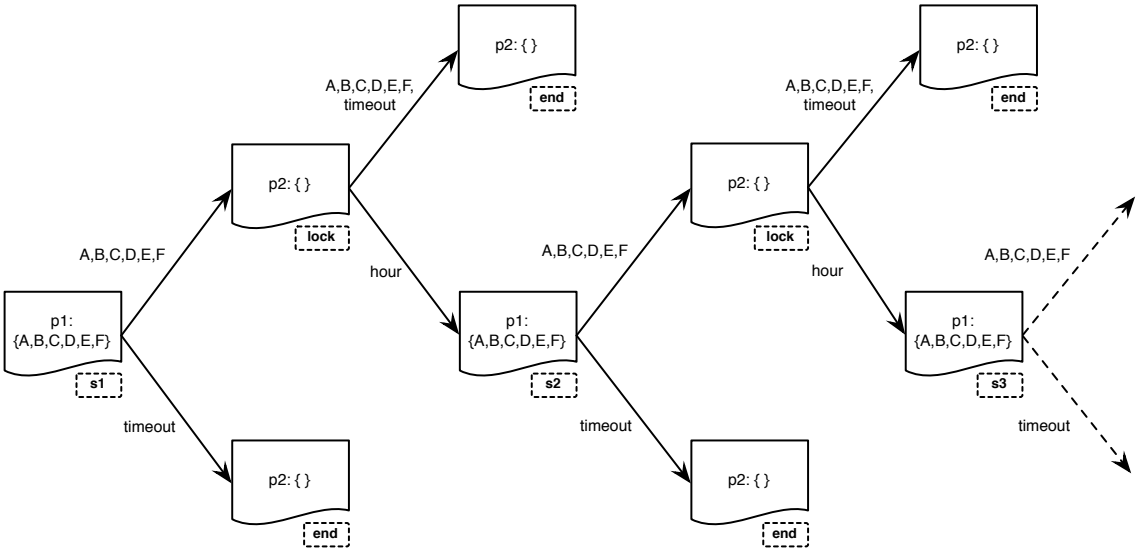
In light of this, we believe that the research question has been answered, and that we have successfully developed a formal model for the construction, analysis, and automatic deployment of EAC policies. In conclusion, we put forward the claim that we have provided assurance that our approach to access control has significant potential to support the appropriate sharing of data.

Appendix A: Example State Spaces

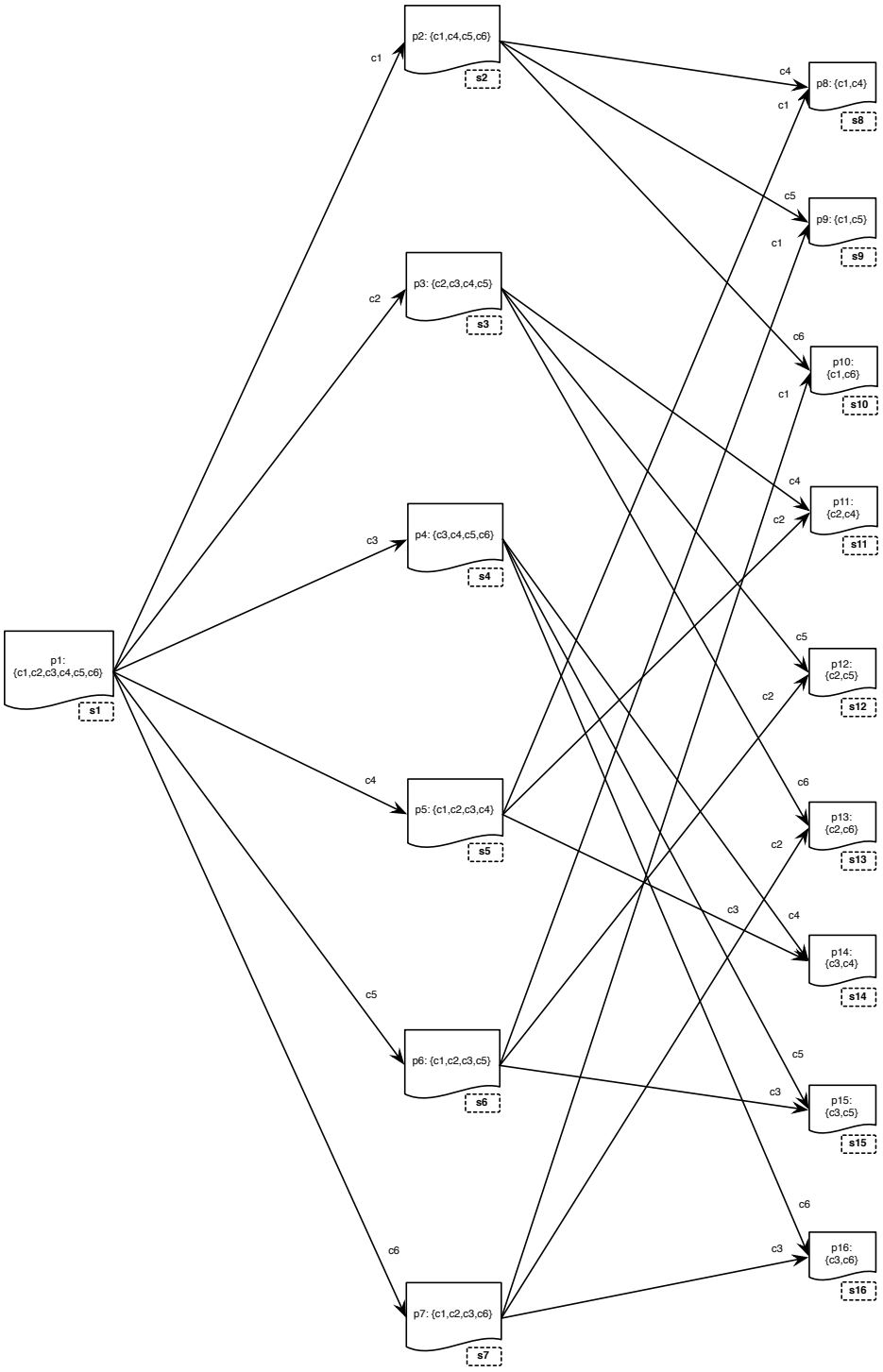
A.1 Images Example State Space



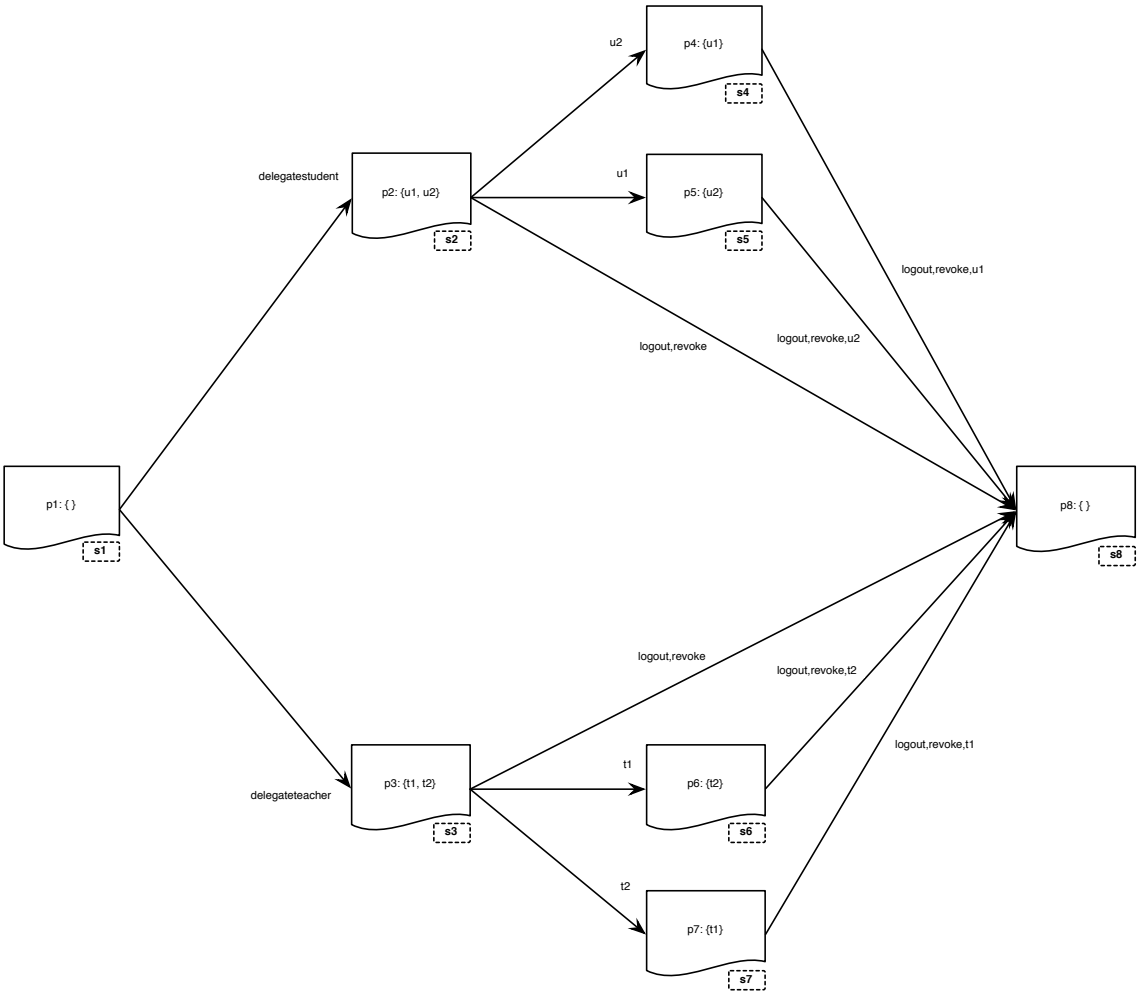
A.2 Temporal Example State Space



A.3 Compartment Example State Space



A.4 Delegation Example State Space



Appendix B: Z model

Our model was initially specified using the Z formal language [113]. The following is an outline of the original specification which aided development of our Alloy model—we note the close similarity of both models. The fuzz tool [138] was used to type check the Z model.

B.1 Rules, Policies and MetaPolicies

Definition B.1. *(EAC Identifiers)*

$$[StateID, MPConditionID, RuleID, PolicyID, MetaPolicyID]$$
$$[Subject, Resource, Action]$$
$$Effect ::= Permit \mid Deny \mid NotApplicable \mid Indeterminate$$

Definition B.2. *(EAC Rule)*

EACRule

$rid : RuleID$
$subject : Subject$
$resource : Resource$
$effect : Effect$
$action : Action$

Definition B.3. *(EAC Policy)*

EACPolicy

$pid : PolicyID$
$rules : \mathrm{iseq}\ RuleID$

Definition B.4. *(EAC MetaPolicy)*

$$
\begin{array}{|l}
\textit{MetaPolicy} \\
\hline
mid : MetaPolicyID \\
initialstate : StateID \\
rule : RuleID \pfun EACRule \\
policy : PolicyID \pfun EACPolicy \\
states : StateID \pfun PolicyID \\
transitions : StateID \pfun \mathsf{seq}\,(MPConditionID \times StateID) \\
\hline
initialstate \in \mathsf{dom}\ states \\
\forall s : \mathsf{ran}\,(\mathsf{ran}\ transitions) \bullet second\ s \in \mathsf{dom}\ states \\
\forall r : \mathsf{dom}\ rule \bullet (rule\, r).rid = r \\
\forall p : \mathsf{dom}\ policy \bullet (policy\, p).pid = p \\
\forall i : \mathsf{ran}\ states \bullet i \in \mathsf{dom}\ policy \\
\forall j : \mathsf{ran}\ policy \bullet \forall k : \mathsf{ran}\ j.rules \bullet k \in \mathsf{dom}\ rule \\
\hline
\end{array}
$$

B.2 Audit Data

Once a subject has accessed resources (successfully or unsuccessfully), all such data is sequentially logged to an audit data source in atomic units. *ADData* formally describes these internal events as storage units, indicating the result of who performed which action on what.

Definition B.5. *(Internal Event)*

$$
\begin{array}{|l}
\textit{ADData} \\
\hline
subject : Subject \\
resource : Resource \\
action : Action \\
result : Result \\
\hline
\end{array}
$$

The *Result* definition indicates the success of this access—accepted or rejected.

Definition B.6. *(Result)*

$$Result ::= Accept \mid Reject$$

Definition B.7. *(External Event)*

$$[External]$$

An *Event* is a free type collection of *external* and *internal* events.

Definition B.8. *(Event)*

$$Event ::= external\ \langle\langle External \rangle\rangle \mid internal\ \langle\langle ADData \rangle\rangle$$

A timestamp is represented as natural numbers.

Definition B.9. *(Timestamp)*

$$Timestamp == \mathbb{N}$$

The audit data source is an injective sequence of ($Event$, $Timestamp$) pairs.

Definition B.10. *(Audit Data Source)*

$$\begin{array}{|l} ADDB \\ \hline data : \mathrm{iseq}\,(Event \times Timestamp) \\ \hline \end{array}$$

An initial audit data source can be defined with the following initialisation schema.

Definition B.11. *(Initial Audit Data Source)*

$$\begin{array}{|l} ADDBInit \\ \hline ADDB' \\ \hline data' = \langle\rangle \\ \hline \end{array}$$

An Algebra of Operations

This algebra of operations is used to access data from our audit data source. With this in mind, we present our algebra of operations in the following fashion. We first define a function and provide an example of its use. The function is then embedded within an operation schema that directly manipulates the audit data. For the sake of simplicity, the following $ADDB$ audit data source has been instantiated with an example data set which reinforces how the function works.

This data set mimics the audit history of an online social networking application with profiles, groups and photo albums (with no external events).

$$\begin{array}{|l} AProfileID, LProfileID, SProfileID, VProfileID, WProfileID : Subject \\ APhotoID, JEventID, LPhotoID, LGroupID, SPhotoID, VPhotoID : Resource \\ view : Action \\ \hline \end{array}$$

$$\begin{array}{|l} addb : ADDB \\ \hline addb = (\!| data == \\ \langle (internal (\!| subject == VProfileID, resource == LPhotoID, action == view, result == Accept |\!), 100), \\ (internal (\!| subject == VProfileID, resource == LGroupID, action == view, result == Accept |\!), 115), \\ (internal (\!| subject == AProfileID, resource == SPhotoID, action == view, result == Accept |\!), 120), \\ (internal (\!| subject == LProfileID, resource == APhotoID, action == view, result == Reject |\!), 200), \\ (internal (\!| subject == WProfileID, resource == VPhotoID, action == view, result == Reject |\!), 300), \\ (internal (\!| subject == SProfileID, resource == JEventID, action == view, result == Reject |\!), 315) \rangle |\!) \\ \hline \end{array}$$

Basic Access Operations

The first function adds an (*Event*, *Timestamp*) pair to *ADDB*. An example of its use follows the definition, and then the function is embedded in the corresponding operation.

Definition B.12. *(Add an Entry)*

$$Add : (Event \times Timestamp) \times \mathrm{iseq}\,(Event \times Timestamp) \pfun \mathrm{iseq}\,(Event \times Timestamp)$$

$$\forall a : (Event \times Timestamp); s : \mathrm{iseq}\,(Event \times Timestamp) \mid a \notin \mathrm{ran}\, s \bullet Add\,(a, s) = s \frown \langle a \rangle$$

$$e : ADData$$

$$e.subject = WProfileID \land e.resource = JEventID \land e.action = view \land e.result = Accept$$

$$\begin{aligned}
&Add\,((e, 400), addb) = \\
&\langle (internal\,(\!|\, subject == VProfileID, resource == LPhotoID, action == view, result == Accept \,|\!), 100), \\
&\ (internal\,(\!|\, subject == VProfileID, resource == LGroupID, action == view, result == Accept \,|\!), 115), \\
&\ (internal\,(\!|\, subject == AProfileID, resource == SPhotoID, action == view, result == Accept \,|\!), 120), \\
&\ (internal\,(\!|\, subject == LProfileID, resource == APhotoID, action == view, result == Reject \,|\!), 200), \\
&\ (internal\,(\!|\, subject == WProfileID, resource == VPhotoID, action == view, result == Reject \,|\!), 300), \\
&\ (internal\,(\!|\, subject == SProfileID, resource == JEventID, action == view, result == Reject \,|\!), 315), \\
&\ (internal\,(\!|\, subject == WProfileID, resource == JEventID, action == view, result == Accept \,|\!), 400) \rangle
\end{aligned}$$

ADDBAdd

$$\Delta ADDB$$
$$a? : Event$$
$$t? : Timestamp$$
$$data' = Add\,((a?, t?), data)$$

Next, the *Delete* function removes the first entry in the *data* sequence of *ADDB*.

Definition B.13. *(Delete first Entry)*

$$Delete : \mathrm{iseq}\,(Event \times Timestamp) \pfun \mathrm{iseq}\,(Event \times Timestamp)$$

$$\forall s : \mathrm{iseq}\,(Event \times Timestamp) \mid s \neq \langle\rangle \bullet Delete\,(s) = tail\ s$$

$$\begin{aligned}
&Delete\,(addb) = \\
&\langle (internal\,(\!|\, subject == VProfileID, resource == LGroupID, action == view, result == Accept \,|\!), 115), \\
&\ (internal\,(\!|\, subject == AProfileID, resource == SPhotoID, action == view, result == Accept \,|\!), 120), \\
&\ (internal\,(\!|\, subject == LProfileID, resource == APhotoID, action == view, result == Reject \,|\!), 200), \\
&\ (internal\,(\!|\, subject == WProfileID, resource == VPhotoID, action == view, result == Reject \,|\!), 300), \\
&\ (internal\,(\!|\, subject == SProfileID, resource == JEventID, action == view, result == Reject \,|\!), 315) \rangle
\end{aligned}$$

ADDBDelete

$$\Delta ADDB$$
$$data \neq \langle\rangle \Rightarrow data' = Delete\,(data)$$
$$data = \langle\rangle \Rightarrow data' = \langle\rangle$$

DeletePos deletes an *Event* at position n. The *squash* function takes a finite function defined upon natural numbers and returns a sequence—it "squashes" the domain to eliminate any spaces created by range restriction, while maintaining the order of remaining maplets [113].

Definition B.14. *(Delete Entry at Position)*

$$DeletePos : \mathbb{N} \times \mathsf{iseq}\,(Event \times Timestamp) \pfun \mathsf{iseq}\,(Event \times Timestamp)$$

$$\forall s : \mathsf{iseq}\,(Event \times Timestamp); n : \mathbb{N} \mid n \in \mathsf{dom}\, s \bullet DeletePos\,(n, s) = squash\,(\{n\} \ndres s)$$

$$\begin{aligned}&DeletePos\,(4, addb) = \\ &\langle (internal\,\langle subject == VProfileID, resource == LPhotoID, action == view, result == Accept\rangle, 100),\\ &\ (internal\,\langle subject == VProfileID, resource == LGroupID, action == view, result == Accept\rangle, 115),\\ &\ (internal\,\langle subject == AProfileID, resource == SPhotoID, action == view, result == Accept\rangle, 120),\\ &\ (internal\,\langle subject == WProfileID, resource == VPhotoID, action == view, result == Reject\rangle, 300),\\ &\ (internal\,\langle subject == SProfileID, resource == JEventID, action == view, result == Reject\rangle, 315)\rangle\end{aligned}$$

ADDBDeletePos

$$\Delta ADDB$$
$$n? : \mathbb{N}$$
$$data \neq \langle\rangle \land 1 \leq n? \leq \#data \Rightarrow data' = DeletePos\,(n?, data)$$
$$data = \langle\rangle \lor \neg\,(1 \leq n? \leq \#data) \Rightarrow data' = data$$

Similarly, *DeleteInterval* deletes data in an interval, specified by two natural numbers, n and m. It also uses the *squash* function to eliminate any spaces created by range restriction, and returns the compacted sequence of (*Event*, *Timestamp*) pairs.

Definition B.15. *(Delete Entries in an Interval)*

$$DeleteInterval : \mathbb{N} \times \mathbb{N} \times \mathsf{iseq}\,(Event \times Timestamp) \pfun \mathsf{iseq}\,(Event \times Timestamp)$$

$$\forall s : \mathsf{iseq}\,(Event \times Timestamp); n, m : \mathbb{N} \mid n\,..\,m \subseteq \mathsf{dom}\, s \bullet DeleteInterval\,(n, m, s) = squash\,(n\,..\,m \ndres s)$$

$$\begin{aligned}&DeleteInterval\,(2, 4, addb) = \\ &\langle (internal\,\langle subject == VProfileID, resource == LPhotoID, action == view, result == Accept\rangle, 100),\\ &\ (internal\,\langle subject == WProfileID, resource == VPhotoID, action == view, result == Reject\rangle, 300),\\ &\ (internal\,\langle subject == SProfileID, resource == JEventID, action == view, result == Reject\rangle, 315)\rangle\end{aligned}$$

ADDBDeleteInterval

$$\Delta ADDB$$
$$n?, m? : \mathbb{N}$$
$$(data \neq \langle\rangle \land n?\,..\,m? \subseteq \mathsf{dom}\, data \Rightarrow data' = DeleteInterval\,(n?, m?, data))$$
$$(data = \langle\rangle \lor \neg\,(n?\,..\,m? \subseteq \mathsf{dom}\, data)) \Rightarrow data' = data$$

GetPos retrieves the audit entry given at position n.

Definition B.16. *(Get Entry at Position)*

$$GetPos : \mathbb{N} \times \mathsf{iseq}\,(Event \times Timestamp) \pfun (Event \times Timestamp)$$

$$\forall s : \mathsf{iseq}\,(Event \times Timestamp); n : \mathbb{N} \mid n \in \mathsf{dom}\, s \bullet GetPos\,(n, s) = s\, n$$

$$GetPos\,(5, addb) = (internal\langle\!|\, subject == WProfileID, resource == VPhotoID, action == view, result == Reject \,|\!\rangle, 300)$$

ADDBGetPos

$$\Xi ADDB$$
$$n? : \mathbb{N}$$
$$d! : Event \times Timestamp$$

$$1 \leq n? \leq \#data$$
$$d! = GetPos\,(n?, data)$$

SubString retrieves the contiguous sequence of data between two positions—specified by two natural numbers, n and m.

Definition B.17. *(Get Entries in an Interval)*

$$SubString : \mathbb{N} \times \mathbb{N} \times \mathsf{iseq}\,(Event \times Timestamp) \pfun \mathsf{iseq}\,(Event \times Timestamp)$$

$$\forall s : \mathsf{iseq}\,(Event \times Timestamp); n, m : \mathbb{N} \mid n\,..\,m \subseteq \mathsf{dom}\, s \bullet SubString\,(n, m, s) = squash\,((n\,..\,m) \lhd s)$$

$$SubString\,(2, 5, addb) = \langle (internal\langle\!|\, subject == VProfileID, resource == LGroupID, action == view, result == Accept \,|\!\rangle, 115),$$
$$(internal\langle\!|\, subject == AProfileID, resource == SPhotoID, action == view, result == Accept \,|\!\rangle, 120),$$
$$(internal\langle\!|\, subject == LProfileID, resource == APhotoID, action == view, result == Reject \,|\!\rangle, 200),$$
$$(internal\langle\!|\, subject == WProfileID, resource == VPhotoID, action == view, result == Reject \,|\!\rangle, 300),$$
$$(internal\langle\!|\, subject == SProfileID, resource == JEventID, action == view, result == Reject \,|\!\rangle, 315)\rangle$$

ADDBSubString

$$\Xi ADDB$$
$$n? : \mathbb{N}$$
$$m? : \mathbb{N}$$
$$a! : \mathsf{iseq}\,(Event \times Timestamp)$$

$$1 \leq n? \leq m? \leq \#data$$
$$a! = SubString\,(n?, m?, data)$$

As the name implies, the next two operations sort the entries in ascending and descending order according to the associated timestamps of events.

Definition B.18. *(Sort Entries in Ascending Time)*

$$
\begin{array}{|l}
SortTimeAscending : \mathsf{iseq}\,(Event \times Timestamp) \pfun \mathsf{iseq}\,(Event \times Timestamp) \\
\hline
\forall s : \mathsf{iseq}\,(Event \times Timestamp) \bullet \\
\quad \mathsf{dom}\,(SortTimeAscending\, s) = \mathsf{dom}\, s \land \mathsf{ran}\,(SortTimeAscending\, s) = \mathsf{ran}\, s \land \\
\quad\quad (\forall i, j : \mathsf{dom}\,(SortTimeAscending\, s) \mid i < j \bullet \\
\quad\quad\quad second\,((SortTimeAscending\, s)\, i) \leq second\,((SortTimeAscending\, s)\, j))
\end{array}
$$

$$
\begin{array}{l}
SortTimeAscending\,(addb) = \\
\langle (internal\,\langle\!|\, subject == VProfileID, resource == LPhotoID, action == view, result == Accept \,|\!\rangle, 100), \\
\ (internal\,\langle\!|\, subject == VProfileID, resource == LGroupID, action == view, result == Accept \,|\!\rangle, 115), \\
\ (internal\,\langle\!|\, subject == AProfileID, resource == SPhotoID, action == view, result == Accept \,|\!\rangle, 120), \\
\ (internal\,\langle\!|\, subject == LProfileID, resource == APhotoID, action == view, result == Reject \,|\!\rangle, 200), \\
\ (internal\,\langle\!|\, subject == WProfileID, resource == VPhotoID, action == view, result == Reject \,|\!\rangle, 300), \\
\ (internal\,\langle\!|\, subject == SProfileID, resource == JEventID, action == view, result == Reject \,|\!\rangle, 315)\rangle
\end{array}
$$

$$
\begin{array}{|l}
\hline
ADDBSortTimeAscending \\
\hline
\Delta ADDB \\
\hline
data' = SortTimeAscending\,(data) \\
\hline
\end{array}
$$

Definition B.19. *(Sort Entries in Descending Time)*

$$
\begin{array}{|l}
SortTimeDescending : \mathsf{iseq}\,(Event \times Timestamp) \pfun \mathsf{iseq}\,(Event \times Timestamp) \\
\hline
\forall s : \mathsf{iseq}\,(Event \times Timestamp) \bullet \\
\quad \mathsf{dom}\,(SortTimeDescending\, s) = \mathsf{dom}\, s \land \mathsf{ran}\,(SortTimeDescending\, s) = \mathsf{ran}\, s \land \\
\quad\quad (\forall i, j : \mathsf{dom}\,(SortTimeDescending\, s) \mid i < j \bullet \\
\quad\quad\quad second\,((SortTimeDescending\, s)\, i) \geq second\,((SortTimeDescending\, s)\, j))
\end{array}
$$

$$
\begin{array}{l}
SortTimeDescending\,(addb) = \\
\langle (internal\,\langle\!|\, subject == SProfileID, resource == JEventID, action == view, result == Reject \,|\!\rangle, 315), \\
\ (internal\,\langle\!|\, subject == WProfileID, resource == VPhotoID, action == view, result == Reject \,|\!\rangle, 300), \\
\ (internal\,\langle\!|\, subject == LProfileID, resource == APhotoID, action == view, result == Reject \,|\!\rangle, 200), \\
\ (internal\,\langle\!|\, subject == AProfileID, resource == SPhotoID, action == view, result == Accept \,|\!\rangle, 120), \\
\ (internal\,\langle\!|\, subject == VProfileID, resource == LGroupID, action == view, result == Accept \,|\!\rangle, 115), \\
\ (internal\,\langle\!|\, subject == VProfileID, resource == LPhotoID, action == view, result == Accept \,|\!\rangle, 100)\rangle
\end{array}
$$

$$
\begin{array}{|l}
\hline
ADDBSortTimeDescending \\
\hline
\Delta ADDB \\
\hline
data' = SortTimeDescending\,(data) \\
\hline
\end{array}
$$

In the next section, we move on to access operations that filter data based on a particular *Subject*, *Resource*, *Result* or *Timestamp*.

Intermediate Access Operations

The following two operations return results based on all accesses, accepted or rejected, operating on internal events only.

Definition B.20. *(Return all Accept Entries)*

$$DBAccept : \mathsf{iseq}\,(Event \times Timestamp) \pfun \mathsf{iseq}\,(Event \times Timestamp)$$

$$\forall s : \mathsf{iseq}\,(Event \times Timestamp) \bullet$$
$$DBAccept\,(s) = s \rhd \{\, a : Event; t : Timestamp \mid$$
$$(\exists i : ADData \bullet a = internal\,(i) \land i.result = Accept)\,\}$$

$$DBAccept\,(addb) =$$
$$\langle (internal \lbag subject == VProfileID, resource == LPhotoID, action == view, result == Accept \rbag, 100),$$
$$(internal \lbag subject == VProfileID, resource == LGroupID, action == view, result == Accept \rbag, 115),$$
$$(internal \lbag subject == AProfileID, resource == SPhotoID, action == view, result == Accept \rbag, 120) \rangle$$

ADDBAccept

$$\Xi ADDB$$
$$a! : \mathsf{iseq}\,(Event \times Timestamp)$$

$$a! = DBAccept\,(data)$$

Definition B.21. *(Return all Reject Entries)*

$$DBReject : \mathsf{iseq}\,(Event \times Timestamp) \pfun \mathsf{iseq}\,(Event \times Timestamp)$$

$$\forall s : \mathsf{iseq}\,(Event \times Timestamp) \bullet$$
$$DBReject\,(s) = s \rhd \{\, a : Event; t : Timestamp \mid$$
$$(\exists i : ADData \bullet a = internal\,(i) \land i.result = Reject)\,\}$$

$$DBReject\,(addb) =$$
$$\langle (internal \lbag subject == LProfileID, resource == APhotoID, action == view, result == Reject \rbag, 200),$$
$$(internal \lbag subject == WProfileID, resource == VPhotoID, action == view, result == Reject \rbag, 300),$$
$$(internal \lbag subject == SProfileID, resource = JEventID, action == view, result == Reject \rbag, 315) \rangle$$

ADDBReject

$$\Xi ADDB$$
$$a! : \mathsf{iseq}\,(Event \times Timestamp)$$

$$a! = DBReject\,(data)$$

This operation returns all the accesses pertaining to a specific *Subject*.

Definition B.22. *(Return all Entries for Subject)*

$$SubjectAccess : Subject \times \mathsf{iseq}\,(Event \times Timestamp) \pfun \mathsf{iseq}\,(Event \times Timestamp)$$

$$\forall s : \mathsf{iseq}\,(Event \times Timestamp); p : Subject \bullet$$
$$SubjectAccess(p, s) = s \rhd \{ a : Event; t : Timestamp \mid$$
$$(\exists i : ADData \bullet a = internal\,(i) \land i.subject = p) \}$$

$$SubjectAccess\,(VProfileID, addb) =$$
$$\langle (internal \langle\!| subject == VProfileID, resource == LPhotoID, action == view, result == Accept |\!\rangle, 100),$$
$$(internal \langle\!| subject == VProfileID, resource == LGroupID, action == view, result == Accept |\!\rangle, 115) \rangle$$

ADDBSubjectAccess

$$\Xi ADDB$$
$$subject? : Subject$$
$$a! : \mathsf{iseq}\,(Event \times Timestamp)$$
$$a! = SubjectAccess\,(subject?, data)$$

Similarly, this operation returns all accesses pertaining to a specific *Resource*.

Definition B.23. *(Return all Entries for Resource)*

$$ResourceAccess : Resource \times \mathsf{iseq}\,(Event \times Timestamp) \pfun \mathsf{iseq}\,(Event \times Timestamp)$$

$$\forall s : \mathsf{iseq}\,(Event \times Timestamp); r : Resource \bullet$$
$$ResourceAccess(r, s) = s \rhd \{ a : Event; t : Timestamp \mid$$
$$(\exists i : ADData \bullet a = internal\,(i) \land i.resource = r) \}$$

$$ResourceAccess\,(APhotoID, addb) =$$
$$\langle (internal \langle\!| subject == LProfileID, resource == APhotoID, action == view, result == Reject |\!\rangle, 200) \rangle$$

ADDBResourceAccess

$$\Xi ADDB$$
$$resource? : Resource$$
$$a! : \mathsf{iseq}\,(Event \times Timestamp)$$
$$a! = ResourceAccess\,(resource?, data)$$

Finally, the following operation returns all accesses within a certain time period.

Definition B.24. *(Return all Entries in Time Interval)*

$$TimePeriodAccess : Timestamp \times Timestamp \times \mathsf{iseq}\,(Event \times Timestamp) \pfun \mathsf{iseq}\,(Event \times Timestamp)$$

$$\forall s : \mathsf{iseq}\,(Event \times Timestamp); n, m : Timestamp \bullet TimePeriodAccess\,(n, m, s) = s \rhd \{ a : Event; t : Timestamp \mid n \leq t \leq m \}$$

$$TimePeriodAccess\,(300, 400, addb) = \langle (internal\,(\!| subject == WProfileID, resource == VPhotoID, action == view, result == Reject |\!), 300), (internal\,(\!| subject == SProfileID, resource == JEventID, action == view, result == Reject |\!), 315) \rangle$$

ADDBTimePeriodAccess

$$\Xi ADDB$$
$$timebegin? : Timestamp$$
$$timeend? : Timestamp$$
$$a! : \mathsf{iseq}\,(Event \times Timestamp)$$
$$a! = TimePeriodAccess\,(timebegin?, timeend?, data)$$

In the next section, we combine these operations to form advanced access operations.

Advanced Access Operations

The following are examples of complex operations that can be defined by composing the functions introduced in the previous subsections. By combining these functions in this manner, we can construct advanced access operations that return the relevant data necessary to evolve policies or capture requirements.

The first two operations return all accepted accesses or all rejected accesses for a particular *Subject*.

Definition B.25. *(Return all Accept Entries for Subject)*

$$SubjectAccessAccepted : Subject \times \mathsf{iseq}\,(Event \times Timestamp) \pfun \mathsf{iseq}\,(Event \times Timestamp)$$

$$SubjectAccessAccepted = SubjectAccess \fatsemi DBAccept$$

$$SubjectAccessAccepted\,(AProfileID, addb) = \langle (internal\,(\!| subject == AProfileID, resource == SPhotoID, action == view, result == Accept |\!), 120) \rangle$$

ADDBSubjectAccessAccepted

$$\Xi ADDB$$
$$subject? : Subject$$
$$a! : \mathsf{iseq}\,(Event \times Timestamp)$$
$$a! = SubjectAccessAccepted\,(subject?, data)$$

Definition B.26. *(Return all Reject Entries for Subject)*

$$SubjectAccessRejected : Subject \times \mathsf{iseq}\,(Event \times Timestamp) \pfun \mathsf{iseq}\,(Event \times Timestamp)$$

$$SubjectAccessRejected = SubjectAccess \fatsemi DBReject$$

$$SubjectAccessRejected\,(LProfileID, addb) =$$
$$\langle (internal \lbag subject == LProfileID, resource == APhotoID, action == view, result == Reject \rbag, 200) \rangle$$

ADDBSubjectAccessRejected

$$\Xi ADDB$$
$$subject? : Subject$$
$$a! : \mathsf{iseq}\,(Event \times Timestamp)$$
$$a! = SubjectAccessRejected\,(subject?, data)$$

The next two operations return all accepted accesses or all rejected accesses for a particular *Resource*.

Definition B.27. *(Return all Accept Entries for Resource)*

$$ResourceAccessAccepted : Resource \times \mathsf{iseq}\,(Event \times Timestamp) \pfun \mathsf{iseq}\,(Event \times Timestamp)$$

$$ResourceAccessAccepted = ResourceAccess \fatsemi DBAccept$$

$$ResourceAccessAccepted\,(LGroupID, addb) =$$
$$\langle (internal \lbag subject == VProfileID, resource == LGroupID, action == view, result == Accept \rbag, 115) \rangle$$

ADDBResourceAccessAccepted

$$\Xi ADDB$$
$$resource? : Resource$$
$$a! : \mathsf{iseq}\,(Event \times Timestamp)$$
$$a! = ResourceAccessAccepted\,(resource?, data)$$

Definition B.28. *(Return all Reject Entries for Resource)*

$$ResourceAccessRejected : Resource \times \mathsf{iseq}\,(Event \times Timestamp) \pfun \mathsf{iseq}\,(Event \times Timestamp)$$

$$ResourceAccessRejected = ResourceAccess \fatsemi DBReject$$

$$ResourceAccessRejected\,(VProfileID, addb) =$$
$$\langle (internal \lbag subject == WProfileID, resource == VPhotoID, action == view, result == Reject \rbag, 300) \rangle$$

ADDBResourceAccessRejected

$$\Xi ADDB$$
$$resource? : Resource$$
$$a! : \mathsf{iseq}\,(Event \times Timestamp)$$
$$a! = ResourceAccessRejected\,(resource?, data)$$

The final two operations return all accepted accesses or all rejected accesses within a given time period.

Definition B.29. *(Return all Accept Entries in Time Interval)*

$$TimePeriodAccessAccepted : Timestamp \times Timestamp \times \mathsf{iseq}\,(Event \times Timestamp) \pfun \mathsf{iseq}\,(Event \times Timestamp)$$

$$TimePeriodAccessAccepted = TimePeriodAccess \fatsemi DBAccept$$

$$TimePeriodAccessAccepted\,(100, 200, addb) =$$
$$\langle (internal\,\langle\!|\, subject == VProfileID, resource == LPhotoID, action == view, result == Accept \,|\!\rangle, 100),$$
$$(internal\,\langle\!|\, subject == VProfileID, resource == LGroupID, action == view, result == Accept \,|\!\rangle, 115),$$
$$(internal\,\langle\!|\, subject == AProfileID, resource == SPhotoID, action == view, result == Accept \,|\!\rangle, 120)\rangle$$

ADDBTimePeriodAccessAccepted

$$\Xi ADDB$$
$$timebegin? : Timestamp$$
$$timeend? : Timestamp$$
$$a! : \mathsf{iseq}\,(Event \times Timestamp)$$
$$a! = TimePeriodAccessAccepted\,(timebegin?, timeend?, data)$$

Definition B.30. *(Return all Reject Entries in Time Interval)*

$$TimePeriodAccessRejected : Timestamp \times Timestamp \times \mathsf{iseq}\,(Event \times Timestamp) \pfun \mathsf{iseq}\,(Event \times Timestamp)$$

$$TimePeriodAccessRejected = TimePeriodAccess \fatsemi DBReject$$

$$TimePeriodAccessRejected\,(200, 400, addb) =$$
$$\langle (internal\,\langle\!|\, subject == LProfileID, resource == APhotoID, action == view, result == Reject \,|\!\rangle, 200),$$
$$(internal\,\langle\!|\, subject == WProfileID, resource == VPhotoID, action == view, result == Reject \,|\!\rangle, 300),$$
$$(internal\,\langle\!|\, subject == SProfileID, resource == JEventID, action == view, result == Reject \,|\!\rangle, 315)\rangle$$

ADDBTimePeriodAccessRejected

$$\Xi ADDB$$
$$timebegin? : Timestamp$$
$$timeend? : Timestamp$$
$$a! : \mathsf{iseq}\,(Event \times Timestamp)$$
$$a! = TimePeriodAccessRejected\,(timebegin?, timeend?, data)$$

Having described an algebra of operations on the audit data source via embedded functions and corresponding examples, we progress to the next section on conditional events that can trigger changes of state.

B.3 Conditional Events

The *condition* function associates an *MPConditionID* with a sequence of *Event* elements.

Definition B.31. *(Condition)*

$$condition : MPConditionID \pfun \mathbb{P}(\mathrm{seq}_1\ Event)$$

The following internal event can cause a change of state if found in *ADDB*.

$$a : ADData$$
$$\vdash a.subject = bob \land a.resource = o1 \land a.action = read \land a.result = Accept$$

$$condition(mc1) = \{\langle internal(a)\rangle\}$$

With external events, the *condition* function maps a *MPConditionID* identifier to singleton "trigger" sequences. Similarly, if located in *ADDB*, a change of state is signalled.

$$network\_exceed : External$$

$$condition(mc) = \{\langle external(network\_exceed)\rangle\}$$

B.4 Evolving the Policy

This function evolves the state of the system by searching for patterns (or "trigger" sequences) in the audit data source. The state evolves to a new policy according to which pattern has been located and returns an *EvalRes* as a signal.

Definition B.32. *(Evaluation Result)*

$$EvalRes ::= True \mid False$$

Definition B.33. *(State-Evolving Function)*

$$evaluateMPC : MPConditionID \times \mathrm{seq}(Event \times Timestamp) \longrightarrow EvalRes$$
$$\vdash \forall c : MPConditionID; d : \mathrm{seq}(Event \times Timestamp) \bullet$$
$$evaluateMPC(c,d) = True \Leftrightarrow$$
$$\exists s : condition(c) \bullet$$
$$subsequence(s, eventsequence(d)) = True$$

Two helper functions support *evaluateMPC*. The first, *eventsequence*, creates a sequence of just *Event* elements by extracting the first element from (*Event*, *Timestamp*) pairs in the current sequence of accesses.

Definition B.34. *(Create Event Sequence)*

$$
\begin{array}{|l}
\hline
eventsequence : \mathsf{seq}\,(Event \times Timestamp) \longrightarrow \mathsf{seq}\; Event \\
\hline
eventsequence\,(\langle\rangle) = \langle\rangle \\
\forall\, s : \mathsf{seq}\,(Event \times Timestamp);\ x : Event \times Timestamp \bullet \\
\qquad eventsequence\,(\langle x\rangle \frown s) = \langle first\,(x)\rangle \frown eventsequence\,(s) \\
\hline
\end{array}
$$

The second function, *subsequence*, is recursive and used to determine if a "trigger" sequence is a contiguous or non-contiguous subsequence of the current sequence of accesses in the audit data source.

Definition B.35. *(Subsequence)*

$$
\begin{array}{|l}
\hline
subsequence : \mathsf{seq}\; Event \times \mathsf{seq}\; Event \longrightarrow EvalRes \\
\hline
\forall\, trigger, data : \mathsf{seq}\; Event \bullet \\
\qquad trigger = \langle\rangle \Rightarrow \\
\qquad\qquad subsequence\,(trigger, data) = True \wedge \\
\qquad trigger \neq \langle\rangle \wedge data = \langle\rangle \Rightarrow \\
\qquad\qquad subsequence\,(trigger, data) = False \wedge \\
\qquad last\ trigger = last\ data \Rightarrow \\
\qquad\qquad subsequence\,(front\ trigger, front\ data) \wedge \\
\qquad last\ trigger \neq last\ data \Rightarrow \\
\qquad\qquad subsequence\,(trigger, front\ data) \\
\hline
\end{array}
$$

An EAC system can be encompassed in the following schema.

Definition B.36. *(System)*

$$
\begin{array}{|l}
\hline
System \\
\hline
ADDB \\
MetaPolicy \\
current : StateID \\
\hline
current \in \mathsf{dom}\; states \\
\hline
\end{array}
$$

An initial *System* is defined as follows—the *data* sequence is empty, and the *current* state is set to the *initialstate* of the *MetaPolicy*.

Definition B.37. *(Initial System)*

$$
\begin{array}{|l}
\hline
SystemInit \\
\hline
System' \\
m? : MetaPolicy \\
\hline
\theta\, MetaPolicy' = m? \\
data' = \langle\rangle \\
current' = m?.initialstate \\
\hline
\end{array}
$$

The following operation on *System* updates *ADDB* with the latest access event.

Definition B.38. *(Add Event)*

AddEvent

$$\Delta System$$
$$\Xi MetaPolicy$$
$$event? : Event$$
$$timestamp? : TimeStamp$$

$$data' = Add((event?, timestamp?), data)$$
$$current' = current$$

EvolvePolicy selects the "priority" pair with the lowest sequence index (and thus highest priority) which satisfies the *evaluateMPC* function.

Definition B.39. *(Evolve Policy)*

EvolvePolicy

$$\Delta System$$
$$\Xi MetaPolicy$$
$$\Xi ADDB$$

$$\begin{aligned}
&(\exists_1 c : MPConditionID; i : \mathrm{dom}\ transitions(current) \bullet \\
&\quad evaluateMPC(c, data) = True \wedge \\
&\quad\quad c = first(transitions(current)\ i) \wedge \\
&\quad\quad\quad (\neg\ \exists j : 1\,..\,i-1 \bullet \\
&\quad\quad\quad\quad evaluateMPC((first(transitions(current)\ j)), data)) \Rightarrow \\
&\quad\quad\quad\quad\quad current' = second(transitions(current)\ i)) \\
&\wedge \\
&(\neg\ (\exists_1 c : MPConditionID; i : \mathrm{dom}\ transitions(current) \bullet \\
&\quad evaluateMPC(c, data) = True \wedge \\
&\quad\quad c = first(transitions(current)\ i) \wedge \\
&\quad\quad\quad (\neg\ \exists j : 1\,..\,i-1 \bullet \\
&\quad\quad\quad\quad evaluateMPC((first(transitions(current)\ j)), data)) \Rightarrow \\
&\quad\quad\quad\quad\quad current' = current))
\end{aligned}$$

An observed event can then be defined as the composition of *AddEvent* and *EvolvePolicy*.

Definition B.40. *(Observation)*

$$Observation \mathrel{\widehat=} AddEvent \fatsemi EvolvePolicy$$

B.5 MetaPolicy Properties

Metapolicy properties can be captured in Z using the following schema with appropriate constraints.

Deterministic metapolicies prohibit the same $MPConditionID$ transitioning to next states.

Definition B.41. *(Deterministic MetaPolicy)*

$$
\begin{array}{|l}
\hline
\textit{DeterministicMetaPolicy} \\
\textit{MetaPolicy} \\
\hline
\forall s : \mathrm{ran}\ \textit{transitions} \bullet \\
\quad \forall i, j : \mathrm{dom}\ s \bullet \\
\quad\quad i \neq j \Rightarrow \textit{first}(s\ i) \neq \textit{first}(s\ j) \\
\hline
\end{array}
$$

Connected metapolicies ensure that all states are reachable from the initial state.

Definition B.42. *(Connected MetaPolicy)*

$$
\begin{array}{|l}
\hline
\textit{ConnectedMetaPolicy} \\
\textit{MetaPolicy} \\
\hline
((\mathrm{dom}\ \textit{states}) \setminus \{\textit{initialstate}\}) \subseteq \bigcup \{ s : \mathrm{ran}\ \textit{transitions} \bullet \{ t : \mathrm{ran}\ s \bullet \mathrm{ran}\ t \} \} \\
\hline
\end{array}
$$

This function returns all the resources that a policy allows access to.

Definition B.43. *(Accessible Resources in a Policy)*

$$
\begin{array}{|l}
\textit{accessibleResources} : (\textit{RuleID} \pfun \textit{EACRule}) \times \textit{EACPolicy} \longrightarrow \mathbb{P}\ \textit{Resource} \\
\hline
\forall r : (\textit{RuleID} \pfun \textit{EACRule}); p : \textit{EACPolicy} \bullet \\
\quad \textit{accessibleResources}\ (r, p) = \{ i : \mathrm{ran}\ (p.\textit{rules}) \bullet (r\ i).\textit{resource} \} \\
\end{array}
$$

Restricted metapolicies have evolution traces that subsequently restrict the number of accessible resources.

Definition B.44. *(Restricted MetaPolicy)*

$$
\begin{array}{|l}
\hline
\textit{RestrictedMetaPolicy} \\
\textit{MetaPolicy} \\
\hline
\forall t : \mathrm{dom}\ \textit{transitions} \bullet \\
\quad \forall p : \mathrm{ran}\ (\textit{transitions}(t)) \bullet \\
\quad\quad \textit{accessibleResources}\ (\textit{rule}, \textit{policy}\ (\textit{states}\ (\textit{second}(p)))) \subseteq \\
\quad\quad\quad \textit{accessibleResources}\ (\textit{rule}, \textit{policy}\ (\textit{states}\ (t))) \\
\hline
\end{array}
$$

B.6 Policy Properties

The high-level requirements captured by metapolicies are *global invariants*—they must hold for each state of the evolution. In light of this, the possibility of modelling all the possible states of $ADDB$ allow us to capture these high-level requirements. We argue that such requirements produce access patterns in $ADDB$. In effect, the $states$ structure represents all the possible, valid sequences that can occur for that metapolicy.

Definition B.45. *(Set of ADDB States)*

$$states : \mathbb{P}\, ADDB$$

$$\forall s : states \bullet (\forall a : \mathrm{iseq}\,(Event \times Timestamp) \mid a \; prefix \; s.data \bullet (\exists t : states \bullet t.data = a))$$

Each element in $states$ is an $ADDB$ state. The constraint means that, in the set, there exists an $ADDB\ data$ sequence element which forms a contiguous part of another $ADDB\ data$ sequence element taken from the front (as defined by the $prefix$ library function in [113]).

EXAMPLE A data source contains three resources: a, b and c. The owner would like the following access control requirements to hold for $charlie$.

1. Charlie can access at most two unique resources.
2. Once resource a has been accessed, then there can be no more accesses.

The first requirement can be captured in the following manner—it is the set of all $data$ bindings in which $charlie$ has made, at most, two resource accesses. We formalise this requirement by filtering the $states$ set using our algebra of functions on $ADDB$.

$$charlie : Subject$$
$$a, b, c : Resource$$
$$access : Action$$

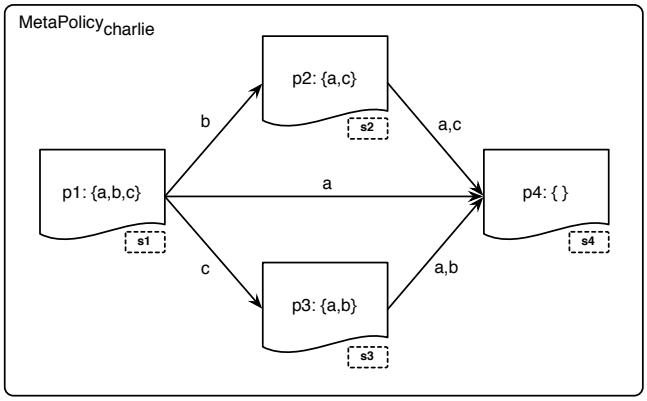
$$\forall s : states \bullet \#SubjectAccessAccepted\,(charlie, s.data) \leq 2$$

A sample of the bindings in the $states$ structure is as follows.

$$\begin{aligned}
&states_1 = \\
&\{\langle\!| data == \langle\rangle |\!\rangle, \\
&\quad \langle\!| data == \langle (internal \langle\!| subject == charlie, resource == a, action == access, result == Accept |\!\rangle, 1) \rangle |\!\rangle, \\
&\quad \langle\!| data == \langle (internal \langle\!| subject == charlie, resource == a, action == access, result == Accept |\!\rangle, 1), \\
&\qquad\qquad (internal \langle\!| subject == charlie, resource == b, action == access, result == Accept |\!\rangle, 2) \rangle |\!\rangle, \ldots\}
\end{aligned}$$

We can abstract away from these bindings and only enumerate the valid resource accesses for this requirement (we notice that each sequence in the set has a size no greater than two).

$$states_1 = \{\langle\rangle, \langle a\rangle, \langle a, b\rangle, \langle a, c\rangle, \langle b\rangle, \langle b, a\rangle, \langle b, c\rangle, \langle c\rangle, \langle c, a\rangle, \langle c, b\rangle\}$$



The second requirement imposes a restriction on the *states* structure that, once resource a has been accessed, then no other accesses can occur. We can formally model this requirement with the following predicate.

$$\begin{aligned}&\forall s : states \bullet \\ &\quad \#(ResourceAccessAccepted(a, s.data)) \leq 1 \land \\ &\qquad (\#(ResourceAccessAccepted(a, s.data)) = 1 \\ &\qquad\quad \Rightarrow (last(s.data)).resource = a)\end{aligned}$$

In every sequence of this set, if a has been accessed, that access must occur last. The corresponding set of valid sequences for this requirement is listed below.

$$states_2 = \{\langle\rangle, \langle a\rangle, \langle b\rangle, \langle b,a\rangle, \langle b,c\rangle, \langle b,c,a\rangle, \langle c\rangle, \langle c,b\rangle, \langle c,b,a\rangle, \langle c,a\rangle\}$$

We see that these high-level requirements serve as global invariants for each state. Data source owners will typically apply several such requirements on a data source. Each requirement can be captured in the form of sequences of valid accesses in the audit data source, i.e. the *states* set is constrained via predicates that reflect these requirements. A combination of requirements is, therefore, the intersection of these sets of valid sequences for each requirement. Therefore, if we were to now combine requirements 1 and 2, the set of possible, valid states is now the *intersection* of the sets produced by each requirement. This set allows us to capture the combined requirements as global invariants that hold in each state of the evolution. The state space for this example is shown in the figure above. The combined set of valid sequences, $states_{\text{valid}}$, can be viewed from the transition arrows in the diagram.

$$\begin{aligned}states_{\text{valid}} &= states_1 \cap states_2 \\ &= \{\langle\rangle, \langle a\rangle, \langle b\rangle, \langle b,a\rangle, \langle b,c\rangle, \langle c\rangle, \langle c,a\rangle, \langle c,b\rangle\}\end{aligned}$$

We have manually demonstrated how metapolicy requirements can produce unique access patterns in our audit data source. These access patterns can be used to verify that policies conform to the requirements captured by metapolicies. In this thesis, such verification is automated using the Alloy Analyzer.

B.7 Transformation

We describe a translation of EAC policies to XACML based on a formal representation of XACML in [79]. The translation to a low-level implementation of our EAC policy allows the deployment of such policies in real systems such as sif [26]. The proof of this translation is outside the scope of this thesis and will be considered in future work.

Why XACML?

Our research ultimately relies on producing machine-readable versions of EAC policies so that it can be used in real systems. While we are aware of the existence of other policy languages such as X-RBAC [97] and SPL [150], we have chosen to utilise XACML for a number of reasons. First, XACML is a standard that has been reviewed by a wide community of experts and users, minimising the need to design a bespoke language. Second, it is powerful in the sense that it offers an expressive language that supports a diverse collection of data types, functions and combining algorithms, that can be easily extended. Third, XACML is sufficiently generic to be deployed in a wide variety of environments, making policy management easier as one policy can be constructed to be used by several applications. Fourth, it supports distributed policies which means that a policy can refer to other policies in disparate locations. This again makes policy management simple as groups of authorised individuals can manage separate portions of the overall policy—with XACML being capable of combining results from different policies into a single decision.

A Formal Representation of XACML

In the construction of a formal translation function, we aim to convert our representation of EAC policies to the formal representation of XACML of [79]. We give a very brief overview of this formal representation of XACML, and identify the areas which will be used in developing our translation. The reader is invited to see [79] for an in-depth description of XACML.

A *Request* models a subject asking to perform an action on a resource within that environment.

Definition B.46. *(Request)*

$$
\begin{array}{|l}
\hline
\mathit{Request} \\
\mathit{request} : \mathit{RequestID} \\
\mathit{sub} : \mathbb{P}_1\, \mathit{Subject} \\
\mathit{act} : \mathbb{P}_1\, \mathit{Action} \\
\mathit{res} : \mathbb{P}_1\, \mathit{Resource} \\
\mathit{env} : \mathbb{P}\, \mathit{Environment} \\
\hline
\# \mathit{act} = 1 \land \# \mathit{env} \leq 1 \\
\hline
\end{array}
$$

The *Target* schema encapsulates the Target element of XACML. This element has been modelled here as a sequence of functions mapping each component of the Target element to an *EvalRes*. These functions map to Boolean values which reflect matches with the request—a value of *True* is returned for *EvalRes* if a match is made, otherwise *False* is returned. For example, the functions in the sequence *act* each capture an XACML Action element. If one of these functions matches the *act* element of the *Request*, then the corresponding rule or policy can be applied (if the other elements also match). Sequences have been used to model these various components as it simplifies the construction of functions that are used to evaluate the matching of requests.

Definition B.47. *(Target)*

$$
\begin{array}{|l}
\textit{Target} \\
\hline
tid : TargetID \\
sub : \mathrm{seq}\,(Subject \longrightarrow EvalRes) \\
act : \mathrm{seq}\,(Action \longrightarrow EvalRes) \\
res : \mathrm{seq}\,(Resource \longrightarrow EvalRes) \\
env : \mathrm{seq}\,(Environment \longrightarrow EvalRes) \\
\hline
\end{array}
$$

We use the following generic function to build up these sequences. This function returns functions which form the component parts of the *Target* schema—they will take the *Request* components as input and return an *EvalRes*: *True* is returned if any of the attributes of the *Request* match; *False* is returned for no match; *Indeterminate* is returned for an error.

Definition B.48. *(Matching Function)*

$$
\begin{array}{|l}
[X] \\
\hline
targetElementEqual : X \longrightarrow X \longrightarrow EvalRes \\
\hline
\forall x : X \bullet targetElementEqual\, x = \{ y : X \setminus \{ x \} \bullet y \mapsto False \} \cup \{ x \mapsto True \} \\
\hline
\end{array}
$$

A rule consists of an identifier, a target for which that rule is applicable, a condition which is evaluated according to the request, and an effect that is returned if the condition evaluates to true. If the condition evaluates to false, or the target is not matched, then the rule will produce an effect of *NotApplicable*. An *Indeterminate* value is returned in the case that an error occurred due to lack of information.

Definition B.49. *(XACML Rule)*

$$
\begin{array}{|l}
\textit{XRule} \\
\hline
rid : RuleID \\
target : TargetID \\
condition : \mathrm{seq}\,(Request \longrightarrow EvalRes) \\
effect : EffectRule \\
\hline
effect \in \{Permit, Deny\} \land \#condition \leq 1 \\
\hline
\end{array}
$$

A policy is composed of an identifier, a target for which the policy is applicable, a sequence of rule identifiers (*RuleID*), and a rule-combining algorithm. There is also a set of elements of type *Obligation* (which can be empty). The XACML standard does not prohibit an empty sequence of rules.

Definition B.50. *(XACML Policy)*

$$
\begin{array}{|l}
\hline
XPolicy \\
\hline
pid : PolicyID \\
target : TargetID \\
inPol : \mathsf{seq}\ RuleID \\
rca : RulComAlgID \\
obli : \mathbb{P}\ Obligation \\
\hline
\end{array}
$$

PolicySet bindings are identified by the *psid* attribute. The *target* represents which policy set is applicable, and *pca* identifies the policy-combining algorithm. The sequence of *PolicyRef* elements can be associated with both *PolicyID* and *PolicySetID* elements, and we note that a policy set may contain an optional set of obligations. The *inPolSet* sequence can be empty.

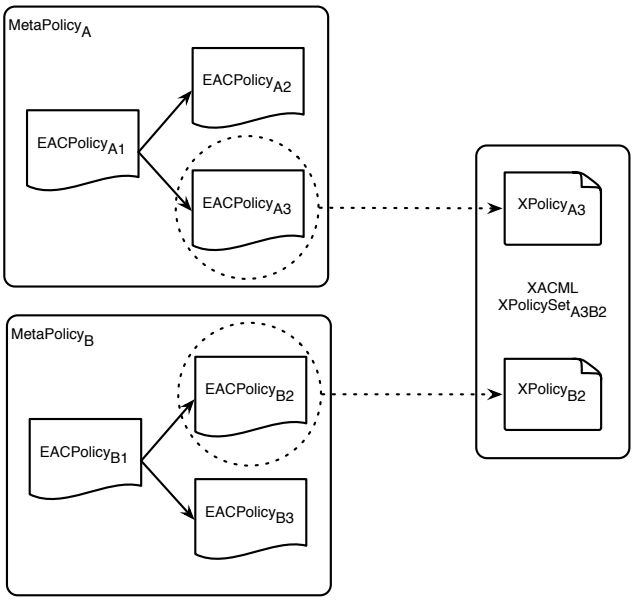
Definition B.51. *(XACML Policy Set)*

$$
\begin{array}{|l}
\hline
XPolicySet \\
\hline
psid : PolicySetID \\
target : TargetID \\
inPolSet : \mathsf{seq}\ PolicyRef \\
pca : PolComAlgID \\
obli : \mathbb{P}\ Obligation \\
\hline
\end{array}
$$

Finally, *XACML* is defined, which is a set of functions that maps identifiers to instances of policy sets, policies, rules and targets. The schema constraints ensure that each function maps an identifier to a schema referencing the same identifier. The definition *rootPol* captures the top level policy set or policy to be evaluated for any particular request.

Definition B.52. *(XACML)*

$$
\begin{array}{|l}
\hline
XACML \\
\hline
getPolicySet : PolicySetID \rightarrowtail\!\!\!\!\!\!\rightarrow XPolicySet \\
getPolicy : PolicyID \rightarrowtail\!\!\!\!\!\!\rightarrow XPolicy \\
getRule : RuleID \rightarrowtail\!\!\!\!\!\!\rightarrow XRule \\
getTarget : TargetID \rightarrowtail\!\!\!\!\!\!\rightarrow Target \\
rootPol : \mathbb{P}\ PolicyRef \\
\hline
(\forall psi : PolicySetID \mid psi \in \mathsf{dom}\ getPolicySet \bullet (getPolicySet\, psi).psid = psi) \\
(\forall pi : PolicyID \mid pi \in \mathsf{dom}\ getPolicy \bullet (getPolicy\, pi).pid = pi) \\
(\forall ri : RuleID \mid ri \in \mathsf{dom}\ getRule \bullet (getRule\, ri).rid = ri) \\
(\forall ti : TargetID \mid ti \in \mathsf{dom}\ getTarget \bullet (getTarget\, ti).tid = ti) \\
\hline
\end{array}
$$



Translation

A high-level view of the translation process is illustrated above. We see that each EAC policy associated with a metapolicy is translated to a formal XACML policy representation, which then makes up a policy set. A policy set can comprise several policies—in our model, one user corresponds to one policy in a sole root policy set. As shown above, each *EACPolicy* element associated with a user (A or B) is converted to a *Policy* element, and then compiled in a *PolicySet* element of [79]. The conversion is handled by our translation operation *EACtoXACMLInit*. Once translated to a *PolicySet* element, another transformation, documented in [79], produces the equivalent machine-readable XACML that can be deployed in access control systems. In effect, this translation allows the deployment of EAC policies in real systems putting our theory into practice. As states evolve, the XACML *PolicySet* binding (containing a policy for each user) is re-generated via *EACtoXACMLInit*.

To begin our formal translation, we define four simple auxiliary functions that can output unique identifiers. The first three functions produce an element of *TargetID*, and generate disjoint sets of identifiers; the final function produces an element of *PolicySetID* given a set of *EACPolicy* elements. We note that these functions are both total and injective so as to ensure that a unique output is created for every valid input.

Definition B.53. *(ID Generator Functions)*

$$
\begin{array}{l}
genRTID : EACRule \rightarrowtail TargetID \\
genPTID : EACPolicy \rightarrowtail TargetID \\
genPSTID : \mathbb{P}\, EACPolicy \rightarrowtail TargetID \\
genPSID : \mathbb{P}\, EACPolicy \rightarrowtail PolicySetID \\
\hline
\mathrm{ran}\ genRTID \cap \mathrm{ran}\ genPTID = \varnothing \\
\mathrm{ran}\ genRTID \cap \mathrm{ran}\ genPSTID = \varnothing \\
\mathrm{ran}\ genPTID \cap \mathrm{ran}\ genPSTID = \varnothing
\end{array}
$$

EACtoXACMLInit is an operation which merges four initialisation schemas. Each of these schemas produces bindings consistent with the formal XACML definitions of [79] (*Target*, *Rule*, *Policy*, *PolicySet*, *XACML*). Our first initialisation schema generates *Target* bindings. In XACML, *targets* are structures that facilitate the matching of access requests. The *targetElementEquals* function of [79] is used to form the *sub*, *act*, *res* and *env* components of this schema, and returns *True* or *False* according to matches with the *Request*.

Definition B.54. *(Target Generator)*

TargetInit

$$Target'$$
$$tid? : TargetID$$
$$sub? : \text{seq } Subject$$
$$act? : \text{seq } Action$$
$$res? : \text{seq } Resource$$
$$env? : \text{seq } Environment$$

$$tid' = tid?$$
$$\forall i : 1 \,..\, \#sub? \bullet sub'\, i = targetElementEqual\,(sub?\, i)$$
$$\forall i : 1 \,..\, \#act? \bullet act'\, i = targetElementEqual\,(act?\, i)$$
$$\forall i : 1 \,..\, \#res? \bullet res'\, i = targetElementEqual\,(res?\, i)$$
$$\forall i : 1 \,..\, \#env? \bullet env'\, i = targetElementEqual\,(env?\, i)$$
$$\#sub' = \#sub? \land \#act' = \#act? \land \#res' = \#res? \land \#env' = \#env?$$

Our next initialisation schema generates a *Rule* binding given an *EACRule* element. The *rid* and *effect* components are taken directly from the *EACRule* element, whereas *target* is generated from a *genRTID* helper function which simply produces a unique *TargetID* type given an *EACRule* element. The *condition* attribute is an empty sequence.

Definition B.55. *(XACML Rule Generator)*

RuleInit

$$Rule'$$
$$er? : EACRule$$

$$rid' = er?.rid$$
$$target' = genRTID\, er?$$
$$effect' = er?.effect$$
$$condition' = \langle\rangle$$

The third initialisation schema creates a *Policy* binding from an *EACPolicy* element. The *target* for this policy is generated using a *genPTID* helper function, whereas the *pid* and *inPol* components are taken directly from the *EACPolicy* element. There are no obligations to be performed, hence *obli* is an empty set. The rule-combining algorithm is set to *rulDenyOverride*—if any rule evaluates to a *Deny*, then the final authorisation decision returned is also *Deny* (closed approach).

Definition B.56. *(XACML Policy Generator)*

$$
\begin{array}{|l}
\textit{PolicyInit} \\
\hline
Policy' \\
ep? : EACPolicy \\
\hline
pid' = ep?.pid \\
target' = genPTID\, ep? \\
inPol' = ep?.rules \\
rca' = rulDenyOverride \\
obli' = \varnothing \\
\hline
\end{array}
$$

It is then necessary to define a policy set initialisation schema that represents the root policy. The following schema generates a *PolicySet* binding given a set of *EACPolicy* elements. The *genPSID* helper function creates a unique *PolicySetID* element for this root policy set. This root policy set holds a sequence of *EACPolicy* elements, each pertaining to one user. An *empty_target_id* means that the target associated with this root policy set allows all requests to filter through to the containing policies as it matches all components.

Definition B.57. *(XACML Policy Set Generator)*

$$
\begin{array}{|l}
\textit{PolicySetInit} \\
\hline
PolicySet' \\
eps? : \mathbb{P}\, EACPolicy \\
\hline
psid' = genPSID\, eps? \\
target' = empty\_target\_id \\
\mathrm{ran}\, inPolSet' = \{ p : eps? \bullet Pol(p.pid) \} \\
pca' = polDenyOverride \\
obli' = \varnothing \\
\hline
\end{array}
$$

A *Context* element unifies a *Subject* element with a *MetaPolicy* element and an *EACPolicy* element. The first two components are needed during the translation, while the third is our *EACPolicy* element that has to be converted to an *XACML* binding. We emphasise that one *Context* element is associated with one user.

Definition B.58. *(Context)*

$$
\begin{array}{|l}
\textit{Context} \\
\hline
user : Subject \\
metapolicy : MetaPolicy \\
policy : EACPolicy \\
\hline
policy \in \mathrm{ran}\,(metapolicy.policy) \\
\forall r : \mathrm{ran}\,(metapolicy.rule) \bullet r.subject = user \\
\hline
\end{array}
$$

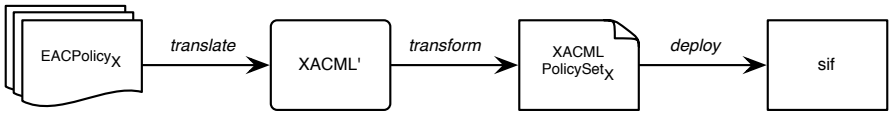
Finally, *EACtoXACMLInit* is defined, which merges these four initialisation schemas to produce our translation operation. *EACtoXACMLInit* is initialised with values from a set of *Context* elements, supporting multiple users. Once initialised, an *XACML* binding is produced that is composed of one policy set with one or more policies (one for each user).

Definition B.59. *(XACML Translator)*

EACtoXACMLInit

$$
\begin{array}{l}
XACML' \\
c? : \mathbb{P}\, Context \\
\hline
\exists\, XPolicySet' \bullet \\
\quad XPolicySetInit[\{ cx : c? \bullet cx.policy \}/eps?] \land psid' \mapsto \theta\, XPolicySet' \in getPolicySet' \\
\\
\forall\, cx : c? \bullet \\
\quad \exists\, XPolicy' \bullet XPolicyInit[cx.policy/ep?] \land pid' \mapsto \theta\, XPolicy' \in getPolicy' \\
\\
\forall\, cx : c? \bullet \\
\quad \forall\, r : cx.policy.rules \bullet \\
\qquad \exists\, XRule' \bullet XRuleInit[(cx.metapolicy.rule\, r)/er?] \land rid' \mapsto \theta\, XRule' \in getRule' \\
\\
\exists\, Target'; tid : TargetID; sub : \mathrm{seq}\, Subject; act : \mathrm{seq}\, Action; \\
\quad res : \mathrm{seq}\, Resource; env : \mathrm{seq}\, Environment \bullet \\
\qquad tid = empty\_target\_id \land sub = \langle\rangle \land act = \langle\rangle \land res = \langle\rangle \land env = \langle\rangle \land \\
\qquad XTargetInit[tid/tid?, sub/sub?, act/act?, res/res?, env/env?] \land \\
\qquad tid' \mapsto \theta\, Target' \in getTarget' \\
\\
\forall\, cx : c? \bullet \\
\quad \exists\, Target'; tid : TargetID; sub : \mathrm{seq}\, Subject; act : \mathrm{seq}\, Action; \\
\qquad res : \mathrm{seq}\, Resource; env : \mathrm{seq}\, Environment \bullet \\
\qquad\quad tid = genPTID\,(cx.policy) \land sub = \langle cx.user \rangle \land \\
\qquad\quad act = \langle\rangle \land res = \langle\rangle \land env = \langle\rangle \land \\
\qquad\quad XTargetInit[tid/tid?, sub/sub?, act/act?, res/res?, env/env?] \land \\
\qquad\quad tid' \mapsto \theta\, Target' \in getTarget' \\
\\
\forall\, cx : c? \bullet \\
\quad \forall\, r : cx.policy.rules \bullet \\
\qquad \exists\, Target'; tid : TargetID; sub : \mathrm{seq}\, Subject; act : \mathrm{seq}\, Action; \\
\qquad\quad res : \mathrm{seq}\, Resource; env : \mathrm{seq}\, Environment \bullet \\
\qquad\qquad tid = genRTID\,(cx.metapolicy.rule\, r) \land \\
\qquad\qquad sub = \langle (cx.metapolicy.rule\, r).subject \rangle \land \\
\qquad\qquad act = \langle (cx.metapolicy.rule\, r).action \rangle \land \\
\qquad\qquad res = \langle (cx.metapolicy.rule\, r).resource \rangle \land env = \langle\rangle \land \\
\qquad\qquad XTargetInit[tid/tid?, sub/sub?, act/act?, res/res?, env/env?] \land \\
\qquad\qquad tid' \mapsto \theta\, Target' \in getTarget'
\end{array}
$$

The predicate section of this definition consists of the following six constraints.



1. The first predicate states that a *PolicySet* binding exists. This binding is created by *PolicySetInit* given a set of *EACPolicy* elements extracted from each *Context* element in the input set. The mapping from a $psid'$ to this binding is a member of the *getPolicySet* function in *XACML*.

2. The second predicate ensures the existence of a *Policy* binding which is produced by *PolicyInit* given an *EACPolicy* element. The mapping from a pid' to this *Policy* binding provides membership for the *getPolicy* function in *XACML*.

3. The third predicate confirms the existence of a *Rule* binding that is produced by *RuleInit* given an *EACRule* element (extracted from the *EACPolicy* element). The mapping from a rid' to this binding is now a member of the *getRule* function of *XACML*.

4. The fourth predicate relates to the *Target* binding produced for the policy set—this creates a binding for an *empty_target_id* and is a member of the *getTarget* function of *XACML*. The other components are left as empty sequences which means this *Target* binding matches all requests.

5. The fifth predicate associates a *Target* binding with a policy—we see that a helper function creates a unique *tid*, and that the subject is set to the *Subject* component of the *Context* element. Other components are left as empty sequences allowing only requests with matching subjects to be evaluated by this *Policy* binding. This produces another mapping for the *getTarget* function.

6. The sixth predicate establishes a *Target* binding for each rule. A helper function produces a unique *tid*, and initialises the *sub*, *act* and *res* components with data from an *EACRule* element. The mapping from this tid' to the *Rule* binding is yet another member of the *getTarget* function of *XACML*.

Having translated EAC policies to a formal representation of XACML, we briefly present the transformation operation of [79], which converts this formal representation of XACML to machine-readable XACML. This end-to-end process, shown in the figure above, validates our work as we can now deploy XACML policies (representing EAC policies) in appropriate access control systems such as sif.

In order to summarise the transformation process of [79], we consider the most demanding component, a *Target* element, as the others (*XPolicySet*, *XPolicy*, *XRule*) are fairly straightforward to transform to XACML. We described the *Target* schema as an abstraction of the *Target* element of the XACML specification. Its components (*Subject*, *Action*, *Resource*, *Environment*) are each modelled as a sequence of functions relating their respective types to *EvalRes* elements. For instance, an element of the sequence *res* is a function of type $Resource \longrightarrow EvalRes$, which represents an XACML *Resource* element.

EXAMPLE Consider an object of type *Target* defined below.

$$
\begin{array}{|l}
rt : Target \\
\hline
rt.tid = rt1 \\
rt.sub = \langle targetElementEquals\,(user)\rangle \\
rt.act = \langle targetElementEquals\,(access)\rangle \\
rt.res = \langle targetElementEquals\,(o1)\rangle \\
rt.env = \langle\rangle
\end{array}
$$

The generic function *targetElementEquals* returns a function for each element in the sequence for that component of the *Target* schema. Therefore, if we were to evaluate *targetElementEquals* $o1$, where o1 is of type *Resource*, then this generates the following function.

$$
\begin{array}{|l}
matchResource : Resource \longrightarrow EvalRes \\
\hline
\forall r : Resource \bullet r = o1 \Rightarrow matchResource\,r = True \\
\forall r : Resource \bullet r \neq o1 \Rightarrow matchResource\,r = False
\end{array}
$$

A function such as this will attempt to match the *Resource* component of the incoming *Request* with value $o1$ and return the relevant result. This function is now equivalent to the following XACML *Resource* fragment. This fragment forms part of the *Target* section in an XACML policy.

```
1 <Resources>
2   <Resource>
3     <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
4       <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">o1</AttributeValue>
5       <ResourceAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string"
6         AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
7     </ResourceMatch>
8   </Resource>
9 </Resources>
```

The functions associated with each component of *Target* are similarly transformed.

This now enables us to return to the sample binding of the initial policy of the images example in the previous section. We only look at a *Target* binding here for the sake of brevity. The *Target* binding for the first *XRule* element is as follows.

$$
\begin{aligned}
genRTID\,(rule1) \mapsto \{\!|\, & tid == genRTID\,(rule1), sub == \langle targetElementEquals\ user\rangle, env == \langle\rangle \\
& act == \langle targetElementEquals\ view\rangle, res == \langle targetElementEquals\ i1\rangle \,|\!\} \\
& \in getTarget'
\end{aligned}
$$

The described transformation process produces the following machine-readable XACML.

```
<Target>
  <Subjects>
    <Subject>
      <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">user</AttributeValue>
        <SubjectAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string"
          AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
      </SubjectMatch>
    </Subject>
  </Subjects>
  <Resources>
    <Resource>
      <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">i1</AttributeValue>
        <ResourceAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string"
          AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
      </ResourceMatch>
    </Resource>
  </Resources>
  <Actions>
    <Action>
      <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">view</AttributeValue>
        <ActionAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string"
          AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
      </ActionMatch>
    </Action>
  </Actions>
</Target>
```

This formal transformation from our version of EAC policies to machine-readable XACML policies acts as a platform for future research which will develop the proof and automation of this process.

B.8 The Images Example

This example concerns an online facility which allows users to pay for viewing images hosted by the site. There is an associated pricing model where some images are worth more than others:

Image	Price
i1	2
i2	2
i3	2
i4	4
i5	4
i6	4
i7	6

The owner of this online facility imposes the following high-level requirements on viewers of these images.

1. One can view any image at most once.
2. One can not have more than three accesses.
3. One can not view a set of images worth more than a value of 10.
4. One can view either only odd or only even images.

Policy writers or authorised personnel would create a metapolicy that captures these initial requirements. The metapolicy then manages the generation of policies according to which conditional events are triggered. The aim is that these policy writers or authorised personnel should not have to constantly modify policies to preserve requirements as evolution proceeds.

Capturing Requirements as Global Invariants

High-level requirements are global invariants in each state of the evolution. We attempt to formally capture these requirements in terms of valid sequences of accesses in the audit data source. Assuming one subject for now, $user$ (since we make the simplifying assumption that metapolicies are per user), we define the set of images and an action.

$$user : Subject$$
$$i1, i2, i3, i4, i5, i6, i7 : Resource$$
$$view : Action$$

The first requirement declares that in every state, all images have been viewed at most once. We can use functions from our algebra of operations to capture this requirement as follows.

$$states_1 = \{ s : states \mid (\forall r : Resource \bullet \\ \#(ResourceAccessAccepted(r, SubjectAccessAccepted(user, s.data))) \leq 1) \}$$

So, for example, in $states_1$, some valid sequences that can be logged include the following. We note that these sequences are a simplification of the *states* structure, as we only focus on resource accesses. We also notice that a resource is logged only once in each *ADDB* state.

$$\langle\rangle, \langle i1\rangle, \langle i1, i2\rangle, \langle i1, i2, i3\rangle, \langle i1, i2, i3, i4\rangle, \langle i1, i2, i3, i4, i5\rangle, \langle i1, i2, i3, i4, i5, i6\rangle$$

Similarly, for the second requirement, it is the case that in every state of *ADDB*, the total number of accesses can not exceed 3. Again, our algebra of operations allows us to retrieve this information, and consequently capture the second requirement using the *states* structure.

$$states_2 = \{ s : states \mid \#SubjectAccessAccepted(user, s.data) \leq 3 \}$$

In $states_2$, we find the following example sequences. Each *ADDB* state sequence has a size no greater than three.

$$\langle\rangle, \langle i1\rangle, \langle i1, i7\rangle, \langle i1, i6, i7\rangle, \langle i5\rangle, \langle i5, i5\rangle, \langle i5, i5, i3\rangle$$

With respect to the third requirement, several domain-specific definitions must be initially defined. The following function maps an image to an integer value representing its value.

$$Values = \{i1 \mapsto 2, i2 \mapsto 2, i3 \mapsto 2, i4 \mapsto 4, i5 \mapsto 4, i6 \mapsto 4, i7 \mapsto 6\}$$

The next "helper" function will recursively sum a sequence of natural numbers (these are the values of each resource in the sequence of internal events).

$$sum : \text{seq } Event \rightarrow\!\!\!\!\!\!+ \; \mathbb{N}$$

$$sum\,\langle\rangle = 0$$
$$sum\,(\langle x\rangle \frown s) = Values\,((internal^{\sim}\, x).resource) + sum\,(s)$$

The third requirement says that in every state, the cumulative sum of the values of the images already accessed cannot exceed a value of 10, and this can be formalised as the following.

$$states_3 = \{ s : states \mid sum(eventsequence(SubjectAccessAccepted(user, s.data))) \leq 10 \}$$

The following sequences can be found in $states_3$.

$$\langle\rangle, \langle i1\rangle, \langle i1, i2\rangle, \langle i1, i2, i3\rangle, \langle i1, i2, i3, i4\rangle, \langle i2\rangle, \langle i2, i2\rangle$$

Finally, we also define additional domain-specific functions for the fourth requirement

that determine if images viewed in every state have been all odd or all even.

$$odd\_ : \mathbb{P}\,\mathbb{N}$$

$$\forall n : \mathbb{N} \bullet odd\, n \Leftrightarrow n \bmod 2 = 1$$

$$even\_ : \mathbb{P}\,\mathbb{N}$$

$$\forall n : \mathbb{N} \bullet even\, n \Leftrightarrow n \bmod 2 = 0$$

These functions are now used to formalise the fourth requirement. We see that, for every *ADDB* state, the sequence of accesses are either all odd or all even images.

$$\begin{aligned} states_4 = \{\, s : states \mid (\forall x : eventsequence\,(SubjectAccessAccepted\,(user, s.data)) \bullet \\ even\,((internal^{\sim}\, x).resource))\,\} \\ \cup \\ \{\, s : states \mid (\forall x : eventsequence\,(SubjectAccessAccepted\,(user, s.data)) \bullet \\ odd\,((internal^{\sim}\, x).resource))\,\} \end{aligned}$$

For example, in $states_4$ the following valid sequences can be found.

$$\langle\rangle, \langle i3\rangle, \langle i3, i5\rangle, \langle i3, i5, i7\rangle, \langle i3, i5, i7, i1\rangle,$$

We combine these four requirements to create the merged set of all valid sequences for this metapolicy. Note that we enumerate these sets here for illustrative purposes; in the general case, these sets would be characterised by predicates.

$$states_{\mathsf{valid}} = states_1 \cap states_2 \cap states_3 \cap states_4$$

$$\begin{aligned} states_{\mathsf{valid}} = \{ & \langle i1\rangle, \langle i1, i3\rangle, \langle i1, i3, i5\rangle, \langle i1, i3, i7\rangle, \langle i1, i5\rangle, \langle i1, i5, i3\rangle, \langle i1, i7\rangle, \langle i1, i7, i3\rangle, \\ & \langle i2\rangle, \langle i2, i4\rangle, \langle i2, i4, i6\rangle, \langle i2, i6\rangle, \langle i2, i6, i4\rangle, \\ & \langle i3\rangle, \langle i3, i1\rangle, \langle i3, i1, i5\rangle, \langle i3, i1, i7\rangle, \langle i3, i5\rangle, \langle i3, i5, i1\rangle, \langle i3, i7\rangle, \langle i3, i7, i1\rangle, \\ & \langle i4\rangle, \langle i4, i2\rangle, \langle i4, i2, i6\rangle, \langle i4, i6\rangle, \langle i4, i6, i2\rangle, \\ & \langle i5\rangle, \langle i5, i1\rangle, \langle i5, i1, i3\rangle, \langle i5, i3\rangle, \langle i5, i3, i1\rangle, \langle i5, i7\rangle, \\ & \langle i6\rangle, \langle i6, i2\rangle, \langle i6, i2, i4\rangle, \langle i6, i4\rangle, \langle i6, i4, i2\rangle, \\ & \langle i7\rangle, \langle i7, i1\rangle, \langle i7, i1, i3\rangle, \langle i7, i3\rangle, \langle i7, i3, i1\rangle, \langle i7, i5\rangle, \\ & \langle\rangle\} \end{aligned}$$

We can now formally instantiate our abstraction for this example.

Instantiating the Abstraction

The abstraction is instantiated to capture the images example in the hopes of producing evolutions that can be used to validate against $states_{\text{valid}}$.

We first define all the identifiers that are used in this example.

$$
\begin{array}{l}
mpid1 : MetaPolicyID \\
mc1, mc2, mc3, mc4, mc5, mc6, mc7 : MPConditionID \\
r1, r2, r3, r4, r5, r6, r7 : RuleID \\
p1, p2, p3, p4, p5, p6, p7, p8, p9, p10, p11, p12, p13, p14, p15 : PolicyID \\
s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, s11, s12, s13, s14, s15 : StateID
\end{array}
$$

The initial policy allows access to all images with seven rules, each allowing access to a different resource. We show just the first rule as an example.

$$
\begin{array}{l}
rule1 : EACRule \\
\hline
rule1.rid = r1 \\
rule1.subject = user \\
rule1.resource = i1 \\
rule1.effect = Permit \\
rule1.action = view
\end{array}
$$

This policy will contain all seven rules in its sequence.

$$
\begin{array}{l}
policy1 : EACPolicy \\
\hline
policy1.pid = p1 \\
policy1.rules = \langle r1, r2, r3, r4, r5, r6, r7 \rangle
\end{array}
$$

We now define conditional events for each transition that occurs in the state diagram. In this example, an evolution occurs whenever a resource is accessed, producing singleton "trigger" sequences. We refer to just resources in our sequences of events again. The "trigger" sequence associated with accessing resource $i1$ actually refers to the following binding.

$$\{\langle internal(\!|subject == user, resource == i1, action == view, result == Accept|\!)\rangle\}$$

Therefore, we produce seven sets of "trigger" sequences using our *condition* function.

$$
\begin{array}{l}
condition(mc1) = \{\langle i1 \rangle\} \\
condition(mc2) = \{\langle i2 \rangle\} \\
condition(mc3) = \{\langle i3 \rangle\} \\
condition(mc4) = \{\langle i4 \rangle\} \\
condition(mc5) = \{\langle i5 \rangle\} \\
condition(mc6) = \{\langle i6 \rangle\} \\
condition(mc7) = \{\langle i7 \rangle\}
\end{array}
$$

The metapolicy for this system can now be modelled with the following bindings.

$$
\begin{array}{l}
imagesmp : MetaPolicy \\
\hline
imagesmp.mid = mpid1 \\
imagesmp.initialstate = s1 \\
imagesmp.rule = \{r1 \mapsto rule1, r2 \mapsto rule2, r3 \mapsto rule3, r4 \mapsto rule4, r5 \mapsto rule5, r6 \mapsto rule6, \\
\qquad r7 \mapsto rule7\} \\
imagesmp.policy = \{p1 \mapsto policy1, p2 \mapsto policy2, p3 \mapsto policy3, p4 \mapsto policy4, p5 \mapsto policy5, \\
\qquad p6 \mapsto policy6, p7 \mapsto policy7, p8 \mapsto policy8, p9 \mapsto policy9, \\
\qquad p10 \mapsto policy10, p11 \mapsto policy11, p12 \mapsto policy12, p13 \mapsto policy13, \\
\qquad p14 \mapsto policy14, p15 \mapsto policy15\} \\
imagesmp.states = \{s1 \mapsto p1, s2 \mapsto p2, s3 \mapsto p3, s4 \mapsto p4, s5 \mapsto p5, s6 \mapsto p6, s7 \mapsto p7, \\
\qquad s8 \mapsto p8, s9 \mapsto p9, s10 \mapsto p10, s11 \mapsto p11, s12 \mapsto p12, s13 \mapsto p13, \\
\qquad s14 \mapsto p14, s15 \mapsto p15\} \\
imagesmp.transitions = \{s1 \mapsto \langle (mc1, s2), (mc2, s3), (mc3, s4), (mc4, s5), (mc5, s6), \\
\qquad\qquad (mc6, s7), (mc7, s8)\rangle, \\
\qquad s2 \mapsto \langle (mc3, s9), (mc5, s10), (mc7, s10)\rangle, \\
\qquad s3 \mapsto \langle (mc4, s11), (mc6, s12)\rangle, \\
\qquad s4 \mapsto \langle (mc1, s9), (mc5, s13), (mc7, s13)\rangle, \\
\qquad s5 \mapsto \langle (mc2, s11), (mc6, s14)\rangle, \\
\qquad s6 \mapsto \langle (mc1, s10), (mc3, s13), (mc7, s15)\rangle, \\
\qquad s7 \mapsto \langle (mc2, s12), (mc4, s14)\rangle, \\
\qquad s8 \mapsto \langle (mc1, s10), (mc3, s13), (mc5, s15)\rangle, \\
\qquad s9 \mapsto \langle (mc5, s15), (mc7, s15)\rangle, \\
\qquad s10 \mapsto \langle (mc3, s15)\rangle, s11 \mapsto \langle (mc6, s15)\rangle, \\
\qquad s12 \mapsto \langle (mc4, s15)\rangle, s13 \mapsto \langle (mc1, s15)\rangle, \\
\qquad s14 \mapsto \langle (mc2, s15)\rangle\}
\end{array}
$$

Finally, we can define a *system* for this example.

$$
\begin{array}{l}
system : System
\end{array}
$$

ADDB is initially empty, while the rest of fields are assigned values from *imagesmp* above.

$$
\begin{array}{l}
system.data = \langle\rangle \\
system.mid = imagesmp.mid \\
system.initialstate = imagesmp.initialstate \\
system.rule = imagesmp.rule \\
system.policy = imagesmp.policy \\
system.states = imagesmp.states \\
system.transitions = imagesmp.transitions \\
system.current = imagesmp.initialstate
\end{array}
$$

Having initialised our *system*, we can advance to a description of policy evolutions.

Evolving Policies

We consider an incremental demonstration of how policies adapt to reflect changes in the system environment. We follow one trace from the set of all possible evolution traces to illustrate how evolution works.

Our *user* first views image $i6$ as the initial policy allows access to all resources at time 1—consequently, the following *Event* is logged.

$$system.data = \langle (internal \langle\!| subject == user, resource == i6, action == view, result == Accept |\!\rangle, 1) \rangle$$

The *system* recognises that there is a new entry in *ADDB*, and executes the *evaluateMPC* function. We see that accessing image $i6$ triggers an evolution because the current sequence of accesses in *ADDB* is a "trigger" sequence (as mapped by $mc6$).

$$evaluateMPC(mc6, system.data) = True$$

The current state evolves and a new policy is created—we observe that rules are not dynamically modified.

$$system.current = s7$$

$$\begin{array}{|l} policy7 : EACPolicy \\ \hline policy7.pid = p7 \\ policy7.rules = \langle r2, r4 \rangle \end{array}$$

Additionally, we note that an absence of rules for the remaining resources implies that they cannot be accessed (in accordance with our closed approach). At this point, the policy has been evolved to only allow access to images $i2$ and $i4$. If *user* chooses to view image $i2$ at time 2, then another event is logged.

$$\begin{aligned} system.data = \langle & (internal \langle\!| subject == user, resource == i6, action == view, result == Accept |\!\rangle, 1), \\ & (internal \langle\!| subject == user, resource == i2, action == view, result == Accept |\!\rangle, 2) \rangle \end{aligned}$$

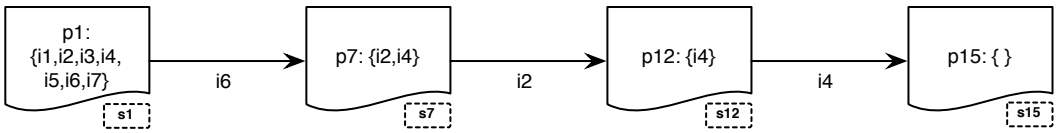
Another evolution takes places because having accessed image $i2$, we see that the current state of *ADDB* provides another "trigger" sequence.

$$evaluateMPC(mc2, system.data) = True$$

The current state, including its policy, are updated again.

$$system.current = s12$$

The new policy is $policy12$.



$$
\begin{array}{|l}
policy12 : EACPolicy \\
\hline
policy12.pid = p12 \\
policy12.rules = \langle r4 \rangle
\end{array}
$$

Finally, *user* has only one image, $i4$, that he or she can access. Another event is logged when this image is viewed at time 3.

$$
\begin{aligned}
system.data = \langle & (internal \langle\!| subject == user, resource == i6, action == view, result == Accept |\!\rangle, 1), \\
& (internal \langle\!| subject == user, resource == i2, action == view, result == Accept |\!\rangle, 2), \\
& (internal \langle\!| subject == user, resource == i4, action == view, result == Accept |\!\rangle, 3) \rangle
\end{aligned}
$$

The current state of *ADDB* provides another "trigger" sequence (from accessing $i4$).

$$evaluateMPC(mc4, system.data) = True$$

This final evolution produces a policy which bars further access to all images.

$$system.current = s15$$

$$
\begin{array}{|l}
policy15 : EACPolicy \\
\hline
policy15.pid = p15 \\
policy15.rules = \langle\rangle
\end{array}
$$

Our *user* has now exhausted one trace of the entire state space as shown in the figure above, and can not progress any further until his or her policy has been reset. The new policy created ($policy15$) contains no rules, which implies that access to all images is denied. We see here that the evolution of policies based on four high-level requirements has allowed *user* to access the images $i6$, $i2$ and $i4$ in that order. In turn, the sequence of accesses logged in *ADDB* is: $\langle i6, i2, i4 \rangle$. When we compare this to all the possible sequences of *ADDB* entries ($states_{\text{valid}}$) calculated from the formalisation of these four requirements, we see that $\langle i6, i2, i4 \rangle$ is indeed a valid sequence of observable transactions in the system. From this manual verification, we can see that this model example (an instantiation of our conceptual framework) has evolved policies based on past transactions, according to high-level requirements embodied in a metapolicy.

In this thesis, we show how the Alloy language and tool is used to automate such verification, and check that each state of the system during evolution is consistent with the requirements of the metapolicy.

Translation to XACML

Each time an evolution occurs, a new EAC policy is generated. Once our metapolicy has been constructed and analysed, we aim to validate our work by translating EAC policies to machine-readable XACML. For demonstration purposes, we formally translate the initial policy of the images example to an *XACML* binding. Later on, this translation will be proved and automated.

First, a context is defined which associates a *Subject* element with a *MetaPolicy* element and an *EACPolicy* element to be translated.

$$
\begin{array}{|l}
con : Context \\
\hline
con.user = user \\
con.metapolicy = imagesmp \\
con.policy = policy1
\end{array}
$$

The *EACtoXACMLInit* initialisation schema is then evaluated by setting the input variable, $c?$, to $\{con\}$ (there is one user and hence one *Context* element in this example). If the identity of a bound variable is known within the quantified expression, then all instances of that variable can be replaced, and the existential quantifier can be removed—this notion of equality forms the basis of the existential one-point rule [131].

An application of this rule to $c?$ and an expansion of *EACtoXACMLInit* allows us to consider each of the six predicates in detail as they build up an *XACML* binding. The first predicate, for instance, can be expanded in the following six stages.

1. As there is only one element in the input set after setting $c?$ to $\{con\}$, the following initial expansion is obtained.

$$
\begin{array}{l}
\exists\, XPolicySet' \bullet \\
\quad XPolicySetInit[\{con.policy\}/eps?] \land psid' \mapsto \theta XPolicySet' \in getPolicySet'
\end{array}
$$

2. $XPolicySet'$ is then expanded using its definition.

$$
\begin{array}{l}
\exists\, psid' : PolicySetID; target' : TargetID; pca' : PolComAlgID; \\
\quad inPolSet' : \mathrm{seq}\ PolicyRef; obli' : \mathbb{P}\ Obligation \bullet \\
\quad XPolicySetInit[\{con.policy\}/eps?] \land psid' \mapsto \theta XPolicySet' \in getPolicySet'
\end{array}
$$

3. Similarly, *XPolicySetInit* is replaced using its definition.

$$
\begin{array}{l}
\exists\, psid' : PolicySetID; target' : TargetID; pca' : PolComAlgID; \\
\quad inPolSet' : \mathrm{seq}\ PolicyRef; obli' : \mathbb{P}\ Obligation \bullet \\
\quad psid' = genPSID(\{con.policy\}) \land \\
\quad target' = empty\_target\_id \land \\
\quad \mathrm{ran}\ inPolSet' = \{ Pol(con.policy.pid) \} \land \\
\quad pca' = polDenyOverride \land obli' = \varnothing \land psid' \mapsto \theta XPolicySet' \in getPolicySet'
\end{array}
$$

4. Next, $\theta XPolicySet'$ is expanded to its entire characteristic binding.

$$\begin{aligned}
&\exists\, psid' : PolicySetID;\ target' : TargetID;\ pca' : PolComAlgID;\\
&\quad inPolSet' : \mathrm{seq}\ PolicyRef;\ obli' : \mathbb{P}\ Obligation \bullet\\
&\quad psid' = genPSID(\{con.policy\}) \land\\
&\quad target' = empty_target_id \land\\
&\quad \mathrm{ran}\ inPolSet' = \{ Pol(con.policy.pid) \} \land\\
&\quad pca' = polDenyOverride \land obli' = \varnothing \land\\
&\quad psid' \mapsto (\!| psid == psid', target == target', inPolSet == inPolSet',\\
&\qquad\qquad pca == pca', obli == obli' |\!) \in getPolicySet'
\end{aligned}$$

5. Having fully expanded the first predicate, more applications of the one-point rule produce the following.

$$\begin{aligned}
&\exists\, inPolSet' : \mathrm{seq}\ PolicyRef \bullet\\
&\quad \mathrm{ran}\ inPolSet' = \{ Pol(con.policy.pid) \} \land\\
&\quad genPSID(\{policy1\}) \mapsto (\!| psid == genPSID(\{policy1\}), target == empty_target_id,\\
&\qquad\qquad inPolSet == inPolSet', pca == polDenyOverride,\\
&\qquad\qquad obli == \varnothing |\!) \in getPolicySet'
\end{aligned}$$

6. Finally, a value exists ($con.policy.pid = p1$) for the only element of the *inPolSet* sequence, which gives us one possibility for this component—applying the one-point rule to eliminate this last existential quantifier now allows us to reduce the first predicate to the following binding.

$$\begin{aligned}
&genPSID(\{policy1\}) \mapsto (\!| psid == genPSID(\{policy1\}), target == empty_target_id,\\
&\qquad\qquad inPolSet == \langle Pol(p1) \rangle, pca == polDenyOverride, obli == \varnothing |\!)\\
&\qquad\qquad \in getPolicySet'
\end{aligned}$$

We observe that this *XPolicySet* binding contains one *XPolicy* reference. In our model, there is always one *XPolicySet* element with one or more *XPolicy* elements each corresponding to a *Subject* element so that the choice of a suitable policy-combining algorithm is irrelevant but still necessary to include in the definition. We also note that our helper functions (such as *genPSID*) are not evaluated since it is the application of a total injective function which produces a unique value given an input.

The remaining five predicates of *EACtoXACMLInit* are now expanded in a similar fashion.

For example, the second predicate generates the following *XPolicy* binding. In the entry below, the *inPol* sequence is associated with the seven rule identifiers of our initial policy. The rule-combining algorithm utilised returns a *Deny* if none of these rules are applicable (technically, as we only have *Permit* rules, a *NotApplicable* is returned but the access mechanism can be configured to deny access, known as a *Deny-biased* system). This is consistent with our sentiment that EAC is a closed approach.

$$\forall cx : \{con\} \bullet$$
$$\quad \exists XPolicy' \bullet XPolicyInit[(cx.policy)/ep?] \land pid' \mapsto \theta XPolicy' \in getPolicy'$$

$$p1 \mapsto \langle\!| pid == p1, target == genPTID(policy1),$$
$$\quad inPol == \langle r1, r2, r3, r4, r5, r6, r7\rangle, rca == rulDenyOverride, obli == \varnothing |\!\rangle$$
$$\quad \in getPolicy'$$

The third predicate produces the following seven *XRule* bindings. The *genRTID* helper function generates a unique *TargetID* binding for each of these *XRule* elements.

$$\forall cx : \{con\} \bullet$$
$$\quad \forall r : cx.policy.rules \bullet$$
$$\quad\quad \exists XRule' \bullet XRuleInit[(cx.metapolicy.rule\, r)/er?] \land rid' \mapsto \theta XRule' \in getRule'$$

$$r1 \mapsto \langle\!| rid == r1, target == genRTID(rule1), effect == Permit, condition == \langle\rangle |\!\rangle \in getRule'$$
$$r2 \mapsto \langle\!| rid == r2, target == genRTID(rule2), effect == Permit, condition == \langle\rangle |\!\rangle \in getRule'$$
$$r3 \mapsto \langle\!| rid == r3, target == genRTID(rule3), effect == Permit, condition == \langle\rangle |\!\rangle \in getRule'$$
$$r4 \mapsto \langle\!| rid == r4, target == genRTID(rule4), effect == Permit, condition == \langle\rangle |\!\rangle \in getRule'$$
$$r5 \mapsto \langle\!| rid == r5, target == genRTID(rule5), effect == Permit, condition == \langle\rangle |\!\rangle \in getRule'$$
$$r6 \mapsto \langle\!| rid == r6, target == genRTID(rule6), effect == Permit, condition == \langle\rangle |\!\rangle \in getRule'$$
$$r7 \mapsto \langle\!| rid == r7, target == genRTID(rule7), effect == Permit, condition == \langle\rangle |\!\rangle \in getRule'$$

Next, we consider the creation of *Target* bindings for policy sets, policies and rules.

The fourth predicate generates a *Target* binding for the only *XPolicySet* element—this *Target* binding allows any request to filter through to the only *XPolicy* binding (a *Target* type filters incoming requests based on *Subject*, *Action*, *Resource* and *Environment* types).

$$\exists Target'; tid : TargetID; sub : \text{seq } Subject; act : \text{seq } Action;$$
$$\quad res : \text{seq } Resource; env : \text{seq } Environment \bullet$$
$$\quad\quad tid = empty\_target\_id \land sub = \langle\rangle \land act = \langle\rangle \land res = \langle\rangle \land env = \langle\rangle \land$$
$$\quad\quad XTargetInit[tid/tid?, sub/sub?, act/act?, res/res?, env/env?] \land$$
$$\quad\quad tid' \mapsto \theta Target' \in getTarget'$$

$$empty\_target\_id \mapsto \langle\!| tid == empty\_target\_id, sub == \langle\rangle, act == \langle\rangle, res == \langle\rangle, env == \langle\rangle |\!\rangle$$
$$\quad\quad \in getTarget'$$

The fifth predicate creates the *Target* binding for the sole *XPolicy* element. The *Subject* element here is set to the *Subject* component associated with the *EACPolicy* element as defined in the input *Context* element—this *Target* binding only allows requests pertaining to that *Subject* value as indicated by empty sequences for the *act*, *res* and *env* components. Additionally, we notice that the *TargetID* binding is generated using a helper function, *genPTID*.

$$
\begin{array}{l}
\forall cx : \{con\} \bullet \\
\quad \exists Target'; tid : TargetID; sub : \mathsf{seq}\ Subject; act : \mathsf{seq}\ Action; \\
\qquad res : \mathsf{seq}\ Resource; env : \mathsf{seq}\ Environment \bullet \\
\qquad\quad tid = genPTID(cx.policy) \land sub = \langle cx.user \rangle \land act = \langle\rangle \land res = \langle\rangle \land env = \langle\rangle \land \\
\qquad\quad XTargetInit[tid/tid?, sub/sub?, act/act?, res/res?, env/env?] \land \\
\qquad\quad tid' \mapsto \theta Target' \in getTarget'
\end{array}
$$

$$
\begin{array}{l}
genPTID(policy1) \mapsto \{| tid == genPTID(policy1), sub == \langle targetElementEquals\ user \rangle, \\
\qquad\qquad act == \langle\rangle, res == \langle\rangle, env == \langle\rangle |\} \\
\qquad\qquad \in getTarget'
\end{array}
$$

The sixth and final predicate of *EACtoXACMLInit* produces the *Target* bindings for the seven *XRule* elements. Each *Target* binding links a *sub*, *act* and *res* with the *subject*, *action* and *resource* component of an *EACPolicy* element respectively. The *MetaPolicy* element for that *Subject* element is used to look up the *EACRule* element mapped to that *RuleID* value. This now produces the following *Target* bindings.

$$
\begin{array}{l}
\forall cx : \{con\} \bullet \\
\quad \forall r : cx.policy.rules \bullet \\
\qquad \exists Target'; tid : TargetID; sub : \mathsf{seq}\ Subject; act : \mathsf{seq}\ Action; \\
\qquad\quad res : \mathsf{seq}\ Resource; env : \mathsf{seq}\ Environment \bullet \\
\qquad\qquad tid = genRTID(cx.metapolicy.rule\,r) \land \\
\qquad\qquad sub = \langle (cx.metapolicy.rule\,r).subject \rangle \land \\
\qquad\qquad act = \langle (cx.metapolicy.rule\,r).action \rangle \land \\
\qquad\qquad res = \langle (cx.metapolicy.rule\,r).resource \rangle \land env = \langle\rangle \land \\
\qquad\qquad XTargetInit[tid/tid?, sub/sub?, act/act?, res/res?, env/env?] \land \\
\qquad\qquad tid' \mapsto \theta Target' \in getTarget'
\end{array}
$$

$$
\begin{array}{l}
genRTID(rule1) \mapsto \{| tid == genRTID(rule1), sub == \langle targetElementEquals\ user \rangle, env == \langle\rangle \\
\qquad\qquad act == \langle targetElementEquals\ view \rangle, res == \langle targetElementEquals\ i1 \rangle |\} \\
\qquad\qquad \in getTarget' \\
genRTID(rule2) \mapsto \{| tid == genRTID(rule2), sub == \langle targetElementEquals\ user \rangle, env == \langle\rangle \\
\qquad\qquad act == \langle targetElementEquals\ view \rangle, res == \langle targetElementEquals\ i2 \rangle |\} \\
\qquad\qquad \in getTarget' \\
genRTID(rule3) \mapsto \{| tid == genRTID(rule3), sub == \langle targetElementEquals\ user \rangle, env == \langle\rangle \\
\qquad\qquad act == \langle targetElementEquals\ view \rangle, res == \langle targetElementEquals\ i3 \rangle |\} \\
\qquad\qquad \in getTarget' \\
genRTID(rule4) \mapsto \{| tid == genRTID(rule4), sub == \langle targetElementEquals\ user \rangle, env == \langle\rangle \\
\qquad\qquad act == \langle targetElementEquals\ view \rangle, res == \langle targetElementEquals\ i4 \rangle |\} \\
\qquad\qquad \in getTarget' \\
genRTID(rule5) \mapsto \{| tid == genRTID(rule5), sub == \langle targetElementEquals\ user \rangle, env == \langle\rangle \\
\qquad\qquad act == \langle targetElementEquals\ view \rangle, res == \langle targetElementEquals\ i5 \rangle |\} \\
\qquad\qquad \in getTarget' \\
genRTID(rule6) \mapsto \{| tid == genRTID(rule6), sub == \langle targetElementEquals\ user \rangle, env == \langle\rangle \\
\qquad\qquad act == \langle targetElementEquals\ view \rangle, res == \langle targetElementEquals\ i6 \rangle |\} \\
\qquad\qquad \in getTarget'
\end{array}
$$

$$genRTID(rule7) \mapsto (\!|tid == genRTID(rule7), sub == \langle targetElementEquals\ user\rangle, env == \langle\rangle$$
$$act == \langle targetElementEquals\ view\rangle, res == \langle targetElementEquals\ i7\rangle|\!)$$
$$\in getTarget'$$

Finally, we collect all the bindings above and associate the name $imagesXACML1$ to the formal XACML representation generated from evaluating $EACtoXACMLInit$ with input $\{con\}$.

$$imagesXACML1 == \{EACtoXACMLInit \mid c? = \{con\} \bullet \theta XACML'\}$$

These mappings manually populate the functions of the $XACML'$ post state ($getPolicySet$, $getPolicy$, $getRule$, $getTarget$). The following now represents the formal XACML representation of the initial policy of the images example.

$$imagesXACML1 ==$$
$$(\!|getPolicySet == \{genPSID(\{policy1\}) \mapsto$$
$$(\!|psid == genPSID(\{policy1\}), target == empty\_target\_id,$$
$$inPolSet == \langle Pol(p1)\rangle, pca == polDenyOverride, obli == \varnothing|\!)\}$$
$$getPolicy == \{p1 \mapsto (\!|pid == p1, target == genPTID(policy1),$$
$$inPol == \langle r1, r2, r3, r4, r5, r6, r7\rangle, rca == rulDenyOverride, obli == \varnothing|\!)\}$$
$$getRule == \{r1 \mapsto (\!|rid == r1, target == genRTID(rule1), effect == Permit, condition == \langle\rangle|\!),$$
$$r2 \mapsto (\!|rid == r2, target == genRTID(rule2), effect == Permit, condition == \langle\rangle|\!),$$
$$r3 \mapsto (\!|rid == r3, target == genRTID(rule3), effect == Permit, condition == \langle\rangle|\!),$$
$$r4 \mapsto (\!|rid == r4, target == genRTID(rule4), effect == Permit, condition == \langle\rangle|\!),$$
$$r5 \mapsto (\!|rid == r5, target == genRTID(rule5), effect == Permit, condition == \langle\rangle|\!),$$
$$r6 \mapsto (\!|rid == r6, target == genRTID(rule6), effect == Permit, condition == \langle\rangle|\!),$$
$$r7 \mapsto (\!|rid == r7, target == genRTID(rule7), effect == Permit, condition == \langle\rangle|\!)\}$$
$$getTarget == \{empty\_target\_id \mapsto (\!|tid == empty\_target\_id,$$
$$sub == \langle\rangle, act == \langle\rangle, res == \langle\rangle, env == \langle\rangle|\!),$$
$$genPTID(policy1) \mapsto (\!|tid == genPTID(policy1), act == \langle\rangle, res == \langle\rangle,$$
$$env == \langle\rangle, sub == \langle targetElementEquals\ user\rangle|\!),$$
$$genRTID(rule1) \mapsto (\!|tid == genRTID(rule1), env == \langle\rangle$$
$$sub == \langle targetElementEquals\ user\rangle,$$
$$act == \langle targetElementEquals\ view\rangle,$$
$$res == \langle targetElementEquals\ i1\rangle|\!),$$
$$genRTID(rule2) \mapsto (\!|tid == genRTID(rule2), env == \langle\rangle$$
$$sub == \langle targetElementEquals\ user\rangle,$$
$$act == \langle targetElementEquals\ view\rangle,$$
$$res == \langle targetElementEquals\ i2\rangle|\!),$$
$$genRTID(rule3) \mapsto (\!|tid == genRTID(rule3), env == \langle\rangle$$
$$sub == \langle targetElementEquals\ user\rangle,$$
$$act == \langle targetElementEquals\ view\rangle,$$
$$res == \langle targetElementEquals\ i3\rangle|\!),$$
$$genRTID(rule4) \mapsto (\!|tid == genRTID(rule4), env == \langle\rangle$$
$$sub == \langle targetElementEquals\ user\rangle,$$
$$act == \langle targetElementEquals\ view\rangle,$$
$$res == \langle targetElementEquals\ i4\rangle|\!),$$

$$
\begin{aligned}
genRTID(rule5) \mapsto \langle\!| & tid == genRTID(rule5), env == \langle\rangle \\
& sub == \langle targetElementEquals\ user \rangle, \\
& act == \langle targetElementEquals\ view \rangle, \\
& res == \langle targetElementEquals\ i5 \rangle |\!\rangle, \\
genRTID(rule6) \mapsto \langle\!| & tid == genRTID(rule6), env == \langle\rangle \\
& sub == \langle targetElementEquals\ user \rangle, \\
& act == \langle targetElementEquals\ view \rangle, \\
& res == \langle targetElementEquals\ i6 \rangle |\!\rangle, \\
genRTID(rule7) \mapsto \langle\!| & tid == genRTID(rule7), env == \langle\rangle \\
& sub == \langle targetElementEquals\ user \rangle, \\
& act == \langle targetElementEquals\ view \rangle, \\
& res == \langle targetElementEquals\ i7 \rangle |\!\rangle, \\
rootPol == \{PolSet(genPSID(\{policy1\}))\} & |\!\rangle
\end{aligned}
$$

The transformation process described in [79] (and briefly in Appendix B.7) demonstrates how the equivalent machine-readable XACML policy can be generated from this formal representation. This XACML policy is listed below, and can be automatically deployed in real access control systems whenever an evolution occurs without manual intervention from a system administrator.

```
<?xml version="1.0" encoding="UTF-8"?>
<PolicySet xmlns="urn:oasis:names:tc:xacml:1.0:policy"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:oasis:names:tc:xacml:1.0:policy cs-xacml-schema-policy-01.xsd"
  PolicyCombiningAlgId="urn:oasis:names:tc:xacml:1.0:policy-combining-algorithm:deny-overrides"
  PolicySetId="imagesxacml1">
  <Target>
    <Subjects>
      <AnySubject/>
    </Subjects>
    <Resources>
      <AnyResource/>
    </Resources>
    <Actions>
      <AnyAction/>
    </Actions>
  </Target>

  <Policy PolicyId="p1" RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:deny-overrides">
    <Target>
      <Subjects>
        <Subject>
          <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">user</AttributeValue>
            <SubjectAttributeDesignator
              DataType="http://www.w3.org/2001/XMLSchema#string"
              AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
          </SubjectMatch>
        </Subject>
      </Subjects>
      <Resources>
        <AnyResource/>
      </Resources>
      <Actions>
        <AnyAction/>
      </Actions>
    </Target>

    <Rule RuleId="r1" Effect="Permit">
      <Target>
        <Subjects>
          <Subject>
            <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">user</AttributeValue>
              <SubjectAttributeDesignator
                DataType="http://www.w3.org/2001/XMLSchema#string"
                AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
```

```
48            </SubjectMatch>
49          </Subject>
50        </Subjects>
51        <Resources>
52          <Resource>
53            <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
54              <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">i1</AttributeValue>
55              <ResourceAttributeDesignator
56                DataType="http://www.w3.org/2001/XMLSchema#string"
57                AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
58            </ResourceMatch>
59          </Resource>
60        </Resources>
61        <Actions>
62          <Action>
63            <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
64            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">view</AttributeValue>
65              <ActionAttributeDesignator
66                DataType="http://www.w3.org/2001/XMLSchema#string"
67                AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
68            </ActionMatch>
69          </Action>
70        </Actions>
71      </Target>
72    </Rule>
73
74    <Rule RuleId="r2" Effect="Permit">
75      <Target>
76        <Subjects>
77          <Subject>
78            <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
79            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">user</AttributeValue>
80              <SubjectAttributeDesignator
81                DataType="http://www.w3.org/2001/XMLSchema#string"
82                AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
83            </SubjectMatch>
84          </Subject>
85        </Subjects>
86        <Resources>
87          <Resource>
88            <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
89              <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">i2</AttributeValue>
90              <ResourceAttributeDesignator
91                DataType="http://www.w3.org/2001/XMLSchema#string"
92                AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
93            </ResourceMatch>
94          </Resource>
95        </Resources>
96        <Actions>
97          <Action>
98            <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
99            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">view</AttributeValue>
100             <ActionAttributeDesignator
101               DataType="http://www.w3.org/2001/XMLSchema#string"
102               AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
103           </ActionMatch>
104         </Action>
105       </Actions>
106     </Target>
107   </Rule>
108
109   <Rule RuleId="r3" Effect="Permit">
110     <Target>
111       <Subjects>
112         <Subject>
113           <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
114           <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">user</AttributeValue>
115             <SubjectAttributeDesignator
116               DataType="http://www.w3.org/2001/XMLSchema#string"
117               AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
118           </SubjectMatch>
119         </Subject>
120       </Subjects>
121       <Resources>
122         <Resource>
123           <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
124             <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">i3</AttributeValue>
125             <ResourceAttributeDesignator
126               DataType="http://www.w3.org/2001/XMLSchema#string"
127               AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
128           </ResourceMatch>
129         </Resource>
```

```
130         </Resources>
131         <Actions>
132           <Action>
133             <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
134             <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">view</AttributeValue>
135               <ActionAttributeDesignator
136                 DataType="http://www.w3.org/2001/XMLSchema#string"
137                 AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
138             </ActionMatch>
139           </Action>
140         </Actions>
141       </Target>
142     </Rule>
143
144     <Rule RuleId="r4" Effect="Permit">
145       <Target>
146         <Subjects>
147           <Subject>
148             <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
149             <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">user</AttributeValue>
150               <SubjectAttributeDesignator
151                 DataType="http://www.w3.org/2001/XMLSchema#string"
152                 AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
153             </SubjectMatch>
154           </Subject>
155         </Subjects>
156         <Resources>
157           <Resource>
158             <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
159               <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">i4</AttributeValue>
160               <ResourceAttributeDesignator
161                 DataType="http://www.w3.org/2001/XMLSchema#string"
162                 AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
163             </ResourceMatch>
164           </Resource>
165         </Resources>
166         <Actions>
167           <Action>
168             <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
169             <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">view</AttributeValue>
170               <ActionAttributeDesignator
171                 DataType="http://www.w3.org/2001/XMLSchema#string"
172                 AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
173             </ActionMatch>
174           </Action>
175         </Actions>
176       </Target>
177     </Rule>
178
179     <Rule RuleId="r5" Effect="Permit">
180       <Target>
181         <Subjects>
182           <Subject>
183             <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
184             <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">user</AttributeValue>
185               <SubjectAttributeDesignator
186                 DataType="http://www.w3.org/2001/XMLSchema#string"
187                 AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
188             </SubjectMatch>
189           </Subject>
190         </Subjects>
191         <Resources>
192           <Resource>
193             <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
194               <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">i5</AttributeValue>
195               <ResourceAttributeDesignator
196                 DataType="http://www.w3.org/2001/XMLSchema#string"
197                 AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
198             </ResourceMatch>
199           </Resource>
200         </Resources>
201         <Actions>
202           <Action>
203             <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
204             <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">view</AttributeValue>
205               <ActionAttributeDesignator
206                 DataType="http://www.w3.org/2001/XMLSchema#string"
207                 AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
208             </ActionMatch>
209           </Action>
210         </Actions>
211       </Target>
```

```
212     </Rule>
213
214     <Rule RuleId="r6" Effect="Permit">
215       <Target>
216         <Subjects>
217           <Subject>
218             <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
219             <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">user</AttributeValue>
220               <SubjectAttributeDesignator
221                 DataType="http://www.w3.org/2001/XMLSchema#string"
222                 AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
223             </SubjectMatch>
224           </Subject>
225         </Subjects>
226         <Resources>
227           <Resource>
228             <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
229               <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">i6</AttributeValue>
230               <ResourceAttributeDesignator
231                 DataType="http://www.w3.org/2001/XMLSchema#string"
232                 AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
233             </ResourceMatch>
234           </Resource>
235         </Resources>
236         <Actions>
237           <Action>
238             <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
239             <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">view</AttributeValue>
240               <ActionAttributeDesignator
241                 DataType="http://www.w3.org/2001/XMLSchema#string"
242                 AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
243             </ActionMatch>
244           </Action>
245         </Actions>
246       </Target>
247     </Rule>
248
249     <Rule RuleId="r7" Effect="Permit">
250       <Target>
251         <Subjects>
252           <Subject>
253             <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
254             <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">user</AttributeValue>
255               <SubjectAttributeDesignator
256                 DataType="http://www.w3.org/2001/XMLSchema#string"
257                 AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
258             </SubjectMatch>
259           </Subject>
260         </Subjects>
261         <Resources>
262           <Resource>
263             <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
264               <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">i7</AttributeValue>
265               <ResourceAttributeDesignator
266                 DataType="http://www.w3.org/2001/XMLSchema#string"
267                 AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
268             </ResourceMatch>
269           </Resource>
270         </Resources>
271         <Actions>
272           <Action>
273             <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
274             <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">view</AttributeValue>
275               <ActionAttributeDesignator
276                 DataType="http://www.w3.org/2001/XMLSchema#string"
277                 AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
278             </ActionMatch>
279           </Action>
280         </Actions>
281       </Target>
282     </Rule>
283   </Policy>
284 </PolicySet>
```

Appendix C: Alloy Model

C.1 Policy and Rule

```
module EAC/policy

open EAC/addb
open util/relation

-- rule and policy identifiers
abstract sig RuleID, PolicyID {}

-- rule attributes
abstract sig Subject, Action {}
abstract sig Effect {}
one sig Permit, Deny extends Effect {}

-- internal event
abstract sig Resource extends Event {}

-- rule
abstract sig EACRule {
    rid: RuleID,
    subject: Subject,
    resource: Resource,
    effect: Effect,
    action: Action
}

-- policy
abstract sig EACPolicy {
    pid: PolicyID,
    rules: seq RuleID
} {
    -- no duplicate rules in the sequence (iseq)
    !rules.hasDups
}
```

C.2 MetaPolicy

```
module EAC/metapolicy

open EAC/policy
open util/relation
open util/ternary

-- metapolicy identifiers
abstract sig MetaPolicyID, StateID, MPConditionID {}

-- priority pairs
abstract sig Priority { mpc: MPConditionID, state: StateID }

-- metapolicy
abstract sig MetaPolicy {
    mid: MetaPolicyID,
    initialstate: StateID,
    rule: RuleID → lone EACRule,
    policy: PolicyID → lone EACPolicy,
    states: StateID → lone PolicyID,
    transitions: StateID → (seq Priority)
} {
    -- initial state is in the domain of states
    initialstate in dom[states]
    -- all states have a mapping
    all s: ran[transitions] | s.state in dom[states]
    -- all rules have a mapping
    all r: dom[rule] | rule[r].rid = r
    -- all policies have a mapping
    all p: dom[policy] | policy[p].pid = p
    -- all states are mapped to a policy
    all i: ran[states] | i in dom[policy]
    -- all rules are mapped to a policy
    all j: ran[policy] | all k: ran[j.rules] | k in dom[rule]
}

-- canonicalize fact ensures two priority atoms with the same
-- mpc and state will be represented by the same priority atom
fact canonicalisation { no disj a,b:Priority | a.mpc=b.mpc && a.state=b.state }
```

C.3 ADDB

```
module EAC/addb

abstract sig Event {}

-- external events are a type of event
abstract sig External extends Event {}

sig ADDB {
    data: seq Event
} {
    -- no duplicate events in the sequence (iseq)
    !data.hasDups
}
```

C.4 Conditional Event

```
module EAC/mpcondition

open EAC/metapolicy
open EAC/addb

-- trigger sequences signal state changes
abstract sig Trigger {
    data: seq Event
}

-- mpconditions are conditional events associated
-- with an id and a set of trigger sequences
abstract sig MPCondition {
    mpid: MPConditionID,
    trigger: set Trigger
} {
    -- each sequence is never empty
    all k: trigger | #k.data != 0
    -- set of triggers is never empty
    trigger != none
}
```

C.5 System

```
module EAC/system

open EAC/addb
open EAC/metapolicy

-- a system models an eac system state
sig System {
    addb: ADDB,
    metapolicy: MetaPolicy,
    current: StateID
} {
    current in dom[metapolicy.states]
}
```

C.6 Framework of Functions

```
module EAC/core

open EAC/metapolicy
open EAC/addb
open EAC/subsequence
open EAC/mpcondition
open EAC/system

-- an observation is a change in the system when an event has occurred
pred observation (s, s': System) {
    let p = getPolicy [s.metapolicy, s.current] |
        some e: (accessibleResources[s.metapolicy, p] + External) |
            addEventEvolve[e, s, s']
}

-- returns the Policy mapped from a StateID
fun getPolicy (m: MetaPolicy, s: StateID) : Policy {
    pid.(m.states[s])
}

-- returns the set of accessible resources in that Policy
fun accessibleResources (m: MetaPolicy, p: Policy) : set Resource {
    (m.rule[p.rules.elems]).resource
}

-- composite function of adding an event and evolving
pred addEventEvolve (e: Event, s, s': System) {
    s'.addb.data = s.addb.data.add[e]
    s'.metapolicy = s.metapolicy
    one c: MPConditionID, i: dom[s.metapolicy.transitions[s.current]] |
      (evaluateMPC [c, s'.addb] and
          c = first[s.metapolicy.transitions[s.current][i]] and
              (no j: seq/Int | 1 <= j and j <= (i - 1) and
                  evaluateMPC [first[s.metapolicy.transitions[s.current][j]], s'.addb]) and
                      s'.current = second[s.metapolicy.transitions[s.current][i]])
}

-- determines if a trigger sequence is in ADDB
pred evaluateMPC (c: MPConditionID, a: ADDB) {
    some t: condition[c] | subsequence[t.data, a.data]
}

-- given an mpid, returns the associated set of triggers
fun condition (c: MPConditionID) : set Trigger {
    mpid.c.trigger
}
```

```
-- get the first element of the tuple
fun first (p: Priority) : MPConditionID {
    p.mpc
}

-- get the second element of the tuple
fun second (p: Priority) : StateID {
    p.state
}
```

C.7 Subsequence

```
module EAC/subsequence

open EAC/addb

-- recursive function unrolled to delegate control based on addb length
pred subsequence (t: seq Resource, a: seq Resource) {
        #a = 4 implies subsequence4 [t, a]
    else
        #a = 3 implies subsequence3 [t, a]
    else
        #a = 2 implies subsequence2 [t, a]
    else
        #a = 1 implies subsequence1 [t, a]
    else
        #a = 0 implies subsequence0 [t, a]
}

pred subsequence4 (t: seq Resource, a: seq Resource) {
    #t != 0 implies
        (t.last = a.last implies subsequence3 [t.butlast, a.butlast]
        else
        subsequence3 [t, a.butlast])
}

pred subsequence3 (t: seq Resource, a: seq Resource) {
    #t != 0 implies
        (t.last = a.last implies subsequence2 [t.butlast, a.butlast]
        else
        subsequence2 [t, a.butlast])
}

pred subsequence2 (t: seq Resource, a: seq Resource) {
    #t != 0 implies
        (t.last = a.last implies subsequence1 [t.butlast, a.butlast]
        else
        subsequence1 [t, a.butlast])
}

pred subsequence1 (t: seq Resource, a: seq Resource) {
    #t != 0 implies
        (t.last = a.last implies subsequence0 [t.butlast, a.butlast]
        else
        subsequence0 [t, a.butlast])
}

pred subsequence0 (t: seq Resource, a: seq Resource) {
    #t = 0
}
```

C.8 Images Example

```
module EAC/images

open EAC/metapolicy
open EAC/addb
open EAC/mpcondition
open EAC/system

open util/ordering [System]

-- basic types
one sig user extends Subject {}
one sig i1, i2, i3, i4, i5, i6, i7 extends Resource {}
one sig view extends Action {}
one sig empty extends External {}
one sig mpid1 extends MetaPolicyID {}
one sig mc1, mc2, mc3, mc4, mc5, mc6, mc7 extends MPConditionID {}
one sig p1, p2, p3, p4, p5, p6, p7 extends PolicyID {}
one sig p8, p9, p10, p11, p12, p13, p14, p15 extends PolicyID {}
one sig s1, s2, s3, s4, s5, s6, s7 extends StateID {}
one sig s8, s9, s10, s11, s12, s13, s14, s15 extends StateID {}
one sig r1, r2, r3, r4, r5, r6, r7 extends RuleID {}

-- metapolicy
one sig imagesmp extends MetaPolicy {} {
  mid = mpid1
  initialstate = s1
  rule = {(r1 → rule1) + (r2 → rule2) + (r3 → rule3) + (r4 → rule4) +
          (r5 → rule5) + (r6 → rule6) + (r7 → rule7)}
  policy = {(p1 → policy1) + (p2 → policy2) + (p3 → policy3) + (p4 → policy4) +
            (p5 → policy5) + (p6 → policy6) + (p7 → policy7) + (p8 → policy8) +
            (p9 → policy9) + (p10 → policy10) + (p11 → policy11) +
            (p12 → policy12) + (p13 → policy13) + (p14 → policy14) +
            (p15 → policy15)}
  states = {(s1 → p1) + (s2 → p2) + (s3 → p3) + (s4 → p4) + (s5 → p5) +
            (s6 → p6) + (s7 → p7) + (s8 → p8) + (s9 → p9) + (s10 → p10) +
            (s11 → p11) + (s12 → p12) + (s13 → p13) + (s14 → p14) +
            (s15 → p15)}
  transitions = {(s1 → ((0 → ps1) + (1 → ps2) + (2 → ps3) + (3 → ps4) +
                        (4 → ps5) + (5 → ps6) + (6 → ps7))) +
                 (s2 → ((0 → ps8) + (1 → ps9) + (2 → ps10))) +
                 (s3 → ((0 → ps11) + (1 → ps12))) +
                 (s4 → ((0 → ps13) + (1 → ps14) + (2 → ps15))) +
                 (s5 → ((0 → ps16) + (1 → ps17))) +
                 (s6 → ((0 → ps18) + (1 → ps19) + (2 → ps20))) +
                 (s7 → ((0 → ps21) + (1 → ps22))) +
                 (s8 → ((0 → ps18) + (1 → ps19) + (2 → ps23))) +
                 (s9 → ((0 → ps23) + (1 → ps20))) +
                 (s10 → (0 → ps24)) +
```

```
                (s11 → (0 → ps25)) +
                (s12 → (0 → ps26)) +
                (s13 → (0 → ps27)) +
                (s14 → (0 → ps28))}
}

-- priority pairs
one sig ps1 extends Priority {} {mpc = mc1 and state = s2}
one sig ps2 extends Priority {} {mpc = mc2 and state = s3}
one sig ps3 extends Priority {} {mpc = mc3 and state = s4}
one sig ps4 extends Priority {} {mpc = mc4 and state = s5}
one sig ps5 extends Priority {} {mpc = mc5 and state = s6}
one sig ps6 extends Priority {} {mpc = mc6 and state = s7}
one sig ps7 extends Priority {} {mpc = mc7 and state = s8}
one sig ps8 extends Priority {} {mpc = mc3 and state = s9}
one sig ps9 extends Priority {} {mpc = mc5 and state = s10}
one sig ps10 extends Priority {} {mpc = mc7 and state = s10}
one sig ps11 extends Priority {} {mpc = mc4 and state = s11}
one sig ps12 extends Priority {} {mpc = mc6 and state = s12}
one sig ps13 extends Priority {} {mpc = mc1 and state = s9}
one sig ps14 extends Priority {} {mpc = mc5 and state = s13}
one sig ps15 extends Priority {} {mpc = mc7 and state = s13}
one sig ps16 extends Priority {} {mpc = mc2 and state = s11}
one sig ps17 extends Priority {} {mpc = mc6 and state = s14}
one sig ps18 extends Priority {} {mpc = mc1 and state = s10}
one sig ps19 extends Priority {} {mpc = mc3 and state = s13}
one sig ps20 extends Priority {} {mpc = mc7 and state = s15}
one sig ps21 extends Priority {} {mpc = mc2 and state = s12}
one sig ps22 extends Priority {} {mpc = mc4 and state = s14}
one sig ps23 extends Priority {} {mpc = mc5 and state = s15}
one sig ps24 extends Priority {} {mpc = mc3 and state = s15}
one sig ps25 extends Priority {} {mpc = mc6 and state = s15}
one sig ps26 extends Priority {} {mpc = mc4 and state = s15}
one sig ps27 extends Priority {} {mpc = mc1 and state = s15}
one sig ps28 extends Priority {} {mpc = mc2 and state = s15}

-- rules
one sig rule1 extends EACRule {} {
    rid = r1 and subject = user and resource = i1 and effect = Permit and action = view
}

one sig rule2 extends EACRule {} {
    rid = r2 and subject = user and resource = i2 and effect = Permit and action = view
}

one sig rule3 extends EACRule {} {
    rid = r3 and subject = user and resource = i3 and effect = Permit and action = view
}
```

```
one sig rule4 extends EACRule {} {
    rid = r4 and subject = user and resource = i4 and effect = Permit and action = view
}

one sig rule5 extends EACRule {} {
    rid = r5 and subject = user and resource = i5 and effect = Permit and action = view
}

one sig rule6 extends EACRule {} {
    rid = r6 and subject = user and resource = i6 and effect = Permit and action = view
}

one sig rule7 extends EACRule {} {
    rid = r7 and subject = user and resource = i7 and effect = Permit and action = view
}

-- policies
one sig policy1 extends EACPolicy {} {
    pid = p1 and rules = {(0 → r1) + (1 → r2) + (2 → r3) + (3 → r4) +
                          (4 → r5) + (5 → r6) + (6 → r7)}
}

one sig policy2 extends EACPolicy {} {
    pid = p2 and rules = {(0 → r3) + (1 → r5) + (2 → r7)}
}

one sig policy3 extends EACPolicy {} {
    pid = p3 and rules = {(0 → r4) + (1 → r6)}
}

one sig policy4 extends EACPolicy {} {
    pid = p4 and rules = {(0 → r1) + (1 → r5) + (2 → r7)}
}

one sig policy5 extends EACPolicy {} {
    pid = p5 and rules = {(0 → r2) + (1 → r6)}
}

one sig policy6 extends EACPolicy {} {
    pid = p6 and rules = {(0 → r1) + (1 → r3) + (2 → r7)}
}

one sig policy7 extends EACPolicy {} {
    pid = p7 and rules = {(0 → r2) + (1 → r4)}
}

one sig policy8 extends EACPolicy {} {
    pid = p8 and rules = {(0 → r1) + (1 → r3) + (2 → r5)}
}
```

```
one sig policy9 extends EACPolicy {} {
    pid = p9 and rules = {(0 → r5) + (1 → r7)}
}

one sig policy10 extends EACPolicy {} {
    pid = p10 and rules = {(0 → r3)}
}

one sig policy11 extends EACPolicy {} {
    pid = p11 and rules = {(0 → r6)}
}

one sig policy12 extends EACPolicy {} {
    pid = p12 and rules = {(0 → r4)}
}

one sig policy13 extends EACPolicy {} {
    pid = p13 and rules = {(0 → r1)}
}

one sig policy14 extends EACPolicy {} {
    pid = p14 and rules = {(0 → r2)}
}

one sig policy15 extends EACPolicy {} {
    pid = p15 and no rules
}

-- trigger sequences
one sig trig1 extends Trigger {} {
    data = {(0 → i1)}
}

one sig trig2 extends Trigger {} {
    data = {(0 → i2)}
}

one sig trig3 extends Trigger {} {
    data = {(0 → i3)}
}

one sig trig4 extends Trigger {} {
    data = {(0 → i4)}
}

one sig trig5 extends Trigger {} {
    data = {(0 → i5)}
}
```

```
one sig trig6 extends Trigger {} {
    data = {(0 → i6)}
}

one sig trig7 extends Trigger {} {
    data = {(0 → i7)}
}

-- mpconditions
one sig mp1 extends MPCondition {} {
    mpid = mc1 and trigger = {trig1}
}

one sig mp2 extends MPCondition {} {
    mpid = mc2 and trigger = {trig2}
}

one sig mp3 extends MPCondition {} {
    mpid = mc3 and trigger = {trig3}
}

one sig mp4 extends MPCondition {} {
    mpid = mc4 and trigger = {trig4}
}

one sig mp5 extends MPCondition {} {
    mpid = mc5 and trigger = {trig5}
}

one sig mp6 extends MPCondition {} {
    mpid = mc6 and trigger = {trig6}
}

one sig mp7 extends MPCondition {} {
    mpid = mc7 and trigger = {trig7}
}

-- domain specific signatures and functions
abstract sig Class {}

one sig Odd, Even extends Class {}

one sig Domain {
    values: Resource → Int,
    groups: Resource → Class
} {
    values = {i1 → 2 + i2 → 2 + i3 → 2 + i4 → 4 + i5 → 4 + i6 → 4 + i7 → 6}
    groups = {i1 → Odd + i2 → Even + i3 → Odd + i4 → Even + i5 → Odd +
                i6 → Even + i7 → Odd}
}
```

```
-- addb initial
one sig imagesaddb extends ADDB {} {
    no data
}

-- system initial
pred init (s: System) {
    s.addb = imagesaddb
    s.metapolicy = imagesmp
    s.current = imagesmp.initialstate
}
```

C.9 Execute

```
module EAC/execute

-- call framework of functions
open EAC/core

-- load example metapolicy here
open EAC/images

-- transitions is a trace of the policy evolution
fact transitions {
    init[first[]]
    all s: System - last[] | let s' = next[s] | observation [s, s']
}

-- find an instance
pred evolve {}

run evolve for exactly 4 System, 4 ADDB, 4 int, 7 seq

-- analyse properties

-- determinism
assert determinism {
all s: System | {
    (all i: dom[s.metapolicy.transitions] |
        all disj j, k: dom[s.metapolicy.transitions[i]] |
            first[s.metapolicy.transitions[i][j]] != first[s.metapolicy.transitions[i][k]])
            }
}

check determinism for exactly 4 System, 4 ADDB, 4 int, 7 seq

assert uniquepriority {
    all s: System | {
        all disj i,j: ran[s.metapolicy.transitions[s.current]] | i != j
        }
}

check uniquepriority for exactly 4 System, 4 ADDB, 4 int, 7 seq

-- connectedness
assert connectedness {
    all s: System | {
        (dom[s.metapolicy.states] - s.metapolicy.initialstate) in ran[s.metapolicy.transitions].state
        }
}

check connectedness for exactly 4 System, 4 ADDB, 4 int, 7 seq
```

```
-- restriction
assert restriction {
  all s: System - last[] | let s' = next[s] | {
      observation [s, s'] implies
          (accessibleResources [s'.metapolicy, (getPolicy [s'.metapolicy, s'.current])] in
              accessibleResources [s.metapolicy, (getPolicy [s.metapolicy, s.current])])
              }
}

check restriction for exactly 4 System, 4 ADDB, 4 int, 7 seq

-- verify consistency

-- binary
assert binary {
    all s: System | {
        all r: accessibleResources[s.metapolicy, (getPolicy[s.metapolicy, s.current])] |
            r !in s.addb.data.elems
            }
}

check binary for exactly 4 System, 4 ADDB, 4 int, 7 seq

-- counting
assert counting {
    all s: System | {
        #s.addb.data = 3 implies
            no accessibleResources[s.metapolicy, (getPolicy[s.metapolicy, s.current])]
            }
}

check counting for exactly 4 System, 4 ADDB, 4 int, 7 seq

-- subscription
assert subscription {
    all s: System | {
        all r: accessibleResources[s.metapolicy, (getPolicy[s.metapolicy, s.current])] |
            ((sum n: s.addb.data.elems | Domain.values[n])) + Domain.values[r] =< 10
            }
}

check subscription for exactly 4 System, 4 ADDB, 5 int, 7 seq
```

```
-- compartment
assert compartment {
  all s: System | {
      all r: accessibleResources[s.metapolicy, (getPolicy[s.metapolicy, s.current])] |
          ((all n: s.addb.data.elems | Domain.groups[n] = Odd) and Domain.groups[r] = Odd)
          or
          ((all n: s.addb.data.elems | Domain.groups[n] = Even) and Domain.groups[r] = Even)
          }
}

check compartment for exactly 4 System, 4 ADDB, 4 int, 7 seq
```

C.10 Results

Executing "Run evolve **for** 4 **int**, 7 **seq**, **exactly** 4 System, 4 ADDB"
Solver=sat4j Bitwidth=4 MaxSeq=7 SkolemDepth=1 Symmetry=20
108418 vars. 6105 primary vars. 311873 clauses. 306240ms.
Instance found. Predicate is consistent. 260ms.

Executing "Check determinism **for** 4 **int**, 7 **seq**, **exactly** 4 System, 4 ADDB"
Solver=sat4j Bitwidth=4 MaxSeq=7 SkolemDepth=1 Symmetry=20
115639 vars. 6138 primary vars. 336259 clauses. 308100ms.
No counterexample found. Assertion may be valid. 173ms.

Executing "Check uniquepriority **for** 4 **int**, 7 **seq**, **exactly** 4 System, 4 ADDB"
Solver=sat4j Bitwidth=4 MaxSeq=7 SkolemDepth=1 Symmetry=20
114930 vars. 6165 primary vars. 324578 clauses. 310230ms.
No counterexample found. Assertion may be valid. 701ms.

Executing "Check connectedness **for** 4 **int**, 7 **seq**, **exactly** 4 System, 4 ADDB"
Solver=sat4j Bitwidth=4 MaxSeq=7 SkolemDepth=1 Symmetry=20
112321 vars. 6109 primary vars. 319366 clauses. 316892ms.
No counterexample found. Assertion may be valid. 107ms.

Executing "Check restriction **for** 4 **int**, 7 **seq**, **exactly** 4 System, 4 ADDB"
Solver=sat4j Bitwidth=4 MaxSeq=7 SkolemDepth=1 Symmetry=20
111898 vars. 6109 primary vars. 318252 clauses. 313880ms.
No counterexample found. Assertion may be valid. 233ms.

Executing "Check binary **for** 4 **int**, 7 **seq**, **exactly** 4 System, 4 ADDB"
Solver=sat4j Bitwidth=4 MaxSeq=7 SkolemDepth=1 Symmetry=20
110514 vars. 6116 primary vars. 315795 clauses. 312859ms.
No counterexample found. Assertion may be valid. 192ms.

Executing "Check counting **for** 4 **int**, 7 **seq**, **exactly** 4 System, 4 ADDB"
Solver=sat4j Bitwidth=4 MaxSeq=7 SkolemDepth=1 Symmetry=20
110822 vars. 6109 primary vars. 317714 clauses. 315637ms.
No counterexample found. Assertion may be valid. 273ms.

Executing "Check subscription **for** 5 **int**, 7 **seq**, **exactly** 4 System, 4 ADDB"
Solver=sat4j Bitwidth=5 MaxSeq=7 SkolemDepth=1 Symmetry=20
113996 vars. 6228 primary vars. 330625 clauses. 339914ms.
No counterexample found. Assertion may be valid. 284ms.

Executing "Check compartment **for** 4 **int**, 7 **seq**, **exactly** 4 System, 4 ADDB"
Solver=sat4j Bitwidth=4 MaxSeq=7 SkolemDepth=1 Symmetry=20
110529 vars. 6116 primary vars. 315897 clauses. 310655ms.
No counterexample found. Assertion may be valid. 356ms.

9 commands were executed. The results are:

#1: **Instance** found. evolve is consistent.

#2: No counterexample found. determinism may be valid.

#3: No counterexample found. uniquepriority may be valid.

#4: No counterexample found. connectedness may be valid.

#5: No counterexample found. restriction may be valid.

#6: No counterexample found. binary may be valid.

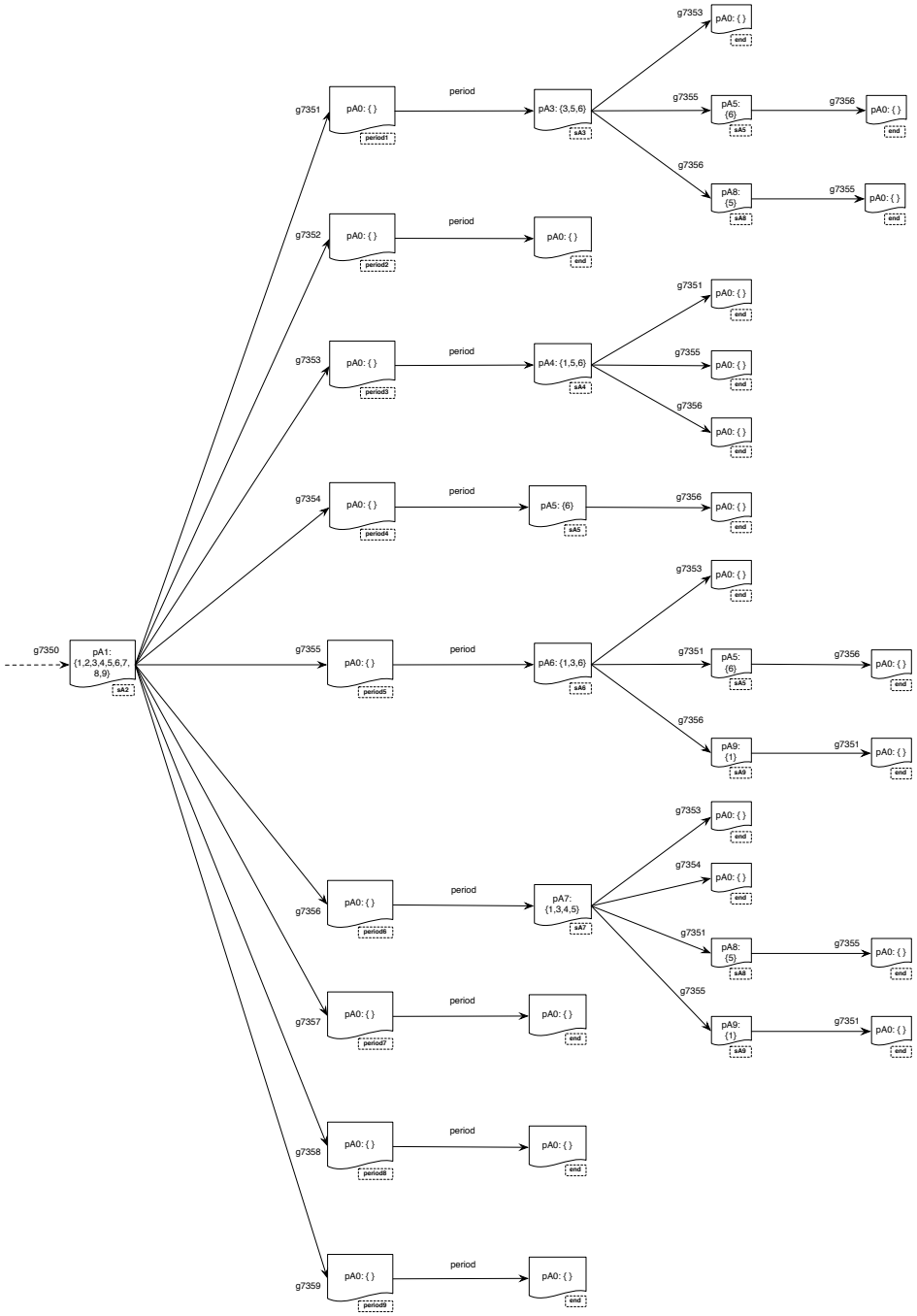
#7: No counterexample found. counting may be valid.

#8: No counterexample found. subscription may be valid.

#9: No counterexample found. compartment may be valid.

Appendix D: Case Study

D.1 State Space: Alice Node 0



D.2 Z Model: Alice Node 0

Using the Z formal language, we instantiate our abstraction to capture our case study for user Alice. We recall that the high-level requirements imposed on Alice are as follows.

- Binary: Alice can access any graduate record but only once.
- Period: Alice can only access two graduate records per hour.
- Subscription: Alice can not access more than 10 rating points worth of graduate data.

There are 10 rows of *Graduate* data in this data source. We also observe the existence of an external event, which is triggered when an hour has elapsed.

$$
\begin{array}{l}
alice : Subject \\
g7350, g7351, g7352, g7353, g7354, g7355, g7356, g7357, g7358, g7359 : Resource \\
period : External
\end{array}
$$

Capturing Requirements as Global Invariants

The first requirement declares that in every state of *ADDB*, any graduate record must have been viewed at most once. We can capture this requirement using our algebra of functions to filter the *states* structure.

$$
\begin{aligned}
states_1 = \{s : states \mid (\forall r : Resource \bullet \\
\#(ResourceAccessAccepted(r, SubjectAccessAccepted(alice, s.data))) \leq 1)\}
\end{aligned}
$$

Similarly, the second requirement says that Alice can only access two graduate records per hour. This suggests that the number of accesses by Alice within the past hour must not exceed two.

$$
\begin{aligned}
states_2 = \{s : states \mid (\forall t : Timestamp \bullet \\
\#(TimePeriodAccessAccepted(t, t+1, SubjectAccessAccepted(alice, s.data))) \leq 2)\}
\end{aligned}
$$

With respect to the third requirement, we define domain-specific definitions, and re-define the following function to map each graduate record to an integer value representing the rating.

$$
\begin{aligned}
Values = \{g7350 \mapsto 5, g7351 \mapsto 2, g7352 \mapsto 5, g7353 \mapsto 3, g7354 \mapsto 4, \\
g7355 \mapsto 2, g7356 \mapsto 1, g7357 \mapsto 5, g7358 \mapsto 5, g7359 \mapsto 5\}
\end{aligned}
$$

Then, we use our *sum* function to get the current value of the ratings accessed by Alice. The third requirement can now be formalised as the following.

$$
states_3 = \{s : states \mid sum(eventsequence(SubjectAccessAccepted(alice, s.data))) \leq 10\}
$$

Instantiating the Abstraction

Our EAC abstraction is now instantiated to capture the details of this case study pertaining to user *Alice*.

A *System* binding is first declared.

$$systemA : System$$

The state of our audit data source, *ADDB*, is initially empty with nothing logged.

$$systemA.data = \langle\rangle$$

Additionally, the identifiers used in this case study for Alice are defined as follows.

$$access : Action$$
$$node0 : MetaPolicyID$$
$$mcA0, mcA1, mcA2, mcA3, mcA4, mcA5, mcA6, mcA7, mcA8, mcA9, mcA10 : MPConditionID$$
$$rA0, rA1, rA2, rA3, rA4, rA5, rA6, rA7, rA8, rA9 : RuleID$$
$$pA0, pA1, pA2, pA3, pA4, pA5, pA6, pA7, pA8, pA9, pA10 : PolicyID$$
$$sA1, sA2, sA3, sA4, sA5, sA6, sA7, sA8, sA9, end : StateID$$
$$period1, period2, period3, period4, period5, period6, period7, period8, period9 : StateID$$

"Trigger" sequences are defined for the set of internal events and the one external event.

$$condition(mcA0) = \{\langle g7350\rangle\}$$
$$condition(mcA1) = \{\langle g7351\rangle, \langle g7350, g7351\rangle, \langle g7350, g7355, period, g7351\rangle\}$$
$$condition(mcA2) = \{\langle g7352\rangle, \langle g7350, g7352\rangle\}$$
$$condition(mcA3) = \{\langle g7353\rangle, \langle g7350, g7353\rangle, \langle g7350, g7351, period, g7353\rangle\}$$
$$condition(mcA4) = \{\langle g7354\rangle, \langle g7350, g7354\rangle, \langle g7350, g7356, period, g7354\rangle\}$$
$$condition(mcA5) = \{\langle g7355\rangle, \langle g7350, g7355\rangle, \langle g7350, g7353, period, g7355\rangle\}$$
$$condition(mcA6) = \{\langle g7356\rangle, \langle g7350, g7356\rangle, \langle g7350, g7353, period, g7356\rangle\}$$
$$condition(mcA7) = \{\langle g7357\rangle, \langle g7350, g7357\rangle\}$$
$$condition(mcA8) = \{\langle g7358\rangle, \langle g7350, g7358\rangle\}$$
$$condition(mcA9) = \{\langle g7359\rangle, \langle g7350, g7359\rangle\}$$
$$condition(mcA10) = \{\langle period\rangle\}$$

The corresponding metapolicy can now be modelled with the following mappings.

$$
\begin{array}{l}
csamp : MetaPolicy \\
\hline
csamp.mid = node0 \\
csamp.initialstate = sA1 \\
csamp.rule = \{rA0 \mapsto ruleA0, rA1 \mapsto ruleA1, rA2 \mapsto ruleA2, rA3 \mapsto ruleA3, rA4 \mapsto ruleA4, \\
\qquad rA5 \mapsto ruleA5, rA6 \mapsto ruleA6, rA7 \mapsto ruleA7, rA8 \mapsto ruleA8, rA9 \mapsto ruleA9\} \\
csamp.policy = \{pA0 \mapsto empty, pA1 \mapsto policyA1, pA2 \mapsto policyA2, pA3 \mapsto policyA3, \\
\qquad pA4 \mapsto policyA4, pA5 \mapsto policyA5, pA6 \mapsto policyA6, pA7 \mapsto policyA7, \\
\qquad pA8 \mapsto policyA8, pA9 \mapsto policyA9\} \\
csamp.states = \{sA1 \mapsto pA1, sA2 \mapsto pA2, sA3 \mapsto pA3, sA4 \mapsto pA4, sA5 \mapsto pA5, sA6 \mapsto pA6, \\
\qquad sA7 \mapsto pA7, sA8 \mapsto pA8, sA9 \mapsto pA9, end \mapsto pA0, period1 \mapsto pA0, \\
\qquad period2 \mapsto pA0, period3 \mapsto pA0, period4 \mapsto pA0, period5 \mapsto pA0, \\
\qquad period6 \mapsto pA0, period7 \mapsto pA0, period8 \mapsto pA0, period9 \mapsto pA0\} \\
csamp.transitions = \{sA1 \mapsto \langle (mcA0, sA2)\rangle, sA2 \mapsto \langle (mcA1, period1), (mcA2, period2), \\
\qquad (mcA3, period3), (mcA4, period4), (mcA5, period5), (mcA6, period6), \\
\qquad (mcA7, period7), (mcA8, period8), (mcA9, period)\rangle, sA3 \mapsto \langle (mcA5, sA5), \\
\qquad (mcA6, sA8), (mcA3, end)\rangle, sA4 \mapsto \langle (mcA1, end), (mcA5, end), (mcA6, end)\rangle, \\
\qquad sA5 \mapsto \langle (mcA6, end)\rangle, sA6 \mapsto \langle (mcA1, sA5), (mcA6, sA9), (mcA3, end)\rangle, \\
\qquad sA7 \mapsto \langle (mcA1, sA8), (mcA5, sA9), (mcA3, end), (mcA4, end)\rangle, \\
\qquad sA8 \mapsto \langle (mcA5, end)\rangle, sA9 \mapsto \langle (mcA1, end)\rangle, period1 \mapsto \langle (mcA10, sA3)\rangle, \\
\qquad period2 \mapsto \langle (mcA10, end)\rangle, period3 \mapsto \langle (mcA10, sA4)\rangle, \\
\qquad period4 \mapsto \langle (mcA10, sA5)\rangle, period5 \mapsto \langle (mcA10, sA6)\rangle, \\
\qquad period6 \mapsto \langle (mcA10, sA7)\rangle, period7 \mapsto \langle (mcA10, end)\rangle, \\
\qquad period8 \mapsto \langle (mcA10, end)\rangle, period9 \mapsto \langle (mcA10, end)\rangle\}
\end{array}
$$

Evolving Policies

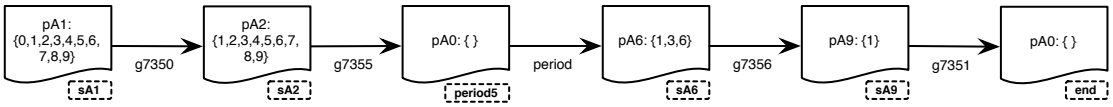
We briefly demonstrate one trace of policy evolutions based on Alice's requests as shown overleaf. Alice can initially access any of 10 graduate records. She first chooses to access graduate record $g7350$. The following event is logged.

$$systemA.data = \langle (internal \langle\!| subject == alice, resource == g7350, action == access, result == Accept |\!\rangle, 1)\rangle$$

A new entry in $ADDB$ executes the $evaluateMPC$ function. We see that accessing resource $g7350$ triggers an evolution because the current sequence of accesses in $ADDB$ is a "trigger" sequence (as mapped by $mcA0$).

$$
\begin{array}{l}
evaluateMPC(mcA0, systemA.data) = True \\
systemA.current = sA2
\end{array}
$$

The current state evolves and the following policy is created.



$$policyA2 : EACPolicy$$

$$policyA2.pid = pA2$$
$$policyA2.rules = \langle rA1, rA2, rA3, rA4, rA5, rA6, rA7, rA8, rA9 \rangle$$

If Alice were to access another resource, $g7355$, within the hour, another evolution takes place. However, access is now denied to all resources as two records have been accessed. When an hour has elapsed, an external event, $period$, is logged in $ADDB$.

$$\begin{aligned} systemA.data = \langle & (internal\langle\!| subject == alice, resource == g7350, action == access, result == Accept |\!\rangle, 1), \\ & (internal\langle\!| subject == alice, resource == g7355, action == access, result == Accept |\!\rangle, 2), \\ & (external(period), 3)\rangle \end{aligned}$$

This external event triggers Alice's current policy to evolve and allow access once more — the state is updated to $sA6$.

The *system* continues to handle Alice's requests until her policy needs to be reset (the cumulative sum of rating points accessed is 10, and so the current policy denies all her requests). We observe her final access log.

$$\begin{aligned} systemA.data = \langle & (internal\langle\!| subject == alice, resource == g7350, action == access, result == Accept |\!\rangle, 1), \\ & (internal\langle\!| subject == alice, resource == g7355, action == access, result == Accept |\!\rangle, 2), \\ & (external(period), 3), \\ & (internal\langle\!| subject == alice, resource == g7356, action == access, result == Accept |\!\rangle, 4), \\ & (internal\langle\!| subject == alice, resource == g7351, action == access, result == Accept |\!\rangle, 5)\rangle \end{aligned}$$

Translation to XACML

In our case study, we consider the context of three users which implies that three XACML policies are embodied in one policy set which can be deployed in an access control system.

Let us assume, that at some point in the evolution of policies, the current context for each user is as follows.

$$conA, conB, conC : Context$$

$$conA.user = alice \wedge conA.metapolicy = csamp \wedge conA.policy = policyA6$$
$$conB.user = bob \wedge conB.metapolicy = csbmp \wedge conB.policy = policyB3$$
$$conC.user = charlie \wedge conC.metapolicy = cscmp \wedge conC.policy = policyC2$$

Let us further assume that the current EAC policy for each user is as follows.

$$
\begin{array}{|l}
policyA6, policyB3, policyC2 : EACPolicy \\
\hline
policyA6.pid = pA6 \land policyA6.rules = \langle rA1, rA3, rA6 \rangle \\
policyB3.pid = pB3 \land policyB3.rules = \langle\rangle \\
policyC2.pid = pC2 \land policyC2.rules = \langle rC0, rC9 \rangle
\end{array}
$$

The corresponding EAC rules are defined below.

$$
\begin{array}{|l}
ruleA1, ruleA3, ruleA6, ruleC0, ruleC9 : EACRule \\
\hline
ruleA1.rid = rA1 \land ruleA1.subject = alice \land ruleA1.resource = g7351 \\
ruleA1.effect = Permit \land ruleA1.action = access \\
ruleA3.rid = rA3 \land ruleA3.subject = alice \land ruleA3.resource = g7353 \\
ruleA3.effect = Permit \land ruleA3.action = access \\
ruleA6.rid = rA6 \land ruleA6.subject = alice \land ruleA6.resource = g7356 \\
ruleA6.effect = Permit \land ruleA6.action = access \\
ruleC0.rid = rC0 \land ruleC0.subject = charlie \land ruleC0.resource = s39465 \\
ruleC0.effect = Permit \land ruleC0.action = access \\
ruleC9.rid = rC9 \land ruleC9.subject = charlie \land ruleC9.resource = s40566 \\
ruleC9.effect = Permit \land ruleC9.action = access
\end{array}
$$

Finally, we assume that $csamp$, $csbmp$, and $cscmp$ are the respective metapolicies.

We now initialise $EACtoXACMLInit$ with the input $c?$ set to $\{conA, conB, conC\}$. The application of the one-point rule to $c?$, and an expansion of $EACtoXACMLInit$ allow us to consider each of its six predicates in the construction of this translation. We remind the reader that our translation process involves generating bindings for one $XPolicySet$, one or more $XPolicy$, one or more $XRule$, and then the corresponding $Target$ bindings for each of these components. These bindings then manually populate an $XACML$ post state, which can be subsequently transformed to XACML and deployed.

The expansion of predicates is discussed in Appendix B.7—in this section, we just list the predicate and corresponding bindings that are created.

The first predicate produces the bindings necessary for an $XPolicySet$ element.

$$
\begin{array}{l}
\exists\, XPolicySet' \bullet \\
\quad XPolicySetInit[\{conA.policy, conB.policy, conC.policy\}/eps?] \land \\
\quad psid' \mapsto \theta XPolicySet' \in getPolicySet'
\end{array}
$$

$$
\begin{array}{l}
genPSID(\{policyA6, policyB3, policyC2\}) \mapsto \langle\!| psid == genPSID(\{policyA6, policyB3, policyC2\}), \\
\qquad\qquad target == empty\_target\_id, \\
\qquad\qquad inPolSet == \langle Pol(pA6), Pol(pB3), Pol(pC2) \rangle, \\
\qquad\qquad pca == polDenyOverride, obli == \varnothing |\!\rangle \\
\qquad\qquad \in getPolicySet'
\end{array}
$$

The second predicate generates the following *XPolicy* bindings. We observe that the universal quantifier expands the input set to three *Context* elements, each containing one *EACPolicy* element per user.

$$\forall cx : \{conA, conB, conC\} \bullet$$
$$\exists XPolicy' \bullet XPolicyInit[(cx.policy)/ep?] \land pid' \mapsto \theta XPolicy' \in getPolicy'$$

$$pA6 \mapsto \langle\!| pid == pA6, target == genPTID(policyA6),$$
$$inPol == \langle rA1, rA3, rA6\rangle, rca == rulDenyOverride, obli == \varnothing |\!\rangle$$
$$\in getPolicy'$$
$$pB3 \mapsto \langle\!| pid == pB3, target == genPTID(policyB3),$$
$$inPol == \langle\rangle, rca == rulDenyOverride, obli == \varnothing |\!\rangle$$
$$\in getPolicy'$$
$$pC2 \mapsto \langle\!| pid == pC2, target == genPTID(policyC2),$$
$$inPol == \langle rC0, rC9\rangle, rca == rulDenyOverride, obli == \varnothing |\!\rangle$$
$$\in getPolicy'$$

The third predicate produces the following *XRule* bindings for each of the three policies.

$$\forall cx : \{conA, conB, conC\} \bullet$$
$$\forall r : cx.policy.rules \bullet$$
$$\exists XRule' \bullet XRuleInit[(cx.metapolicy.rule\, r)/er?] \land rid' \mapsto \theta XRule' \in getRule'$$

$$rA1 \mapsto \langle\!| rid == rA1, target == genRTID(ruleA1), effect == Permit, condition == \langle\rangle |\!\rangle \in getRule'$$
$$rA3 \mapsto \langle\!| rid == rA3, target == genRTID(ruleA3), effect == Permit, condition == \langle\rangle |\!\rangle \in getRule'$$
$$rA6 \mapsto \langle\!| rid == rA6, target == genRTID(ruleA6), effect == Permit, condition == \langle\rangle |\!\rangle \in getRule'$$
$$rC0 \mapsto \langle\!| rid == rC0, target == genRTID(ruleC0), effect == Permit, condition == \langle\rangle |\!\rangle \in getRule'$$
$$rC9 \mapsto \langle\!| rid == rC9, target == genRTID(ruleC9), effect == Permit, condition == \langle\rangle |\!\rangle \in getRule'$$

The fourth predicate details the creation of a *Target* binding for the *XPolicySet* element.

$$\exists Target'; tid : TargetID; sub : \text{seq } Subject; act : \text{seq } Action;$$
$$res : \text{seq } Resource; env : \text{seq } Environment \bullet$$
$$tid = empty\_target\_id \land sub = \langle\rangle \land act = \langle\rangle \land res = \langle\rangle \land env = \langle\rangle \land$$
$$XTargetInit[tid/tid?, sub/sub?, act/act?, res/res?, env/env?] \land$$
$$tid' \mapsto \theta Target' \in getTarget'$$

$$empty\_target\_id \mapsto \langle\!| tid == empty\_target\_id, sub == \langle\rangle, act == \langle\rangle, res == \langle\rangle, env == \langle\rangle |\!\rangle$$
$$\in getTarget'$$

The fifth predicate considers the generation of *Target* bindings for three *XPolicy* elements.

$$
\begin{array}{l}
\forall cx : \{conA, conB, conC\} \bullet \\
\quad \exists Target'; tid : TargetID; sub : \mathbb{seq}\, Subject; act : \mathbb{seq}\, Action; \\
\qquad res : \mathbb{seq}\, Resource; env : \mathbb{seq}\, Environment \bullet \\
\qquad\quad tid = genPTID(cx.policy) \land sub = \langle cx.user \rangle \land \\
\qquad\quad act = \langle\rangle \land res = \langle\rangle \land env = \langle\rangle \land \\
\qquad\quad XTargetInit[tid/tid?, sub/sub?, act/act?, res/res?, env/env?] \land \\
\qquad\quad tid' \mapsto \theta Target' \in getTarget'
\end{array}
$$

$$
\begin{array}{l}
genPTID(policyA6) \mapsto \{| tid == genPTID(policyA6), sub == \langle targetElementEquals\ alice \rangle, \\
\qquad\qquad act == \langle\rangle, res == \langle\rangle, env == \langle\rangle |\} \\
\qquad\qquad \in getTarget' \\
genPTID(policyB3) \mapsto \{| tid == genPTID(policyB3), sub == \langle targetElementEquals\ bob \rangle, \\
\qquad\qquad act == \langle\rangle, res == \langle\rangle, env == \langle\rangle |\} \\
\qquad\qquad \in getTarget' \\
genPTID(policyC2) \mapsto \{| tid == genPTID(policyC2), sub == \langle targetElementEquals\ charlie \rangle, \\
\qquad\qquad act == \langle\rangle, res == \langle\rangle, env == \langle\rangle |\} \\
\qquad\qquad \in getTarget'
\end{array}
$$

The sixth predicate generates the *Target* bindings for each *XRule* element.

$$
\begin{array}{l}
\forall cx : \{conA, conB, conC\} \bullet \\
\quad \forall r : cx.policy.rules \bullet \\
\qquad \exists Target'; tid : TargetID; sub : \mathbb{seq}\, Subject; act : \mathbb{seq}\, Action; \\
\qquad\quad res : \mathbb{seq}\, Resource; env : \mathbb{seq}\, Environment \bullet \\
\qquad\qquad tid = genRTID(cx.metapolicy.rule\, r) \land \\
\qquad\qquad sub = \langle (cx.metapolicy.rule\, r).subject \rangle \land \\
\qquad\qquad act = \langle (cx.metapolicy.rule\, r).action \rangle \land \\
\qquad\qquad res = \langle (cx.metapolicy.rule\, r).resource \rangle \land env = \langle\rangle \land \\
\qquad\qquad XTargetInit[tid/tid?, sub/sub?, act/act?, res/res?, env/env?] \land \\
\qquad\qquad tid' \mapsto \theta Target' \in getTarget'
\end{array}
$$

$$
\begin{array}{l}
genRTID(ruleA1) \mapsto \{| tid == genRTID(ruleA1), sub == \langle targetElementEquals\ alice \rangle, \\
\qquad\qquad env == \langle\rangle, act == \langle targetElementEquals\ access \rangle, \\
\qquad\qquad res == \langle targetElementEquals\ g7351 \rangle |\} \\
\qquad\qquad \in getTarget' \\
genRTID(ruleA3) \mapsto \{| tid == genRTID(ruleA3), sub == \langle targetElementEquals\ alice \rangle, \\
\qquad\qquad env == \langle\rangle, act == \langle targetElementEquals\ access \rangle, \\
\qquad\qquad res == \langle targetElementEquals\ g7353 \rangle |\} \\
\qquad\qquad \in getTarget' \\
genRTID(ruleA6) \mapsto \{| tid == genRTID(ruleA6), sub == \langle targetElementEquals\ alice \rangle, \\
\qquad\qquad env == \langle\rangle, act == \langle targetElementEquals\ access \rangle, \\
\qquad\qquad res == \langle targetElementEquals\ g7356 \rangle |\} \\
\qquad\qquad \in getTarget'
\end{array}
$$

$$
\begin{aligned}
genRTID(ruleC0) \mapsto (\!|\, & tid == genRTID(ruleC0), sub == \langle targetElementEquals\ charlie\rangle, \\
& env == \langle\rangle, act == \langle targetElementEquals\ access\rangle, \\
& res == \langle targetElementEquals\ s39465\rangle \,|\!) \\
& \in getTarget' \\
genRTID(ruleC9) \mapsto (\!|\, & tid == genRTID(ruleC9), sub == \langle targetElementEquals\ charlie\rangle, \\
& env == \langle\rangle, act == \langle targetElementEquals\ access\rangle, \\
& res == \langle targetElementEquals\ s40566\rangle \,|\!) \\
& \in getTarget'
\end{aligned}
$$

Finally, we associate the name $csXACML632$ with the combination of bindings produced from our translation process here. This is the formal XACML representation generated from evaluating $EACtoXACMLInit$ with input $\{conA, conB, conC\}$.

$$csXACML632 == \{EACtoXACMLInit \mid c? = \{conA, conB, conC\} \bullet \theta XACML'\}$$

This gives us the following mappings which manually populate the functions of the $XACML'$ post state ($getPolicySet'$, $getPolicy'$, $getRule'$, $getTarget'$). $csXACML632$ represents the formal XACML policy set containing three policies (one for each user, Alice, Bob, and Charlie) at a particular point in time during system evolution.

$$
\begin{aligned}
& csXACML632 == \\
& (\!|\, getPolicySet == \{genPSID(\{policyA6, policyB3, policyC2\}) \mapsto \\
& \qquad (\!|\, psid == genPSID(\{policyA6, policyB3, policyC2\}), \\
& \qquad\ target == empty\_target\_id, \\
& \qquad\ inPolSet == \langle Pol(pA6), Pol(pB3), Pol(pC2)\rangle, \\
& \qquad\ pca == polDenyOverride, obli == \varnothing \,|\!)\} \\
& \quad getPolicy == \{pA6 \mapsto (\!|\, pid == pA6, target == genPTID(policyA6), inPol == \langle rA1, rA3, rA6\rangle, \\
& \qquad\ rca == rulDenyOverride, obli == \varnothing \,|\!), \\
& \qquad pB3 \mapsto (\!|\, pid == pB3, target == genPTID(policyB3), inPol == \langle\rangle, \\
& \qquad\ rca == rulDenyOverride, obli == \varnothing \,|\!), \\
& \qquad pC2 \mapsto (\!|\, pid == pC2, target == genPTID(policyC2), inPol == \langle rC0, rC9\rangle, \\
& \qquad\ rca == rulDenyOverride, obli == \varnothing \,|\!)\} \\
& \quad getRule == \{rA1 \mapsto (\!|\, rid == rA1, target == genRTID(ruleA1), \\
& \qquad\ effect == Permit, condition == \langle\rangle \,|\!), \\
& \qquad rA3 \mapsto (\!|\, rid == rA3, target == genRTID(ruleA3), \\
& \qquad\ effect == Permit, condition == \langle\rangle \,|\!), \\
& \qquad rA6 \mapsto (\!|\, rid == rA6, target == genRTID(ruleA6), \\
& \qquad\ effect == Permit, condition == \langle\rangle \,|\!), \\
& \qquad rC0 \mapsto (\!|\, rid == rC0, target == genRTID(ruleC0), \\
& \qquad\ effect == Permit, condition == \langle\rangle \,|\!), \\
& \qquad rC9 \mapsto (\!|\, rid == rC9, target == genRTID(ruleC9), \\
& \qquad\ effect == Permit, condition == \langle\rangle \,|\!),
\end{aligned}
$$

$$
\begin{aligned}
getTarget == \{ & empty\_target\_id \mapsto \langle\!| tid == empty\_target\_id, \\
& \qquad sub == \langle\rangle, act == \langle\rangle, res == \langle\rangle, env == \langle\rangle |\!\rangle, \\
& genPTID(policyA6) \mapsto \langle\!| tid == genPTID(policyA6), act == \langle\rangle, res == \langle\rangle, \\
& \qquad env == \langle\rangle, sub == \langle targetElementEquals\ alice \rangle |\!\rangle, \\
& genPTID(policyB3) \mapsto \langle\!| tid == genPTID(policyB3), act == \langle\rangle, res == \langle\rangle, \\
& \qquad env == \langle\rangle, sub == \langle targetElementEquals\ bob \rangle |\!\rangle, \\
& genPTID(policyC2) \mapsto \langle\!| tid == genPTID(policyC2), act == \langle\rangle, res == \langle\rangle, \\
& \qquad env == \langle\rangle, sub == \langle targetElementEquals\ charlie \rangle |\!\rangle, \\
& genRTID(ruleA1) \mapsto \langle\!| tid == genRTID(ruleA1), env == \langle\rangle, \\
& \qquad sub == \langle targetElementEquals\ alice \rangle, \\
& \qquad act == \langle targetElementEquals\ access \rangle, \\
& \qquad res == \langle targetElementEquals\ g7351 \rangle |\!\rangle, \\
& genRTID(ruleA3) \mapsto \langle\!| tid == genRTID(ruleA3), env == \langle\rangle, \\
& \qquad sub == \langle targetElementEquals\ alice \rangle, \\
& \qquad act == \langle targetElementEquals\ access \rangle, \\
& \qquad res == \langle targetElementEquals\ g7353 \rangle |\!\rangle, \\
& genRTID(ruleA6) \mapsto \langle\!| tid == genRTID(ruleA6), env == \langle\rangle, \\
& \qquad sub == \langle targetElementEquals\ alice \rangle, \\
& \qquad act == \langle targetElementEquals\ access \rangle, \\
& \qquad res == \langle targetElementEquals\ g7356 \rangle |\!\rangle, \\
& genRTID(ruleC0) \mapsto \langle\!| tid == genRTID(ruleC0), env == \langle\rangle, \\
& \qquad sub == \langle targetElementEquals\ charlie \rangle, \\
& \qquad act == \langle targetElementEquals\ access \rangle, \\
& \qquad res == \langle targetElementEquals\ s39465 \rangle |\!\rangle, \\
& genRTID(ruleC9) \mapsto \langle\!| tid == genRTID(ruleC9), env == \langle\rangle, \\
& \qquad sub == \langle targetElementEquals\ charlie \rangle, \\
& \qquad act == \langle targetElementEquals\ access \rangle, \\
& \qquad res == \langle targetElementEquals\ s40566 \rangle |\!\rangle, \\
rootPol == \{ & PolSet(genPSID(\{policyA6, policyB3, policyC2\}))\} |\!\rangle
\end{aligned}
$$

The complete formal XACML definition above can now be transformed to the equivalent XACML policy set below. We leverage the transformation process defined in [79] (and briefly presented in Appendix B.7) to produce the machine-readable XACML policy set. We note that each time an evolution occurs, this entire XACML policy set is re-generated using our translation process, and then deployed.

```
<?xml version="1.0" encoding="UTF-8"?>
<PolicySet xmlns="urn:oasis:names:tc:xacml:1.0:policy"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:oasis:names:tc:xacml:1.0:policy cs-xacml-schema-policy-01.xsd"
  PolicyCombiningAlgId="urn:oasis:names:tc:xacml:1.0:policy-combining-algorithm:deny-overrides"
  PolicySetId="csxacml632">
  <Target>
    <Subjects>
      <AnySubject/>
    </Subjects>
    <Resources>
      <AnyResource/>
    </Resources>
    <Actions>
      <AnyAction/>
    </Actions>
  </Target>

```

```
<Policy PolicyId="pa6" RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:deny-overrides">
  <Target>
    <Subjects>
      <Subject>
        <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">alice</AttributeValue>
          <SubjectAttributeDesignator
            DataType="http://www.w3.org/2001/XMLSchema#string"
            AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
        </SubjectMatch>
      </Subject>
    </Subjects>
    <Resources>
      <AnyResource/>
    </Resources>
    <Actions>
      <AnyAction/>
    </Actions>
  </Target>

  <Rule RuleId="ra1" Effect="Permit">
    <Target>
      <Subjects>
        <Subject>
          <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">alice</AttributeValue>
            <SubjectAttributeDesignator
              DataType="http://www.w3.org/2001/XMLSchema#string"
              AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
          </SubjectMatch>
        </Subject>
      </Subjects>
      <Resources>
        <Resource>
          <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">g7351</AttributeValue>
            <ResourceAttributeDesignator
              DataType="http://www.w3.org/2001/XMLSchema#string"
              AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
          </ResourceMatch>
        </Resource>
      </Resources>
      <Actions>
        <Action>
          <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">access</AttributeValue>
            <ActionAttributeDesignator
              DataType="http://www.w3.org/2001/XMLSchema#string"
              AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
          </ActionMatch>
        </Action>
      </Actions>
    </Target>
  </Rule>

  <Rule RuleId="ra3" Effect="Permit">
    <Target>
      <Subjects>
        <Subject>
          <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">alice</AttributeValue>
            <SubjectAttributeDesignator
              DataType="http://www.w3.org/2001/XMLSchema#string"
              AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
          </SubjectMatch>
        </Subject>
      </Subjects>
      <Resources>
        <Resource>
          <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">g7353</AttributeValue>
            <ResourceAttributeDesignator
              DataType="http://www.w3.org/2001/XMLSchema#string"
              AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
          </ResourceMatch>
        </Resource>
      </Resources>
      <Actions>
        <Action>
          <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">access</AttributeValue>
            <ActionAttributeDesignator
```

```
                DataType="http://www.w3.org/2001/XMLSchema#string"
                AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
            </ActionMatch>
          </Action>
        </Actions>
      </Target>
    </Rule>

    <Rule RuleId="ra6" Effect="Permit">
      <Target>
        <Subjects>
          <Subject>
            <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">alice</AttributeValue>
              <SubjectAttributeDesignator
                DataType="http://www.w3.org/2001/XMLSchema#string"
                AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
            </SubjectMatch>
          </Subject>
        </Subjects>
        <Resources>
          <Resource>
            <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
              <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">g7356</AttributeValue>
              <ResourceAttributeDesignator
                DataType="http://www.w3.org/2001/XMLSchema#string"
                AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
            </ResourceMatch>
          </Resource>
        </Resources>
        <Actions>
          <Action>
            <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
              <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">access</AttributeValue>
              <ActionAttributeDesignator
                DataType="http://www.w3.org/2001/XMLSchema#string"
                AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
            </ActionMatch>
          </Action>
        </Actions>
      </Target>
    </Rule>
  </Policy>

  <Policy PolicyId="pb3" RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:deny-overrides">
    <Target>
      <Subjects>
        <Subject>
          <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">bob</AttributeValue>
            <SubjectAttributeDesignator
              DataType="http://www.w3.org/2001/XMLSchema#string"
              AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
          </SubjectMatch>
        </Subject>
      </Subjects>
      <Resources>
        <AnyResource/>
      </Resources>
      <Actions>
        <AnyAction/>
      </Actions>
    </Target>
  </Policy>

  <Policy PolicyId="pc2" RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:deny-overrides">
    <Target>
      <Subjects>
        <Subject>
          <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">charlie</AttributeValue>
            <SubjectAttributeDesignator
              DataType="http://www.w3.org/2001/XMLSchema#string"
              AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
          </SubjectMatch>
        </Subject>
      </Subjects>
      <Resources>
        <AnyResource/>
      </Resources>
      <Actions>
        <AnyAction/>
```

```
      </Actions>
    </Target>

    <Rule RuleId="rc0" Effect="Permit">
      <Target>
        <Subjects>
          <Subject>
            <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
              <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">charlie</AttributeValue>
              <SubjectAttributeDesignator
                DataType="http://www.w3.org/2001/XMLSchema#string"
                AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
            </SubjectMatch>
          </Subject>
        </Subjects>
        <Resources>
          <Resource>
            <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
              <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">s39465</AttributeValue>
              <ResourceAttributeDesignator
                DataType="http://www.w3.org/2001/XMLSchema#string"
                AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
            </ResourceMatch>
          </Resource>
        </Resources>
        <Actions>
          <Action>
            <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
              <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">access</AttributeValue>
              <ActionAttributeDesignator
                DataType="http://www.w3.org/2001/XMLSchema#string"
                AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
            </ActionMatch>
          </Action>
        </Actions>
      </Target>
    </Rule>

    <Rule RuleId="rc9" Effect="Permit">
      <Target>
        <Subjects>
          <Subject>
            <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
              <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">charlie</AttributeValue>
              <SubjectAttributeDesignator
                DataType="http://www.w3.org/2001/XMLSchema#string"
                AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"/>
            </SubjectMatch>
          </Subject>
        </Subjects>
        <Resources>
          <Resource>
            <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
              <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">s40566</AttributeValue>
              <ResourceAttributeDesignator
                DataType="http://www.w3.org/2001/XMLSchema#string"
                AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
            </ResourceMatch>
          </Resource>
        </Resources>
        <Actions>
          <Action>
            <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
              <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">access</AttributeValue>
              <ActionAttributeDesignator
                DataType="http://www.w3.org/2001/XMLSchema#string"
                AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
            </ActionMatch>
          </Action>
        </Actions>
      </Target>
    </Rule>
  </Policy>
</PolicySet>
```

D.3 Alloy Model: Alice Node 0

```
module EAC/csamp0

open EAC/metapolicy
open EAC/addb
open EAC/mpcondition
open EAC/system

open util/ordering [System]

-- basic types
one sig alice extends Subject {}
one sig g7350, g7351, g7352, g7353, g7354 extends Resource {}
one sig g7355, g7356, g7357, g7358, g7359 extends Resource {}
one sig rA0, rA1, rA2, rA3, rA4, rA5, rA6, rA7, rA8, rA9 extends RuleID {}
one sig access extends Action {}
one sig period extends External {}
one sig node0 extends MetaPolicyID {}
one sig mcA0, mcA1, mcA2, mcA3, mcA4, mcA5 extends MPConditionID {}
one sig mcA6, mcA7, mcA8, mcA9, mcA10 extends MPConditionID {}
one sig pA0, pA2, pA3, pA4, pA5, pA6, pA7, pA8, pA9 extends PolicyID {}
one sig sA2, sA3, sA4, sA5, sA6, sA7, sA8, sA9, end extends StateID {}
one sig period1, period2, period3, period4 extends StateID {}
one sig period5, period6, period7, period8, period9 extends StateID {}

-- metapolicy
one sig csamp extends MetaPolicy {} {
  mid = node0
  initialstate = sA2
  rule = {(rA0 → ruleA0) + (rA1 → ruleA1) + (rA2 → ruleA2) + (rA3 → ruleA3) +
          (rA4 → ruleA4) + (rA5 → ruleA5) + (rA6 → ruleA6) + (rA7 → ruleA7) +
          (rA8 → ruleA8) + (rA9 → ruleA9)}
  policy = {(pA0 → policyempty) + (pA2 → policyA2) + (pA3 → policyA3) +
            (pA4 → policyA4) + (pA5 → policyA5) + (pA6 → policyA6) +
            (pA7 → policyA7) + (pA8 → policyA8) + (pA9 → policyA9)}
  states = {(sA2 → pA2) + (sA3 → pA3) + (sA4 → pA4) + (sA5 → pA5) + (sA6 → pA6) +
            (sA7 → pA7) + (sA8 → pA8) + (sA9 → pA9) + (end → pA0) +
            (period1 → pA0) + (period2 → pA0) + (period3 → pA0) + (period4 → pA0) +
            (period5 → pA0) + (period6 → pA0) + (period7 → pA0) + (period8 → pA0) +
            (period9 → pA0)}
  transitions = {(sA2 → ((0 → psA1) + (1 → psA2) + (2 → psA3) + (3 → psA4) +
                         (4 → psA5) + (5 → psA6) + (6 → psA7) + (7 → psA8) + (8 → psA9))) +
                 (sA3 → ((0 → psA11) + (1 → psA12) + (2 → psA13))) +
                 (sA4 → ((0 → psA15) + (1 → psA16) + (2 → psA17))) +
                 (sA5 → (0 → psA15)) +
                 (sA6 → ((0 → psA11) + (1 → psA20) + (2 → psA21))) +
                 (sA7 → ((0 → psA11) + (1 → psA23) + (2 → psA24) + (3 → psA25))) +
                 (sA8 → (0 → psA16)) +
                 (sA9 → (0 → psA17)) +
```

```
                (period1 → (0 → psA10)) +
                (period2 → (0 → psA26)) +
                (period3 → (0 → psA14)) +
                (period4 → (0 → psA18)) +
                (period5 → (0 → psA19)) +
                (period6 → (0 → psA22)) +
                (period7 → (0 → psA26)) +
                (period8 → (0 → psA26)) +
                (period9 → (0 → psA26))}
}

-- priority pairs
one sig psA1 extends Priority {} {mpc = mcA1 and state = period1}
one sig psA2 extends Priority {} {mpc = mcA2 and state = period2}
one sig psA3 extends Priority {} {mpc = mcA3 and state = period3}
one sig psA4 extends Priority {} {mpc = mcA4 and state = period4}
one sig psA5 extends Priority {} {mpc = mcA5 and state = period5}
one sig psA6 extends Priority {} {mpc = mcA6 and state = period6}
one sig psA7 extends Priority {} {mpc = mcA7 and state = period7}
one sig psA8 extends Priority {} {mpc = mcA8 and state = period8}
one sig psA9 extends Priority {} {mpc = mcA9 and state = period9}
one sig psA10 extends Priority {} {mpc = mcA10 and state = sA3}
one sig psA11 extends Priority {} {mpc = mcA3 and state = end}
one sig psA12 extends Priority {} {mpc = mcA5 and state = sA5}
one sig psA13 extends Priority {} {mpc = mcA6 and state = sA8}
one sig psA14 extends Priority {} {mpc = mcA10 and state = sA4}
one sig psA15 extends Priority {} {mpc = mcA6 and state = end}
one sig psA16 extends Priority {} {mpc = mcA5 and state = end}
one sig psA17 extends Priority {} {mpc = mcA1 and state = end}
one sig psA18 extends Priority {} {mpc = mcA10 and state = sA5}
one sig psA19 extends Priority {} {mpc = mcA10 and state = sA6}
one sig psA20 extends Priority {} {mpc = mcA1 and state = sA5}
one sig psA21 extends Priority {} {mpc = mcA6 and state = sA9}
one sig psA22 extends Priority {} {mpc = mcA10 and state = sA7}
one sig psA23 extends Priority {} {mpc = mcA4 and state = end}
one sig psA24 extends Priority {} {mpc = mcA1 and state = sA8}
one sig psA25 extends Priority {} {mpc = mcA5 and state = sA9}
one sig psA26 extends Priority {} {mpc = mcA10 and state = end}

-- rules
one sig ruleA0 extends EACRule {} {
    rid = rA0 and subject = alice and resource = g7350 and effect = Permit and action = access
}

one sig ruleA1 extends EACRule {} {
    rid = rA1 and subject = alice and resource = g7351 and effect = Permit and action = access
}

one sig ruleA2 extends EACRule {} {
    rid = rA2 and subject = alice and resource = g7352 and effect = Permit and action = access
}
```

```
one sig ruleA3 extends EACRule {} {
    rid = rA3 and subject = alice and resource = g7353 and effect = Permit and action = access
}

one sig ruleA4 extends EACRule {} {
    rid = rA4 and subject = alice and resource = g7354 and effect = Permit and action = access
}

one sig ruleA5 extends EACRule {} {
    rid = rA5 and subject = alice and resource = g7355 and effect = Permit and action = access
}

one sig ruleA6 extends EACRule {} {
    rid = rA6 and subject = alice and resource = g7356 and effect = Permit and action = access
}

one sig ruleA7 extends EACRule {} {
    rid = rA7 and subject = alice and resource = g7357 and effect = Permit and action = access
}

one sig ruleA8 extends EACRule {} {
    rid = rA8 and subject = alice and resource = g7358 and effect = Permit and action = access
}

one sig ruleA9 extends EACRule {} {
    rid = rA9 and subject = alice and resource = g7359 and effect = Permit and action = access
}

-- policies
one sig policyA2 extends EACPolicy {} {
    pid = pA2 and rules = {(0 → rA1) + (1 → rA3) + (2 → rA4) + (3 → rA5) + (4 → rA6)}
}

one sig policyA3 extends EACPolicy {} {
    pid = pA3 and rules = {(0 → rA3) + (1 → rA5) + (2 → rA6)}
}

one sig policyA4 extends EACPolicy {} {
    pid = pA4 and rules = {(0 → rA5) + (1 → rA6)}
}

one sig policyA5 extends EACPolicy {} {
    pid = pA5 and rules = {(0 → rA6)}
}

one sig policyA6 extends EACPolicy {} {
    pid = pA6 and rules = {(0 → rA1) + (1 → rA3) + (2 → rA6)}
}
```

```
one sig policyA7 extends EACPolicy {} {
    pid = pA7 and rules = {(0 → rA1) + (1 → rA3) + (2 → rA4) + (3 → rA5)}
}

one sig policyA8 extends EACPolicy {} {
    pid = pA8 and rules = {(0 → rA5)}
}

one sig policyA9 extends EACPolicy {} {
    pid = pA9 and rules = {(0 → rA1)}
}

one sig policyempty extends EACPolicy {} {
    pid = pA0 and no rules
}

-- trigger sequences
one sig trigA0 extends Trigger {} {
  data = {(0 → g7350)}
}

one sig trigA1 extends Trigger {} {
  data = {(0 → g7351)}
}

one sig trigA2 extends Trigger {} {
  data = {(0 → g7352)}
}

one sig trigA3 extends Trigger {} {
  data = {(0 → g7353)}
}

one sig trigA4 extends Trigger {} {
  data = {(0 → g7354)}
}

one sig trigA5 extends Trigger {} {
  data = {(0 → g7355)}
}

one sig trigA6 extends Trigger {} {
  data = {(0 → g7356)}
}

one sig trigA7 extends Trigger {} {
  data = {(0 → g7357)}
}
```

```
one sig trigA8 extends Trigger {} {
  data = {(0 → g7358)}
}

one sig trigA9 extends Trigger {} {
  data = {(0 → g7359)}
}

one sig trigA10 extends Trigger {} {
  data = {(0 → period)}
}

-- mpconditions
one sig mpA0 extends MPCondition {} {
  mpid = mcA0 and trigger = {trigA0}
}

one sig mpA1 extends MPCondition {} {
  mpid = mcA1 and trigger = {trigA1}
}

one sig mpA2 extends MPCondition {} {
  mpid = mcA2 and trigger = {trigA2}
}

one sig mpA3 extends MPCondition {} {
  mpid = mcA3 and trigger = {trigA3}
}

one sig mpA4 extends MPCondition {} {
  mpid = mcA4 and trigger = {trigA4}
}

one sig mpA5 extends MPCondition {} {
  mpid = mcA5 and trigger = {trigA5}
}

one sig mpA6 extends MPCondition {} {
  mpid = mcA6 and trigger = {trigA6}
}

one sig mpA7 extends MPCondition {} {
  mpid = mcA7 and trigger = {trigA7}
}

one sig mpA8 extends MPCondition {} {
  mpid = mcA8 and trigger = {trigA8}
}
```

```
one sig mpA9 extends MPCondition {} {
  mpid = mcA9 and trigger = {trigA9}
}

one sig mpA10 extends MPCondition {} {
  mpid = mcA10 and trigger = {trigA10}
}

-- addb initial
one sig csaaddb extends ADDB {} {
    no data
}

-- domain specific signatures and functions
one sig Domain {
    values: Resource → Int
} {
    values = {g7350 → 5 + g7351 → 2 + g7352 → 5 + g7353 → 3 + g7354 → 4 +
                g7355 → 2 + g7356 → 1 + g7357 → 5 + g7358 → 5 + g7359 → 5}
}

-- system initial
pred init (s: System) {
    s.addb = csaaddb
    s.metapolicy = csamp
    s.current = csamp.initialstate
}

-- transitions is a trace of the policy evolution
fact transitions {
    init[first[]]
    all s: System - last[] | let s' = next[s] | observation [s, s']
}

-- find an instance
pred evolve {}

run evolve for exactly 5 System, 5 ADDB, 5 int, 9 seq

-- analyse properties

-- determinism
assert determinism {
all s: System | {
    (all i: dom[s.metapolicy.transitions] |
        all disj j, k: dom[s.metapolicy.transitions[i]] |
            first[s.metapolicy.transitions[i][j]] != first[s.metapolicy.transitions[i][k]])
            }
}

check determinism for exactly 5 System, 5 ADDB, 5 int, 9 seq
```

```
assert uniquepriority {
    all s: System | {
        all disj i,j: ran[s.metapolicy.transitions[s.current]] | i != j
        }
}

check uniquepriority for exactly 5 System, 5 ADDB, 5 int, 9 seq

-- connectedness
assert connectedness {
    all s: System | {
        (dom[s.metapolicy.states] - s.metapolicy.initial) in ran[s.metapolicy.transitions].state
        }
}

check connectedness for exactly 5 System, 5 ADDB, 5 int, 9 seq

-- restriction
assert restriction {
  all s: System - last[] | let s' = next[s] | {
      observation [s, s'] implies
          (accessibleResources [s'.metapolicy, (getPolicy [s'.metapolicy, s'.current])] in
              accessibleResources [s.metapolicy, (getPolicy [s.metapolicy, s.current])])
              }
}

check restriction for exactly 5 System, 5 ADDB, 5 int, 9 seq

-- verify consistency

-- binary
assert binary {
    all s: System | {
        all r: accessibleResources[s.metapolicy, (getPolicy[s.metapolicy, s.current])] |
            r !in s.addb.data.elems
            }
}

check binary for exactly 5 System, 5 ADDB, 5 int, 9 seq

-- period
assert period {
    all s: System - last[] | let s' = next[s] | {
      observation [s, s'] implies
        (s'.addb.data.last = period implies
            no accessibleResources[s.metapolicy, (getPolicy[s.metapolicy, s.current])]
            and
            #accessibleResources[s.metapolicy, (getPolicy[s.metapolicy, s'.current])] >= 0)
            }
}
```

```
check period for exactly 5 System, 5 ADDB, 5 int, 9 seq

-- subscription
assert subscription {
    all s: System | {
        all r: accessibleResources[s.metapolicy, (getPolicy[s.metapolicy, s.current])] |
            ((sum n: s.addb.data.elems | Domain.values[n])) + Domain.values[r] =< 5
            }
}

check subscription for exactly 5 System, 5 ADDB, 5 int, 9 seq

Executing "Run evolve for 5 int, 9 seq, exactly 5 System, 5 ADDB"
   Solver=sat4j Bitwidth=5 MaxSeq=9 SkolemDepth=1 Symmetry=20
   202043 vars. 8784 primary vars. 646917 clauses. 2920179ms.
   Instance found. Predicate is consistent. 506ms.

Executing "Check determinism for 5 int, 9 seq, exactly 5 System, 5 ADDB"
   Solver=sat4j Bitwidth=5 MaxSeq=9 SkolemDepth=1 Symmetry=20
   212210 vars. 8825 primary vars. 681413 clauses. 2957243ms.
   No counterexample found. Assertion may be valid. 387ms.

Executing "Check uniquepriority for 5 int, 9 seq, exactly 5 System, 5 ADDB"
   Solver=sat4j Bitwidth=5 MaxSeq=9 SkolemDepth=1 Symmetry=20
   211150 vars. 8841 primary vars. 664776 clauses. 2963923ms.
   No counterexample found. Assertion may be valid. 899ms.

Executing "Check connectedness for 5 int, 9 seq, exactly 5 System, 5 ADDB"
   Solver=sat4j Bitwidth=5 MaxSeq=9 SkolemDepth=1 Symmetry=20
   207258 vars. 8789 primary vars. 656989 clauses. 2956395ms.
   No counterexample found. Assertion may be valid. 394ms.

Executing "Check restriction for 5 int, 9 seq, exactly 5 System, 5 ADDB"
   Solver=sat4j Bitwidth=5 MaxSeq=9 SkolemDepth=1 Symmetry=20
   205586 vars. 8789 primary vars. 653351 clauses. 2858049ms.
   Counterexample found. Assertion is invalid. 462ms.

Executing "Check binary for 5 int, 9 seq, exactly 5 System, 5 ADDB"
   Solver=sat4j Bitwidth=5 MaxSeq=9 SkolemDepth=1 Symmetry=20
   204514 vars. 8799 primary vars. 651461 clauses. 2853156ms.
   No counterexample found. Assertion may be valid. 560ms.

Executing "Check subscription for 5 int, 9 seq, exactly 5 System, 5 ADDB"
   Solver=sat4j Bitwidth=5 MaxSeq=9 SkolemDepth=1 Symmetry=20
   208564 vars. 8799 primary vars. 668068 clauses. 2861171ms.
   No counterexample found. Assertion may be valid. 687ms.

Executing "Check period for 5 int, 9 seq, exactly 5 System, 5 ADDB"
   Solver=sat4j Bitwidth=5 MaxSeq=9 SkolemDepth=1 Symmetry=20
   231765 vars. 8800 primary vars. 750717 clauses. 3575365ms.
   No counterexample found. Assertion may be valid. 689ms.
```

8 commands were executed. The results are:

#1: **Instance** found. evolve is consistent.

#2: No counterexample found. determinism may be valid.

#3: No counterexample found. uniquepriority may be valid.

#4: No counterexample found. connectedness may be valid.

#5: **Counterexample** found. restriction is invalid.

#6: No counterexample found. binary may be valid.

#7: No counterexample found. period may be valid.

#8: No counterexample found. subscription may be valid.

D.4 Results: Alice, Bob, Charlie Analysis

	Metapolicy Properties			Policy Consistency					
Node	Deterministic	Connected	Restricted	Binary	Period	Subscription	States	Atoms	Average Time (min)
0	Yes	Yes	No	Yes	Yes	Yes	18	176	48.7
1	Yes	Yes	No	Yes	Yes	Yes	22	200	51.0
2	Yes	Yes	No	Yes	Yes	Yes	18	176	48.7
3	Yes	Yes	No	Yes	Yes	Yes	20	186	49.9
4	Yes	Yes	No	Yes	Yes	Yes	19	185	49.4
5	Yes	Yes	No	Yes	Yes	Yes	22	200	51.0
6	Yes	Yes	No	Yes	Yes	Yes	24	215	52.2
7	Yes	Yes	No	Yes	Yes	Yes	18	176	48.7
8	Yes	Yes	No	Yes	Yes	Yes	18	176	48.7
9	Yes	Yes	No	Yes	Yes	Yes	18	176	48.7

The analysis results of Alice's metapolicy

	Metapolicy Properties			Policy Consistency					
Node	Deterministic	Connected	Restricted	Reject	Liveness	System	States	Atoms	Average Time (s)
0	Yes	Yes	Yes	Yes	Yes	Yes	2	56	2.1

The analysis results of Bob's metapolicy

	Metapolicy Properties			Policy Consistency					
Node	Deterministic	Connected	Restricted	Counting	CM	Emergency	States	Atoms	Average Time (s)
0	Yes	Yes	No	Yes	Yes	Yes	11	105	47.2

The analysis results of Charlie's metapolicy