Policy Conflict Detection Using Alloy: An Explorative Study

by

Shahin Sheidaei

B.Sc., Iran University of Science and Technology, 2006

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

In the School of

Interactive Arts and Technology

Simon Fraser University

Summer 2010

Approval

Name: **Shahin Sheidaei**

Degree: **Master of Science**

Title of Thesis: **Policy Conflict Detection Using Alloy: An Explorative Study**

Examining Committee:

Chair:

_____

Dr. Steve DiPaola
Associate Professor, School of Interactive Arts and Technology

_____

Dr. Marek Hatala
Senior Supervisor
Associate Professor, School of Interactive Arts and Technology

_____

Dr. Dragan Gašević
Supervisor
Associate Professor, Athabasca University

_____

Dr. Halil Erhan
External Examiner
Assistant Professor, School of Interactive Arts and Technology

Date Defended/Approved: April 12, 2010

Declaration of
Partial Copyright Licence

Abstract

Policy conflicts are inevitable in policy-based systems. Handling conflicts is considered to be so vital in policy-based system, that several policy languages introduced built-in functions to handle them.

In this thesis, we investigate an innovative approach for policy conflict detection. We investigate inclusion of MDE (Model Driven Engineering) concept in the policy conflict detection method. We inspect the practicality of analysing policies along with policy language's meta-model in order to detect conflicts. We will examine feasibility of policy conflict detection using Alloy and PML (Policy Modelling Language).

In our work, we systematically explore ways of modelling policies in Alloy. We have successfully introduced proper modelling approach for policy conflict detection and analysis of the policies according to PML meta-model. However, we have also shown that a one-pass analysis of detecting conflicts in addition to analysing policies according to the PML meta-model is not achievable.

"To My Parents & My Sister"
–for their support and encouragement throughout my entire life.

Acknowledgment

I am honoured and delighted to acknowledge the many people whose counsel, support, and encouragement have contributed immensely to the completion of my thesis.

First and foremost, I would like to express my deep gratitude to my senior supervisor, Dr. Marek Hatala, whose constant encouragement played the most important role in keeping me on the track of my studies.

When lots of pressure had made it almost impossible for me to keep up with my studies, his constant support and understanding of the situation were the main incentives for me to keep going.

My sincere appreciations also go to Dr. Dragan Gašević for his constant guidance, feedback, and fortitude during my studies. His motivating attitudes and his extreme patience in bearing with my interruptions to his own work gave me the great opportunity to fully understand and grasp the research ideas.

Finally, I would like to thank my great friends and lab mates, Ty Mey Eap, Dr. Carlo Torniai, Nima Kaviani, Bardia Mohabbati and Melody Siadaty for bearing with me and providing a great research and social atmosphere at school.

Table of Contents

List of Figures

List of Tables

1 Introduction

Changing business processes such as observed in insurance, telecommunication, transportation, travel industries require software solutions to be updated frequently. In order to adapt the changes, the business practices need systems that can be reconfigured without a need in their software design and implementation. As an answer to this need, policy-based approaches are suggested. A policy-based system, executes operations interpreting modifiable policy collections structured following specific policy schemas. A policy is a business rule that defines a choice in the behaviour of a system [13] and separates business rules from software implementation [3]. This system approach makes policy definition and policy languages a promising solution for the systems used where regular business process changes take place.

Policy-based systems present various issues when interaction of independently designed systems becomes a requirement. For example, expansion of organisational structures entails communication of the systems designed for different purposes or organizations. Making these systems interact is a challenging task. The particular challenge emerges when the policies in a system change without considering the policies in the others that they interact. The inconsistencies and conflicts between policies call for additional mechanisms by which potential policy conflicts can be detected before the new policies implemented. System design should always anticipate these policy conflicts.

Before policy conflicts are resolved within a system, they should be detected first. Detection of conflicts within a policy-based system needs analysis of concrete policies within the system. By concrete policies within this thesis, we mean policies that represent real policies in the system in the real world. Concrete is a term we borrowed from Model Driven Engineering Method (which will be described in section [2.8]), where concrete refers to instances in the model rather than meta-model. The analysis tries to detect these conflicts mainly based on the semantics and structure of concrete policies. Talking about semantics and structure of a language, Model Driven Engineering (MDE) contains concept of meta-model. Utilizing a meta-model of the language in the analysis process connected deeply to semantics and structure of a language seems a rational choice. In this thesis, we will investigate analysis of a policy-based system while concrete policies with meta-model of the underlying policy language are present in the model.

Alloy is a software abstraction language. Alloy has been gaining a lot of attention lately. It has been applied in many different domains for various practices. Using the modelling capabilities of Alloy, one can make a model of a system and analyse it. This model, if analyzed, could reveal information about system's shortcomings. Considering policy conflicts, one hypothesis is to model and analyse concrete policies using Alloy.

In the Section 2 a brief introduction of policy languages, policy conflicts, Alloy and Model Driven Engineering is given. In Section 3, problem addressed in this thesis will be discussed in detail. In Section 4, we will discuss the methodology used in this thesis. Section 5 describes different approaches used to model policies in Alloy. Section 6 provides a summary of all approaches' outcomes with in depth discussion and conclusions will be presented in Section 7.

2 Literature Review

In the first part of this Section (Section 2.1), we will briefly introduce policy languages in general. Then four different policy languages including XACML, KAoS, Rei and Ponder will be introduced (Section 2.2). Section 2.3 describes Policy Modelling Language (PML), its background and its relation to other policy languages. Following that, different policy conflict approaches will be discussed in Section 2.5 and 2.6. Section 2.7 contains an overview of Alloy modelling language and its usages in various research studies. In Section 2.8, we will discuss concept of Model Driven Engineering (MDE).

2.1 Policies

2.1.1 Definition

A policy is a rule that defines a choice in the behaviour of a system [13]. A policy can also be defined as a "statement enabling or constraining execution of some type of action by one or more actors when some specific conditions take place" [12]. Changing policies can easily change the behaviour of a system without the need for any alteration in the implementation or deployment of the existing software. All the work in the background is done by a part of the system generally called "policy engine".

Policy is generally defined over some domain. Domain provides a flexible mean of categorizing objects in a system. This categorization can be based on different factors such as geographical boundaries, object type, class of a resource etc. A domain, in definition, is similar to a "Set". Like "Set", domain has nesting feature. A domain may

contain other domains as well. The child domain would be called sub-domain. An object of sub-domain would be an object of parent domain.

Policies are defined as a relationship between subjects and targets [3]. “Subject” refers to users, principals or automated manager components that have management responsibility. A “Target” refers to resources in a domain. A subject interacts with the target by invoking methods accessible through the target’s interface. For example, a target can be a document within a system and its interface can include an access method available in order for a subject to use it.

2.2 Policy Languages

In this section, we discuss the existing policy languages XACML, Ponder, Rei and KAoS, and their built-in conflict detection and resolution approaches. XACML was selected since it is broadly used and is widely accepted in the industry. Rei and KAoS were selected because they are representative of different underlying logics; i.e. computational logic and descriptive logic respectively. Ponder was selected because of being a declarative object-oriented language. Ponder’s syntax, compared to other three policy languages, are closer to the programming languages (such as JAVA).

2.2.1 XACML: eXtensible Access Control Markup Language

XACML policies present access control policies using XML [45]. OASIS (Organization for the Advancement of Structured Information Standards) is the committee that standardized the latest version of XACML policy language as of 2005 [14]. XACML policy language expresses policies for information access over the

Internet. XACML policies have the structure of subject-target-action-condition. The subject can be identity, group or a role. The target object can refer to a single element within an XML document. XACML introduces three policy elements, Rule, Policy and PolicySet [14]. The Rule is a Boolean expression. Policy is a set of Rule elements and PolicySet contains various Policy elements.

A Rule is the basic element of a policy. It identifies an authorization constraint that can exist in the policy in which it is included. A Rule is composed of a target, effect and a set of conditions. Target identifies the set of requests to which the rule applies. An effect is either “permit” or “deny”, if “permit” is chosen, Rule grants access and in case “deny” is chosen, Rule prohibits access. A set of conditions specifies when the rule applies to a request.

A Policy is a combination of one or more Rules. A Policy contains a target (same as the Rule target), a set of rules, and a rule combination algorithm. Rule combination algorithm will be discussed in Section 2.6.1.

As multiple Rules compose a Policy, multiple Policies compose a PolicySet. PolicySet represents the conditions to apply in case the decision has to take into account requirements specified by multiple parties, or in case the decision can be made via different approaches. A PolicySet is defined by a target, a set of Policy (ies) (or other PolicySets), and a policy combination algorithm. The policy combination algorithms proposed is the same as rule combination algorithms but only for policies. XACML policy language supports roles.

2.2.2 Rei

Rei is a policy framework that enables specification, analysis, and reasoning about policies. Rei was introduced as an outcome of Me-Centric project in 2002 [10]. Rei is based on Prolog, a declarative and rule-based logic programming language [24]. Thus, policies can be represented in Prolog syntax if required. In addition, Policies can also be expressed in RDF[1] [10].

Rei supports right, obligation, prohibition and dispensation types of policies**.** We will discuss different policy types in Section 2.4. The policy language contains meta-policy specifications for conflict resolution. These include constructs for specifying precedence of modality and priority of policies.

Rei's concepts of right, permission, obligation, dispensation, and policy rules are represented as Prolog predicates. There is also a Graphical User Interface (GUI) provided for Rei. Similar to functionality of Prolog, a feature available in Rei is to create and use variables. Rei also permits users to specify role-based access control policies or policies relating not only to individuals but also to groups of entities.

2.2.3 Knowledgeable Agent-Oriented System (KAoS)

KAoS is a framework that provides policy and domain management services [12]. KAoS policy language has been used in variety of distributed computing applications. KAoS has been used for Semantic Web Services Workflow Composition [7], "Grid Policy Management", "Coalition Search" and "Rescue and the Semantic Firewall" as three application of KAoS [8]. KAoS also provides domain services, reasoning and

[1] Resource Description Framework (RDF) is a W3C standard [57] created to standardize defining and using metadata, in the format that can be easily associated with resources and shared on the web.

representation of DAML-based policies in collaboration with Nomads[2] [9]. KAoS policy language allows for the specification, management, conflict resolution and enforcement of policies. Policies in KAoS are represented in a semantic web ontology[3] language called OWL[4] [50].

KAoS support four types of policy: Permission (PositiveAuthorization in KAoS), Prohibition (NegativeAuthorization in KAoS), Positive Obligation (PositiveObligation in KAoS) and Negative Obligation (NegativeObligation in KAoS).

Along with KAoS a graphical tool, called KAoS Policy Administration Tool (KPAT), is provided. KPAT assists users in the policy specification and application. In addition, KPAT also can detect and resolve conflicts within newly defined policies. As policies, domains, and application entities are defined using the KPAT, the appropriate policy representations (in KAoS syntax) are generated automatically in the background insulating the user from having to know KAoS syntax or from coding directly in it as in Rei and Ponder.

Policies can also be created using KAoS's API in addition to the KPAT GUI. KPAT guides a user through a creation process using ontology defined ranges to always narrow user choices to the most appropriate set of values; only these valid in the given context.

[2] Nomads combines the capabilities of Aroma, an enhanced Java compatible Virtual Machine (VM), with the Oasis agent execution environment.

[3] Ontology is a formal explicit description of concepts in a domain [42].

[4] Web Ontology Language (OWL) is a semantic language for publication and sharing of ontologies on the Web.

2.2.4 Ponder

Ponder is a declarative, object-oriented language for specifying security and management policies [13]. Ponder allows general security policies to be specified. Damianou et. al. [13] define a policy as a set of rules that defines a choice in the behaviour of a system.

Ponder's supported policy types are authorisation, obligation, delegation, information filtering and refrain policies. Detail description of different types of policies in Ponder will be given in Sections 2.4.1 and 2.4.2.

Ponder distinguishes between basic and composite policies. A basic policy is considered a rule governing choices in system behaviour and is specified by a declaration that includes a set of subjects and a set of targets. These sets are used to define the managed objects that the policy operates over.

Ponder composite policies facilitate policy management in large, complex enterprise systems. They provide the ability to group policies and structure them to reflect organisational structure, preserve the natural way system administrators operate or simply provide reusability of common definitions. This simplifies the task of policy administrators.

Ponder introduces a domain browser as an integrated tool. The Ponder domain browser provides a user interface for all aspects of an integrated management environment. It can be used to group or select objects for applying, monitoring or to perform management operations on policies.

Another integrated tool for Ponder is policy editor tool. Policy editor tool is integrated with both the domain browser and the Ponder compiler. It provides an easy to use development environment for specifying, reviewing and modifying policies.

The Ponder compiler maps policies to low-level representations suitable for the underlying system or into XML. Ponder is capable of converting policies using three different methods: The first method uses a Java back-end interface, which transforms Ponder authorization policies into access control policies for the Java platform. The second method translates Ponder authorization policies to Windows 2000 security templates and Firewall rules. The last method can map Ponder authorization policies to Linux access controls.

2.3 PML: Policy Modelling Language

As a new and emerging approach in modelling policy languages, this Section introduces Policy Modelling Language (PML) [17]. PML is a general policy modelling language intended to provide a bridge between various policy languages.

Kaviani et al. [17] have designed a modelling language for policies. As one of its features, PML introduces the capability of policies being visually presented. REWERSE II Rule Markup Language (R2ML) [18] is a language that is designed to represent policies, rules and enable rule interchange. R2ML provides a support for PML goal to bridge between policy languages. R2ML has a graphical syntax (for representing rules) via UML-based Rule Modelling Language (URML) [40]. PML has used this feature for visual presentation of policies, which are represented as rules in R2ML. Designers of PML language, by introducing PML, have tried to integrate definition of policies into the

software development process. Authors argue that there has not been enough value put on the role of policies in the definition, design and integration of software systems. Thus, they proposed PML as a modelling language for policies. One of the main factors of PML design that authors have pointed out is the capability of "easily representing and integrating policies with other pieces of software at the design time".

PML provides a common ground between different policy languages by adapting and conforming to Model Driven Engineering (MDE) methods [39] (c.f. Section 2.8). MDE's main aim is to make software reusability possible by providing means to transform a single design (or model) into different platforms. PML consists of different parts. Following the MDE approach, PML consists of a meta-model which defines the abstract syntax of the language, a UML profile which is a graphical syntax for PML, XML-bases syntax to present policies and a set of transformation that provide transformation between PML and other policy languages.

PML has a layered architecture consisting of three layers. The first layer is to express policy language's logic (which was modelled as computational logic or description logic). The second layer, which is called "General Policy Concepts, is describing policy concepts shared across multiple policy languages. These concepts are the four concepts of permission, prohibition, obligation and dispensation (will be discussed in Section 2.4). The third layer, Language Specific Concepts, is to model the specific types of policies in any policy language. Figure 1 visualises three layers of PML.

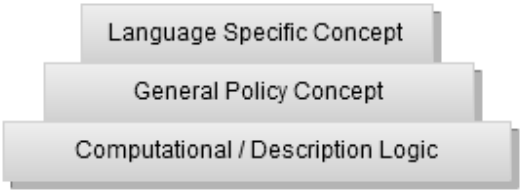


Figure 1 - PML Layers

Another key characteristic of PML is that it is specifically designed to support modelling and interchange of policies by generalizing main policy types and characteristics observed in major policy languages existent today. For instance, the transformation between KAoS and Rei and PML has been shown in [11].

2.4 Policy Types

In this section, different policy types will be introduced. Any policy language supports variety of policy types. We introduce policy types in two different categories, primitive policy types and non-primitive policy types. Primitive policy types are policy types that are commonly covered and supported by different policy languages, i.e. Permission, Prohibition, Positive Obligation and Negative Obligation. The other types of policies will be referred to as non-primitive policy types in this thesis.

2.4.1 Primitive Policy Types

According to [3] two major categories of policies can be distinguished: obligation and authorization. Obligation and Authorization within themselves can be divided to positive and negative. A positive obligation forces an actor (subject) to do an action on

the target (object) of the domain. Negative obligation obliges actor not to do something. Authorization policies give authorization to an actor while bringing no obligation with them. They also have both negative and positive sides. The main usage of Authorization policies is essentially for access control policies, to protect resources and services from unauthorized access. Obligation policies are event-triggered, events can be simple, i.e. an internal failure of a service event, or an external event notified by monitoring service components, e.g. a temperature exceeding a threshold or a component failing.

Authorisation Plus policies are also called Permission policies while Authorisation Minus policies are referred to as Prohibition policies. Obligation (in Rei) and Dispensation are the same as positive and negative obligation. We will refer to Positive / Negative Authorisation policies using A+/A- and to Positive / Negative Obligation using O+/O-.

Other policy types exists other than those introduced in this section; we refer to them as non-primitive policy types. These types of policies typically address a specific need and are usually used within one or a limited number of policy languages. In the following section, we will briefly introduce both primitive and non-primitive policy types in XACML, Rei, KAoS and Ponder policy languages.

2.4.2 Primitive and non-Primitive Policy types

Ponder supports authorisation, obligation, delegation, information filtering and refrain policies [13]. Authorization and Obligation policies are described earlier in Section 2.4.1. Information filtering policies are policies that are used to transform the information input or output parameters in an action. Basically, a filter policy checks input

and output conditions (set by the policy writer) and based on those conditions different responses will be given. For example, a request to locate a person within a company is asked. This request will be responded based on an information filtering policy defined within the system. The response to this request will contain detail information of the location if it has been asked from a person within that company. On the other hand, it will only contain a yes/no answer to an outsider to indicate if that person is present in the company or not. Delegation policies, often used in access control systems, are to transfer access rights temporarily. A delegation policy is always associated with an authorisation policy, which specifies the access rights to be delegated. Negative delegation policies forbid delegation. Refrain policies define actions that subjects must refrain from performing (i.e. must not perform) on target objects even though it may actually be permitted to perform the action within the system. Refrain policies are similar to negative authorisation policies, but are enforced by subjects rather than target access controllers. They are used for situations where negative authorisation policies are inappropriate because the targets are not trusted to enforce the policies (e.g., they may not wish to be protected from the subject). Table 1 presents a summarization of supported policy types in XACML, KAoS, Rei and Ponder policy languages in addition to PML. Prior to Table 1 four sample policies are defined below.

- Permission Policy: An Actor is permitted to access a resource.
- Prohibition Policy: An Actor is prohibited from accessing a resource.
- Positive Obligation Policy: An Actor is obliged to access a resource, after a specified event within the system.

- Negative Obligation: An Actor is obliged not to access a resource, after a specified event within the system.

Table 1 - Policy Type Support in Policy Languages

	Permission (A+)	Prohibition (A-)	Positive Obligation (O+)	Negative Obligation (O-)	Other Supported Types
XACML	✓	✓	✓	✓	
KAoS	✓	✓	✓	✓	
Rei	✓	✓	✓	✓	
Ponder	✓	✓	✓	✓	delegation, information filtering and refrain
PML	✓	✓	✓	✓	

2.5 Policy Conflicts

Policies within a domain might generate incompatible responses to a request, resulting in conflicts. A policy conflict is inconsistency among policies [3]. These inconsistencies can arise when multiple policies apply to the same object. For example, an authorization (positive authorization) policy may define an action in a system where another authorization policy forbids it to be accomplished (negative authorization). Cases like this in a system are considered as policy conflicts. As an example in health care domain, task of prescribing a person with a drug might conflict with an external policy from an insurance company that forbids usage of that specific drug. There might be more

than one actor in charge of completing a task. Considering systems that have more than one actor (responsible for this task) can be a source of this conflict. Policy inconsistency can arise because of omissions, errors or conflicting requirements for different managers specified in policies. Interactions of actors within a system such as introducing new resources, defining new policies and change in policies can be counted as a source of conflict. Definition of new policies without considering existing policies within a system could also be a source of policy conflict. Effect of merging of two policies, if supported within a system, can be counted as another possible source for policy conflicts. Whatever the cause of the conflict is, the important thing is that it needs to be resolved.

When we are talking about conflicts among policies, we mean the conflicts among policies in one domain. A Policy conflict may occur where there is more than one applicable policy in a domain. If the policies have different outcomes, policy conflict will happen. In other words, conflict among policies arise when we have overlapping policies, these overlaps may be in different parts of the policy. Conflicts generally occur if the policies are about the same action, on the same target but different modalities (authorization, obligation).

Conflicts can be categorized in different ways considering different aspects of a policy. According to Moffett and Sloman [3] four different categories of policy conflicts can be defined. The categorization, presented in Table 2, is based on what parts of policy might overlap and cause conflicts.

Table 2 - Policy Conflict types (Moffett and Sloman [3])

Policy Conflict types based on overlapping elements of a policy
Subject overlap
• Policies associated with subject object with inconsistent outcome • E.g. Two different policies associated for a specific actor, being inconsistent with each other
Target object overlap
• Policies associated with target object with inconsistent outcome • E.g. Two different policies associated for a specific target, being inconsistent with each other
Double overlap (target and subject overlap)
• Policies associated with both target object and subject with inconsistent outcome • E.g. Two different policies associated for a specific actor and target, being inconsistent with each other
Subject – Target overlap (subject of one and target of the other one)
• Policies associated with target object or subject with inconsistent outcome • E.g. Two different policies associated one for a specific actor and the other one for a specific target, being inconsistent with each other

Conflicts can also be categorized according to what in a business process can be the source of the conflict [3]. In Table 3 this categorization and its cause are briefly introduced.

Table 3 - Policy Conflict Causes

Type of conflict	**Situation**
Conflict of interest	A person is assigned to handle tasks from different domains (s/he has two different kinds of responsibility which may conflict)
Conflict of duties	A person needs to accomplish a task, which usually takes two roles to finalize it.
Conflict of priorities	A resource is available but the request for that resource is larger than its availability, thus a priority conflict happens

Similar to policies, policy conflicts are related to the underlying domain. One might categorize policy conflicts from a different point of view and based on the domain under study. For an example, we consider a role-based system. Policy conflicts can be categorized in a role-based system according to the aspect of occurrence of policy conflict, which would result in Table 4. According to [4], four different categories of conflicts and their possible way of detection can be distinguished.

Table 4 - Policy Conflict types in a role-based system (Dunlop et. al [4])

Type of conflict	**Possible occurrence reasons and detection approaches to them**
Internal Policy Conflict	Policies assigned to roles are not compatible with each other.
	Detection of this type of conflict is possible when a new role is added to the system.
External Policy Conflict	External roles exist in the system.
	May be detected when a new user is assigned to a role and/or when a policy is assigned to a role.
Policy Space[5] Conflict	Two or more policy space manage the same set of subjects[6] and attempt to enforce different and conflicting policies over them.
	Detected when a new space is initially identified at the runtime or when a new policy is assigned to a role.
Role Conflict	A user obtains a set of incompatible role assignments
	Detection of this conflict requires ensuring that users are not operating with a union of privileges, which are determined to be incompatible.[7]

2.5.1 Modality Conflicts

Policy conflicts can be classified considering primitive policy types (as described in Section 2.4.1). Inconsistencies in policy specifications may result in conflicts among policies. Six different combinations from these four primitive policy types can be distinguished, O+/O- , O+/A+, O+/A-, O-/A-, O-/A+, A+/A-. Three of these six combinations will result in conflict. This type of policy conflict, which refers to the conflict between positive and negative policies applying to the same object, is called

[5] Policy space refers to a set of policies defined within a domain.

[6] Subjects can be assigned to different domains. Thus, they can be managed by different policies from different domains.

[7] For example, a user has a permission over the resource A through one role while having prohibition access over the same resource A through another role.

modality conflicts [46]. Modality conflicts can be categorized in three categories as follow:

- Permission (A+) / Prohibition (A-): A subject has permitted and prohibited to perform an action on a target object at the same time.
- Positive Obligation (O+) / Negative Obligation (O-): A subject is obliged to perform an action on a target while at the same time is obliged not to do that action.
- Positive Obligation (O+) / Prohibition (A-): A subject (actor) is obliged to perform an action on a target while at the same time is prohibited from doing that action.

It simply can be seen that the remaining combinations of policy types will not end up in conflicts. For example, Permission / Positive Obligation (O+/A+) policy types are not inconsistent with each other. Positive Obligation is obliging a subject to perform an action on a target object, as the Permission policy type is permitting the subject (Actor) to perform that action.

2.5.2 Static / Dynamic Conflict Detection

Different strategies for detecting policy conflicts can be categorized by considering when the detection process is applied. Depending on the time of application, two different classes of policy conflict detection approaches can be defined: static and dynamic. The static approaches are considered the approaches that are being applied at the design and specification phase. The static methods are used to analyze elements of the

system and their interaction based on a formal description. Static conflict detection method can also be utilized to be used at run-time. In this case, the method uses a snapshot of a system created at a certain moment and detects the conflicts at that moment. The advantage of using static policy conflict detection is that the solution to the conflict is known before it happens (considering a solution provided at the time of static conflict detection by domain manager / policy manager) [6].

The dynamic detection method is the other class of conflict detection methods. The dynamic detection methods take information about the running system and make an up-to-date image of the system. The dynamic detection method figures out (reasons) if conflicts have taken place based on the image and upcoming events in the system. As the main advantages of this method, the information can be gathered as the system is running. Usually this type of conflict detection has some default strategy defined to deal with the policy conflict. However, some conflicts cannot be resolved and should be reported to the manager of the system.

Dynamic (run-time) conflict detection methods usually consist of series of acts that should not be performed at the same time, e.g. an act for positive obligation and prohibition (negative authorization) policies. Run-time conflict detection approach will detect a conflict based on monitoring a system for these series of acts. Once a conflict has been detected, based on the default strategy of a system, a solution to that conflict will be provided. The outcome of the resolution of each conflict resolution can be different. Not all conflicts should be detected at run-time. Some conflicts might rise due to design errors in the system similar to compile time errors in programming. These conflicts should be detected using a static approach.

The disadvantage of using static conflict detection is since the system is not functional, some of the conflicts detected might never appear in the system and therefore there is no need to dedicate resources to resolve them. However, on the other hand this feature can also be considered an advantage of static conflict detection. If Static conflict detection method cannot find any possible conflicts, then the policy-based system can be considered a conflict free system. Dynamic policy conflict detection only relies on the real world situations and by using some predefined rules tries to overcome the policy conflict situation at the time when they happen. Dynamic policy conflict detection can be used best with a system to log conflicts and notify the manager about those conflicts for future reference.

2.5.3 Policy Conflict Resolution

The solution to policy conflicts could be defined via automated responses set by domain manager or an individual response for each occurrence from a human manager. A human administrator may not notice all the conflicts happening in a system while a formal approach to a policy conflict detection and resolution could resolve all the conflicts aroused in a system.

Kamoda et. al stated that different approaches to detect and resolve conflicts in each domain might be needed [5]. As in each domain, meaning of the conflict concept can be different. In spite of this fact, solutions to policy conflict detection / resolution can benefit from some general approaches for the policy conflict resolution step.

One of the solutions is to give policies or modalities priorities [3]. In the case of conflict among set of policies, the policy (or set of policies) with highest priority would

be selected. An alternative solution could be to consider policies (from the set of conflicting policies) in a sequential order until the first encounter of conflict happens. This ordering can be based on the priorities or other distance factors that a domain manager defines over the domain. Another possible solution involves definition of a distance between an object and policies. This distance can be defined based on the domain hierarchy. For instance, a policy defined for a sub-domain can be defined to be nearer in distance to an object than a policy that is defined for the parent domain. As an example in a university domain, an instructor is associated with a policy granting access to all students' records while there is a policy forbidding access to every student's record for everyone within that domain. An instructor, being a member of both instructors and everyone in that domain, is nearer to the policy granting him access compared to the one forbidding him. The concept of distance should always be thought of as a measurement for relevance of a policy. The nearer a policy to an object, the more relevant it is to that object. The nearest policy's outcome could be selected as final outcome of policy conflict resolution.

Above presented methods are some solutions for policy conflict resolution method. Sometimes general solutions cannot resolve all conflicts in the specific domain. In this case, specific conflict resolution mechanism is needed.

2.6 Policy Conflict and Policy Languages

Conflict detection approaches cannot be discussed without taking into consideration the underlying policy languages and the specific needs of the systems. Most of policy languages have some built in functions to deal with policy conflicts. This

section discusses tools and approaches that handle policy conflicts in previously introduced policy languages, XACML, Rei, KAoS and Ponder. PML supports four types of policies. However, looking at the PML language, one can easily notice that there is no mechanism for policy conflict detection or resolution. There also does not exist any previous work on detection of conflicts among policies in PML.

2.6.1 XACML

In terms of policy conflict approaches, XACML has an internal functionality called the rule-combining algorithm [14]. The rule-combining algorithm defines a procedure for the authorization decision based on evaluation of different rules or policies. Domain manager is responsible for the choice of rule-combining algorithm for set of policies or the whole domain. There are various choices embedded in XACML as standard algorithms. These algorithms are Deny-Overrides, Ordered-Deny-Overrides, Permit-Overrides, First-Applicable and Only-One-Applicable. Rule-combining algorithms combine the effects of all the rules in a policy to arrive at a final authorization decision.

In the Deny-Override case, if one Rule or Policy evaluates to “Deny”, regardless of other elements (i.e. other Rules or Policies), the final result is “Deny”. The same applies for the Permit-Override but to the “Permission” evaluation of one <Rule> or <Policy>. Ordered-Deny-Overrides is similar to Deny-Override with an exception that relevant rules are order and evaluated. The evaluation order is the same as the order in which the policies (or rules) are added in the policy. As for the First-Applicable, the first Rule or Policy that applies to the request would be evaluated and returned as the result.

Only-One-Applicable can be applied only for Policy and PolicySet and is not applicable for Rules. It returns “NotApplicable” if no policy applies to the request. It returns “Undetermined” if more than one Policy or PolicySet applies to the request.

It is also possible in XACML to define one’s own algorithm to resolve policy conflicts. Figure 2 presents a sample rule-combining algorithm to detect policy conflicts in XACML. The policy is only applicable to the requests from "SampleServer"[8] server. The Policy has a Rule with a Target that requires an action of "login"[9] and a Condition[10] that applies only if the Subject is trying to log in between 9 am and 5 pm. It has the second rule as well. If the first Rule provided here does not apply, then a default Rule is used that always returns Deny. The selection for the Rule-combining algorithm was set to be for Permit-Overrides. This means that if only one rule (or policy) in the set of policy (or policy set) permits access, the final outcome would be permit. The default outcome of this policy would be “Deny” since the last rule’s effect without any condition is “Deny”. However, if the first rules’ condition was satisfied the first rule effect would be “Permit”. A simple policy conflict would happen between these two policies. However, since the rule-combining algorithm is permit overrides, the final effect of the policy would be “Permit” since at least one rule’s effect is “Permit”.

[8] “SampleServer” has been indicated by first element of <attributeValue> in the definition of policy in Figure 2, line 11 of the policy counting comment lines.

[9] “login” has been indicated by second element of <attributeValue> in the definition of policy in Figure 2, line 34 of the policy counting comment lines.

[10] “Condition” has been indicated by first element of <Condition> in the definition of policy in Figure 2, line 42 of the policy counting comment lines.

```
<Policy PolicyId="SamplePolicy"
        RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:permit-overrides">

  <!-- This Policy only applies to requests on the SampleServer -->
  <Target>
    <Subjects>
      <AnySubject/>
    </Subjects>
    <Resources>
      <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">SampleServer</AttributeValue>
        <ResourceAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string"
                                     AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
      </ResourceMatch>
    </Resources>
    <Actions>
      <AnyAction/>
    </Actions>
  </Target>

  <!-- Rule to see if we should allow the Subject to login -->
  <Rule RuleId="LoginRule" Effect="Permit">

    <!-- Only use this Rule if the action is login -->
    <Target>
      <Subjects>
        <AnySubject/>
      </Subjects>
      <Resources>
        <AnyResource/>
      </Resources>
      <Actions>
        <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">login</AttributeValue>
          <ActionAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string"
                                     AttributeId="ServerAction"/>
        </ActionMatch>
      </Actions>
    </Target>

    <!-- Only allow logins from 9am to 5pm -->
    <Condition FunctionId="urn:oasis:names:tc:xacml:1.0:function:and">
      <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:time-greater-than-or-equal"
        <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:time-one-and-only">
          <EnvironmentAttributeSelector DataType="http://www.w3.org/2001/XMLSchema#time"
                                        AttributeId="urn:oasis:names:tc:xacml:1.0:environment:current-time"/>
        </Apply>
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#time">09:00:00</AttributeValue>
      </Apply>
      <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:time-less-than-or-equal"
        <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:time-one-and-only">
          <EnvironmentAttributeSelector DataType="http://www.w3.org/2001/XMLSchema#time"
                                        AttributeId="urn:oasis:names:tc:xacml:1.0:environment:current-time"/>
        </Apply>
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#time">17:00:00</AttributeValue>
      </Apply>
    </Condition>

  </Rule>
  <!-- A final, "fall-through" Rule that always Denies -->
  <Rule RuleId="FinalRule" Effect="Deny"/>

</Policy>
```

Figure 2 - XACML Policy, presenting rule-combining algorithm in action

2.6.2 Rei

Rei's mechanism to deal with policy conflict is utilizing Meta-policies [10]. Specifically, meta-policies are policies about how policies are defined and interpreted, and how conflicts are resolved.

Meta policies in Rei control conflicting policies statically in two ways, by specifying priorities and precedence between policies. In Rei, every policy can be associated with an identifier. Identifiers are used to apply the priority or precedence mechanism over conflicting policies. With a use of a special overrides construct in Rei one can override the priorities between any two rules. For example, if rule A1 is giving Mark the right to print and rule B1 is prohibiting Mark from printing, by using overrides(A1, B1), the conflict between the two rules can be resolved as A1 will be given priority over B1. Figure 3 shows this example in Rei's syntax.

```
 a1**has(mary, right(print, [time-now(12.00)]))
b1**has(mary, prohibition(print, [lab-member(X, ai)]))
 overrides(a1, b1)
```

Figure 3 – An example in Rei, presenting Meta-Policy use in last line

As of precedence, it is possible to specify which modality holds precedence over the other in meta-policies. The domain manager (or the policy designer) can associate certain precedence for a set of actions satisfying the associated conditions. The special constructs for precedence in Rei are metaRuleAction and metaRuleAgent.

As an example, consider a meta-policy presented in Figure 4. This meta-policy specifies that negative modality holds precedence for all employees of Xerox Labs, i.e. if conflicts are because of negative and positive policies applying at the same time; the policy with the negative modality has the precedence over other policies.

```
metaRuleAgent([employee(X, xeroxLabs)], negative-modality)
```

Figure 4 – Meta-Policy, an example of Precedence in Rei

2.6.3 KAoS

KAoS resolves the conflicts using a precedence concept. Policy precedence conditions are needed to properly execute the automatic conflict resolution algorithm. When policy conflicts occur, precedence conditions are used to determine which of the two (or more) policies is the most important. The conflict can then be resolved automatically in favour of the most important policy. Alternatively, the conflicts can be brought to the attention of a human administrator who can make the decision manually.

There are three types of policy conflicts that can be handled in KAoS: positive vs. negative authorization; positive vs. negative obligation; and positive obligation vs. negative authorization [12]. We will discuss these types of policy conflicts not only in KAoS buy in any policy language in section [2.5.1].

KAoS is also capable of detecting potential conflicts between policies at design time such as when a user tries to add a new policy. The KAoS conflict detection uses algorithms based on the Stanford's Java Theorem Prover (JTP) [47]. KAoS identifies the policy clashes by using the special mechanisms and tries to resolve conflicts by ordering

policies according to their precedence. Figure 5 presents a policy in KAoS. As it can be seen, the <policy:hasPriority> element is referring to priority of the policy. In case a policy conflict occurs, policy P1 has the highest priority.

```
<policy:NegAuthorizationPolicy rdf:ID="P1">
<policy:controls rdf:resource="#P1Action" />
<policy:hasSiteOfEnforcement rdf:resource="&policy;ActorSite" />
<policy:hasPriority>1</policy:hasPriority>
<policy:hasUpdateTimeStamp>446744445544</policy:hasUpdateTimeStamp>
</policy:NegAuthorizationPolicy>
```

Figure 5 - KAoS Policy priorities - KAoS mechanism to handling policy conflicts

2.6.4 Ponder

Similar to KAoS and Rei, Ponder uses meta-policies to deal with policy conflicts by specifying policies for groups of policies [13]. One usage of Meta-policies in Ponder is to disallow the simultaneous execution of conflicting policies. A meta-policy in Ponder is specified as a sequence of OCL[11] [41] expressions the last one of which must evaluate to true or false. Based on the domain, policies and the whole system, the Meta-policies can be used differently to resolve conflicts. Figure 6 presents a sample meta-policy in Ponder. The meta-policy discussed here is intended to check if a policy is authorising a manager to retract policies for which he is the subject. This might happen in a single policy with overlapping subjects and targets. This can be expressed in Ponder as follows:

[11] Object Constraint Language (OCL) is a declarative language for describing rules that apply to Unified Modelling Language (UML) models.

```
inst meta selfManagement1 raises selfMngmntConflict (pol) {
[pol] = this.authorisations -> select (p | p.action->exists (a |a.name = "retract" and
a.parameter -> exists (p1 |p1.oclType.name = "policy" and p1.subject = p.subject))) ;
pol->notEmpty ;
}
```

Figure 6 – Meta-Policy in Ponder

The body of the meta-policy contains two OCL expressions. One must have knowledge of OCL in order to understand how the Meta-Policy has been described in Figure 6. In a nutshell, it select all p where the name is “retract” and while it has the “policy” that specific p is the subject of another policy.

2.7 Alloy

Alloy is a formal specification language based on first-order relational logic [20]. Alloy has been utilized to explore abstract software models and to assist in finding flaws in these models. Alloy models are used to analyse systems under study. Alloy models are based on statements written in terms of atoms and relations between atoms. Any property or behaviour is expressed as a constraint using set, logical and relational operators.

Alloy Analyzer as part of Alloy is a “model-finder” tool that uses a constraint solver (based on SAT solver technology [20]) to analyze models written in Alloy. The Alloy Analyzer translates constraints from Alloy model into Boolean constraints, which are fed to an off-the-shelf SAT solver.

There are two types of analysis offered by Alloy Analyzer. One is Simulation and the other one is Checking. Simulation involves finding instances of a model satisfying the

model specification. Checking involves finding counterexample instances to the model specification. When the Alloy Analyzer succeeds in finding a solution to a formula, it produces both graphical and textual output of the solution. To make instance finding feasible, a user may specify a scope for the analysis of a model. The scope puts a bound on how many instances of an entity may be observed in as instance of model. Thus limits the number of instances of model to be examined. If no specific number of entities is provided, Alloy Analyzer uses a default number for all entities in the model. A complete definition of Alloy language is being introduced in Appendix 2 (Section 11).

Alloy has been used in different studies and various researches. In Sections 2.7.1 to 2.7.4, various projects and researches in which Alloy was involved are discussed. Nevertheless, usage of Alloy is not only bounded to the ones described here. To best of our knowledge, the most relevant researches to this thesis's goal are reviewed and debated in the following sections. A comprehensive discussion regarding comparison of this thesis's approach with the following approaches will be given in Outcome and Discussion, Section 6.

2.7.1 Alloy and UML based modelling

This section provides discussion over usage of Alloy in studies included UML models. Jacqueline et. al. in [21] used Alloy in order to analyze a MOF-compliant meta-model. The meta-model was expected to be used for the "measurement of coupling and cohesion metrics in Object-Oriented systems". The meta-model was expressed using UML and OCL. Authors translated their meta-model into Alloy model. Then, they have used Alloy Analyzer to generate sample instances. These sample instances were used to

assist them in improving their meta-model and thus fix any possible flaws in the meta-model specification. For example, they used Alloy Analyzer to generate random instances of the meta-model that conform to the well-formedness rules.

Mostefaoui and Vachon in [33] used Alloy to analyze a UML profile. The UML profile that they were analyzing is called Aspect-UML. Aspect-UML is a profile introduced in their research. Aspect-UML is used within Aspect-Oriented (A-O) programming. A-O introduces concept “aspects” which allows developers to modify the behaviour of a base program, something similar to the role of policies in a system. In their work, authors have used Alloy to check the model for conflicting aspect interactions. They translated concrete models into Alloy and not the meta-model of Aspect-UML. In their paper, authors have described modelling steps in detail. In their modelling steps, they provide presentation for elements of the UML-Profile in Alloy. Afterwards, they have used assertions in Alloy for the analysis.

2.7.2 Alloy and Policy Languages

In this section, the focus is on investigating different usages of Alloy utilized for modelling policy languages. Several studies have been conducted related to XACML policy language. To the best of our knowledge, studies related to other discussed policy languages (i.e. Ponder, KAoS and Rei) using Alloy have not taken place. Thus, it leaves us with only XACML as the major policy language to discuss in this section.

Martin and Xie in [26] presented an approach in which test cases were generated for XACML policies. They have introduced a tool called Cirg that generates test cases based on change-impact analysis. The Cirg is not using Alloy but as authors suggested,

they are expecting to use Alloy as an alternative to Cirg and in the future work for generating these tests. Likewise, authors used an approach to synthesise Verified Access Control Systems in XACML [27]. In the presented approach, they generated XACML policies. These policies were presented formally (Using a language called RW [52]) and were also checked for conflicts. The conflicts detected were the first type of conflict, Permission / Prohibition type of conflict (as discussed in Section 2.5.1). In their work, authors introduced a framework to translate the Alloy generated policies into the XACML format. The authors are suggesting that they can use Alloy for the verification method of their approach as a future work. Kolovski et. al. in [36] provide a formalization of XACML using description logics (DL). Having XACML represented in DL, authors easily used it to compare, verify and query the policies in XACML. Hughes and Bultan in [28] translated XACML policies to Alloy model and checked their properties using the Alloy Analyzer. The authors translated access control policies to a simple form that partitions the input domain to four classes: permit, deny, error, and not applicable. Then they expressed XACML policies with several ordering relations. These relations are used to specify the properties of the policies and the relationships among them. Followed by that, they expressed an approach to check these ordering relations within Alloy and using Alloy Analyzer. Using Alloy analyzer, they check if a combination of XACML policies does or does not reproduce the properties of its sub-policies. Accomplishing all these steps, they finally concluded that automated verification of XACML policies is feasible.

2.7.3 Alloy and Role-Based Access Control

This section is about practices of Alloy in several studies related to the Role Based Access Control (RBAC) systems. RBAC systems share the same basis with policy languages in a sense that they both can express restricting / granting access to a specific object in a system.

Toahchoodee et. al. in [23] utilized Alloy to analyze a UML model of a RBAC model. In their work, they have presented a new RBAC model that includes contextual information such as time and location unlike the traditional RBAC model. The UML model presented application and its access control requirements in a formal specification language. Alloy was used in order to automatically verify the UML model. They argue, "Although formal analysis can be done on UML specifications that are augmented with OCL constraints, there is not much tool support for automated analysis." Moreover, this is the reason they use UML2Alloy[12] to convert their model into Alloy model and use it to analyze the model.

Schaad and Moffett in [24] utilized Alloy to analyze different extension of RBAC models. These extensions include RBAC96 and RBAC97. Per authors' claim, having these extensions in a system might cause conflicts. As stated in the paper, there are some prerequisite conditions to check in order to avoid conflicts but authors believe there is a need for a "framework for the specification and analysis of role-based access control models and required constraints". In their approach, they present a model for each of the extensions (RBAC96 and RBAC97) and their subsections. Following that, they define a

[12] UML2Alloy is a tool for transforming UML models into Alloy model. More about this tool will be discussed in Section 4 as we are using it in this thesis.

set of constraints that is needed to be held true[13]. Then they have provided these concepts (constraints) presented in Alloy. Later on, they have used assertion to check if these concepts (constraints) can be held in according to different extensions of RBAC.

Hu and Ahn in [35] proposed a methodology to support automatic analysis for access control systems, in a framework called Assurance Management Framework (AMF). In their research, authors attempted to verify formal specifications of a role-based access control model. They used one of the profiles of RBAC[14] as their RBAC model and proposed an Alloy model to expresses it. Based on that model they utilized Alloy Analyzer to analyze their model and find out any counterexamples to refine the specification of the role-based access model presented in their study.

2.7.4 Various studies using Alloy

Other research studies in which Alloy is utilized are discussed in this section. The range of researches varies from modelling Java Authentication and Authorization Service (JAAS) and Policy based systems to modelling different studies about ontologies.

Schaeffer et. al. in [22] formally represented the underlying system as an Alloy model. The model was intended to be used for "Policy-Based Self-Managed Cell Interactions". As the title suggests the SMC (Self-Managed Cell) interaction are policy based. In their approach, authors have used Alloy to formally represent SMC interactions. They have defined various signatures (in Alloy model) regarding for elements within their system. For each operation in their system, they used an Alloy construct

[13] The constraints, as defined in their study, are Static Separation of Duty (SSoD), Dynamic Separation of Duty (SDSoD) and the Operational Separation of Duty (OpSoD).
[14] NIST/ANSI RBAC [35]

“Predicate”. They used the Alloy Analyzer to verify the consistency of SMC collaborations, i.e. interaction among Self-Managed Cells. To the best of our understanding within their model, they used Alloy to check the model for the existence of the corresponding authorisation policies for any obligation policy (one of the conflict types as discussed in Section 2.5.1). In the “Model-Checking and Policy Analysis” section of their research paper, they are mentioning that Type checking is needed to be done for the elements of their model (policies). They have mentioned that the Type checking is done using Predicates and an example of a Predicate has been presented in their paper**.**

Nakajima and Tamai in [31] used Alloy to analyze the system designed in JAAS framework. JAAS is a security framework in JAVA. Based on the model presented in the paper, they analyze the framework in three different categories. In different categories, i.e. different scenarios of JAAS framework modelled in Alloy, they use different assertions in Alloy to check on the consistency of the model.

Layouni et. al. in [25] used Alloy to detect conflicts in a language called APPEL. The definition of a conflict is based on the pre/post conditions in the APPEL language. They defined three categories of conflicts: concurrency conflict, disabling conflict and result conflicts. The categorization of conflicts is based on preconditions and post-conditions conflicts and overlapping of them (a double overlap conflict as presented in Table 2). They also explained different situations in which a conflict can appear. Different situation in which a conflict is appearing is checked using Alloy assertions. Checking of these assertions and not getting any counterexample, authors concluded consistency of the system.

Alloy has also been utilized in studies in which ontologies has been used. Dong et. al. in [34] have utilized Alloy to analyze web ontologies. Authors believe that FaCT and RACER, as existing ontology reasoners, have been developed to reason about ontologies with a high degree of automation. Dong et. al. argue that complex ontology-related properties may not be expressible within the current web ontology languages. Hence, they propose their approach to use Alloy to analyze web ontologies. They present models for DAML+OIL and RDF in Alloy. The model presents everything in Resource signature in Alloy. Class is a simple extension of Resource signature. Property is also a signature, having a sub_val value for a relation between signatures presenting value of a property. For each Property (such as hasValue, subPropertyOf ...), authors have defined a function or predicate in Alloy. The authors used a combination of Alloy analyzer and RACER for checking the inconsistency of ontology. Firstly, RACER is used to automatically determine the consistency of the ontology. If the ontology is inconsistent, a small partition of concepts in the ontology closely-related to the offending concept(s) were chosen and Alloy Analyzer was used to check for the source of the inconsistency. Wang et. al. in [37] used almost the same approach specifically for OWL.

The usage of Alloy is not limited only to the studies discussed above. Alloy has also been used to analyze cryptographic primitives, security protocols and Application Programming Interfaces (APIs) in [29], to analyze exception flow in software architecture. Filho et. al in [30] and Shaffer et. al in [32] used Alloy to perform analysis of the Domain Model to automatically detect potential security policy violations in the system.

2.8 Model Driven Engineering

The Model Driven Engineering (MDE) is a promising approach for software developers, suggesting that one should first develop a model of the system under study, and then transforms it into an executable software entity (e.g. deployed code) [53].

For better understanding of MDE, we can compare it to the object-oriented paradigm. In Object-Oriented paradigm, the main principle is that everything is an object while in MDE everything is considered to be a model. Considering classes, objects, instantiation and inheritance as major concepts in object-oriented technology, MDE introduces relations between a model, meta-model, model transformations, representation and conformance relations. In the following sections a brief introduction on model, meta-model and model transformation will be given.

2.8.1 Model

Models play a major role in MDE. A model is a simplified view of reality or, more specifically, a model is a set of statements about a system under study. In fact, one can say that a model is a clear set of formal elements that describes something being developed for a specific purpose and that can be analyzed using various methods. A model in MDE must possess the following five key characteristics of Abstraction, Understandability, Accuracy, Predictiveness and Inexpensiveness [53]. UML is used to present Models.

2.8.2 Meta-model

A meta-model is a model that defines the language for expressing a model. In fact, a meta-model is a specification model. Meta-models are used to validate models

represented in a specific language. That is, a meta-model makes statements about what can be expressed in the valid models of a certain modelling language. Generally, a meta-model is any language specification written in English, such as OWL language specification or UML specification [53].

Meta-models are used as abstract syntax for modelling languages. That is, meta-models specify rules for structuring sentences in modelling languages. This implies that each model needs to be conformant to the meta-model of the modelling language in which the model is specified.

2.8.3 Model Transformation

Model transformation is the process of converting one model to another model of the same system. One can consider model transformation as a process in which a target model will be automatically generated from a source model, according to a transformation definition. The transformation definition itself is also expressed as a model transformation language.

In fact, a model transformation means converting an input model (or a set of input models) which conforms to one meta-model, to another model(s), which conforms to another meta-model. This conversion is done by defining rules that match and/or navigate elements of source models resulting in the production of elements of the target model. The transformation itself is a model, which conforms to some transformation meta-model. Model transformation can be done using a language called Query View Transformation (QVT) [54]. Object Management Group (OMG) has introduced this language.

3 Problem specification

Throughout Section 2, we have introduced different policy languages, policy conflict and two major categories of policy conflict detection approaches (i.e. static and dynamic). In addition, as presented in Section 2.6, each policy language's approach to handle policy conflicts (i.e. policy conflict detection and/or resolution methods) has been discussed. These approaches are not simillar among all policy languages. For example, meta-model was used by both KAoS and Rei policy languages while XACML has its own approach called rule-combining algorithm to handle policy conflicts. Looking deeper into these approaches, we realise that they are mostly for the resolution part of handling policy conflicts. Since policy conflict detection is a domain related issue and one could not detect policy conflict without considering domain information. It is understandable why policy languages have not developed a policy conflict detection method.

As mentioned earlier in Section 2.7, detecting conflicts one needs to analyse policies within the system. In this thesis, we have chosen Alloy for policy conflict detection analysis. Alloy has recently attracted significant attention in the community and has been used in a variety of researches (as presented in Section 2.7). Alloy has been used to analyze XACML policies and detect conflicts among them [36]. In the mentioned study, a unique mapping from XACML elements to Alloy elements has been introduced. Using the introduced mapping, authors created the Alloy model and then used it for analysing XACML policies. One can find similar approaches of providing mappings into Alloy in other reviewed policy conflict detection studies. In these studies (as relevant studies discussed in Section 2.7), the authors typically used mappings to model concrete

policies in Alloy, but without a complete definition of the policy syntactical and static semantic constraints.

In this thesis, we introduce a novel conflict detection method. We investigate an approach in which the concept of Meta-model (introduced in section 2.8.2) has been utilized in the policy conflict detection process. Meta-model contains semantic and structural information about a language and its instances. This is a natural requirement, as policy detection also depends on the definition of the policy language and specific statements stated in the policy language definition. No major work has been done to integrate Meta-model of a policy language while analysing concrete policies (i.e. instances of the meta-model). Using a policy language meta-model, we believe that we can benefit from the semantics and structure defined in the meta-model in order to detect policy conflicts.

Prior to this study**,** Kelsen and Ma [55] have provided a lightweight semantics of Modelling Languages in Alloy. However, no representation of policy modelling language in Alloy has been introduced yet. In this thesis, our choice of modelling language is PML. PML, as described in Section 2.3, conforms to the principals of the Model Driven Engineering (MDE), thus has a meta-model defined. The goal of this thesis is to investigate the feasibility of conflict detection between policies presented in PML using Alloy. The research question we are trying to answer is whether Alloy can be utilized to detect policy conflicts using concrete policies and meta-model present in one Alloy model.

4 Methodology

This section discusses systematic steps carried out in this thesis in detail. In this section, methodology of detecting policy conflicts using Alloy will be discussed. In brief, PML has been used to represent policies. PML meta-model has also been developed in Alloy to be used in the process of detecting conflicts. Alloy language was used for modelling and Alloy Analyzer was used to detect conflicts among policies. In this process, a tool called UML2Alloy was used for transforming PML meta-model to Alloy. In Section 4.1, Alloy language and choice of Policy Modelling Language and in Section 4.2, choices of conflict detection algorithm are explained. In Section 4.3, the methodology used in this thesis is described in detail. The process we followed branches according to different modelling approaches. Various modelling approaches will be described thoroughly in Section 5.

4.1 Modelling

Alloy is an abstraction modelling language and its main components are signatures and relations [20]. In this thesis, Alloy has been used to model policies and to detect conflict among policies. Alloy comes with a tool called Alloy Analyzer. In this thesis, Alloy Analyzer was used to generate sample concrete policies from the model and to find policy conflicts (through its “Check” algorithm as discussed in Section 2.7). The model used by Alloy Analyzer has been created through a transformation from UML representation, which will be discussed in Section 4.3. The Alloy model contains concrete policies and PML meta-model (in some of the modelling approaches in Section 5). Since we investigate policy conflicts at design time, the conflict detection method

proposed in this thesis is categorized as static conflict detection method. Therefore, no real data can be assumed to exist within the system. Thus, a way to generate sample data in the system is needed. This need would be satisfied by generation of instances from the Alloy model. The instances would be representing concrete policies, which have been generated by Alloy Analyzer.

Another feature of Alloy Analyzer has also been utilized to help us determining the exact state of instances in the model, in which counterexample (in case of using assertion command in Alloy) or an example (in case of using predicate command in Alloy) has taken place[15]. This feature, in general, enables researchers to specify the conditions (values) of instances in an Alloy model when certain circumstance is encountered, whether it is a counterexample or an example as a part of a model. Assume a conflict has been detected by Alloy Analyzer, the information provided by Alloy Analyzer on the exact values of instances (involved in this conflict) will provide valuable information. Based on the provided information, domain manager can deal with conflicting policies by revising the conflicting policies or by providing a resolution for that situation.

In this thesis PML meta-model was utilized through its UML representation. The UML representation of PML meta-model is transformed using UML2Alloy tool. Steps of this transformation will be thoroughly described in section 4.3.

[15] Definition of Predicate and Assertion can be found in the Appendix, Section 9.3

4.2 Policy conflict detection method

As discussed earlier in Section 2.5.2, there are two major categories of conflict detection algorithms. One is static conflict detection and the other one is dynamic conflict detection. The approach used in this thesis is applied at the design time and generates different states of a system and it is categorized as static conflict detection. This approach with some changes can be altered to be used as dynamic conflict detection method. We will discuss this issue briefly in Section 6.

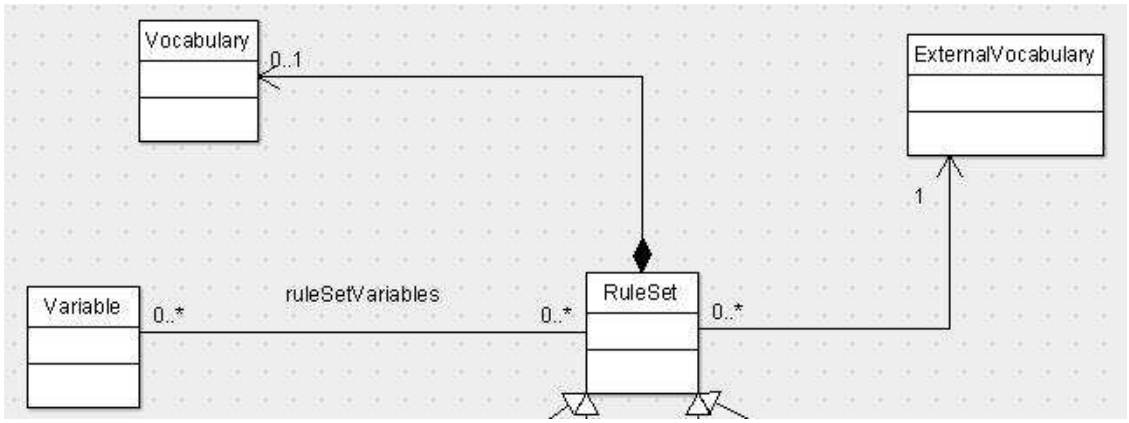
4.3 Process

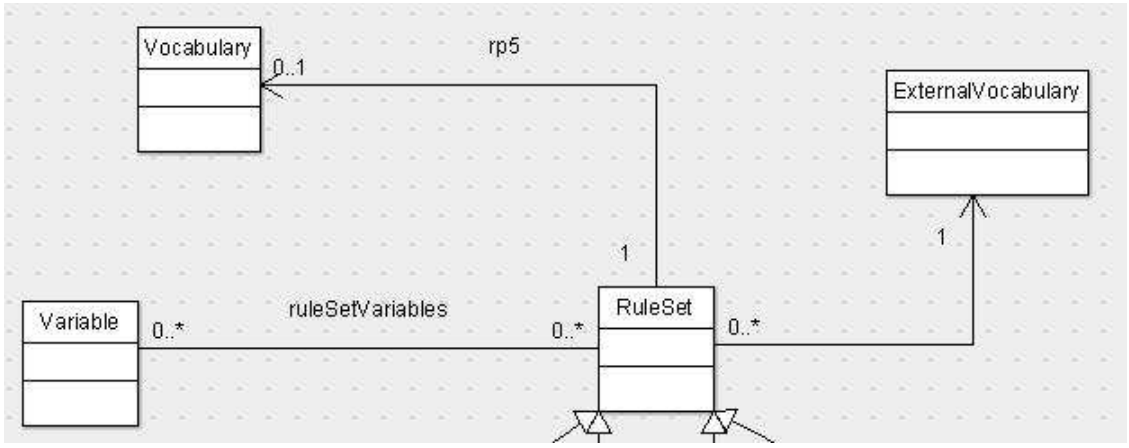
PML meta-model has been transferred into the Alloy model. The transformation of PML meta-model in Alloy can be called the first step in this thesis. This transformation has been performed using UML2Alloy. As its name suggests, UML2Alloy is a tool that performs the transformation of UML models into Alloy models. UML2Alloy accepts XMI[16] files as input. The files with XMI extensions can be created using a tool called AgroUML. ArgoUML is an open source tool for designing and editing UML models [44].

UML2Alloy is in its early versions. Therefore, some UML elements are not acceptable by UML2Alloy and cannot be transferred into Alloy model. Two elements that affect our work are aggregation and composition relations. UML2Alloy is not recognizing aggregation / composition relations in its current version. However, there is an alternative to these relations. The alteration will be to express the

[16] XMI stands for XML Metadata Interchange, a standard by Object Management Group (OMG) for exchanging information via XML [56].

composition/aggregation relations in some other way. As suggested in [16] aggregation and composition relations can be expressed using OCL constraint. Using the approach presented in [16], we transform each aggregation / composition relation in PML meta-model into OCL constraints. Hence, the transformations of the following classes in the PML model were affected: ObjectTerm, Slot, Term, Vocabulary, AndOrNafNegFormula, DataTerm, EventExpression, Logical Formula, ProductionRule, ReactionRule, DerivationRule, RuleSet, VocabularyEntry, Atom, DataLiteral, IntegrityRule and Implication. These classes have at least one composition/aggregation associated with them. Figure 7 shows a sample transformation of a composition transformed into OCL format. The upper part of the figure presents the composition and the lower part is showing the presentation in OCL.





OCL #1

context Vocabulary inv voca1 : Vocabulary . allInstances -> forAll (p : Vocabulary | RuleSet . allInstances -> exists (a : RuleSet | a . rp5 -> includes (p)))

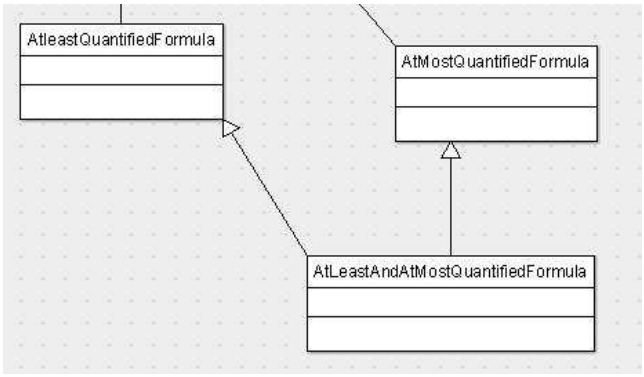
OCL #2

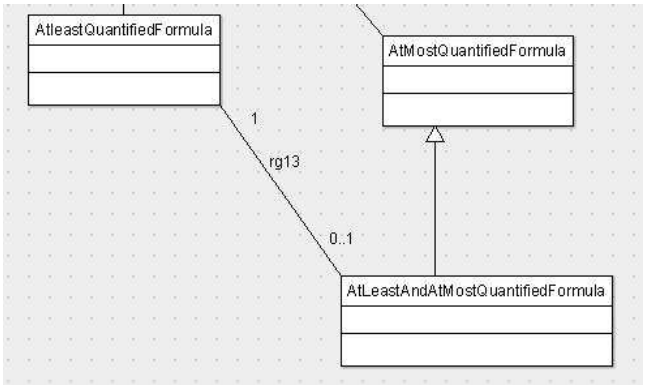
context Vocabulary inv voca2 : Vocabulary . allInstances -> forAll (p : Vocabulary | RuleSet . allInstances -> forAll (a1 , a2 : RuleSet | (a1 . rp5 -> includes (p) and a2 . rp5 -> includes (p)) implies a1 = a2))

Figure 7 – Transformation of composition in to OCL

UML2Alloy cannot also transform classes that have been generalized from more than one class. In order to circumvent this limitation, once again, we follow the suggestion from [16] and express generalization relations using OCL constraint. In this

regard, the following classes in the PML model were affected: StrongNegation, ReferencePropertyAtom, Property, ObjectVariable, ObjectName, NegationAsFailure, GenericVariable, GenericEntityName, Disjuction, Conjunction, DataVariable, Atom, and AtLeastAndAtMostQuantifiedFormula. These classes have more than one generalization associated with them. Figure 8 shows a sample transformation of generalization transformed into OCL. The upper part of the figure shows a sample generalization and the lower part shows the transformation into OCL.





context AtLeastAndAtMostQuantifiedFormula inv propg1 :
AtLeastAndAtMostQuantifiedFormula . allInstances -> forAll (p1 , p2 :
AtLeastAndAtMostQuantifiedFormula | (p1 <> p2) implies p1 . rg13 <> p2 . rg13)

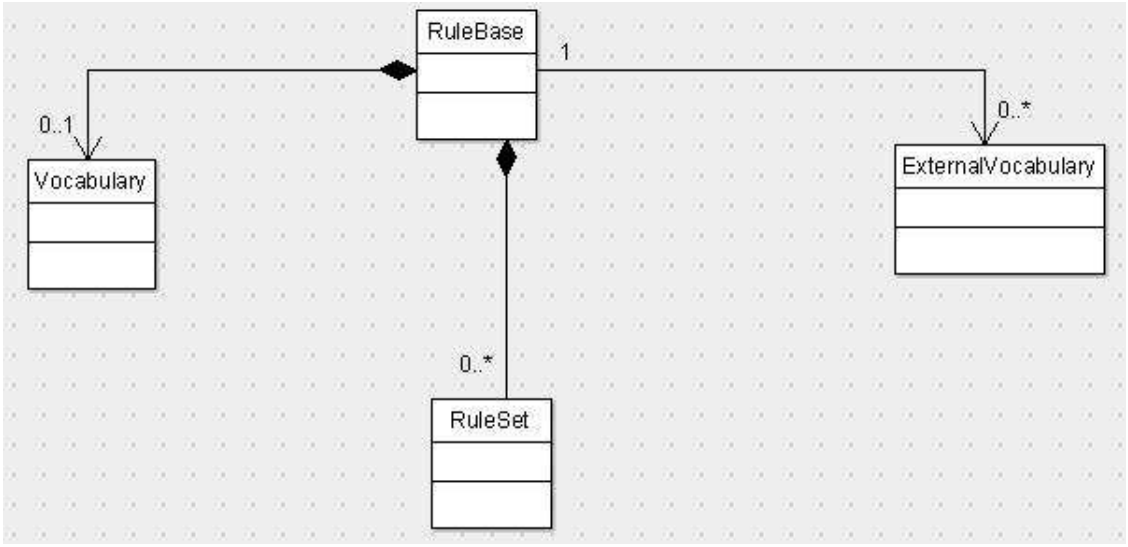
Figure 8 – Transformation of generalization into OCL

After transforming these relations into OCL constraints, the UML model is acceptable by UML2Alloy. Using UML2Alloy with this PML meta-model, it would translate the meta-model into an Alloy model. In this transformation, all the classes were transformed into signatures. All relationships between classes were transformed into relationship between signatures. The cardinality of relations in UML profile would determine the cardinality of relation in Alloy. The OCL constraints representing aggregation/composition are transformed into predicates in Alloy. Table 5 shows the

basic steps of this transformation at a glance. Figure 9 shows various samples of translated PML elements in Alloy. The sample of transformation of a Class and a Relation between classes (in UML representation of PML meta-model) are shown in this figure. As it can be seen, RuleBase (a class in UML diagram) is transformed into a signature definition. Association relations between RuleBase and other classes (in UML) are also transformed to relations in Alloy. For example, the relation between RuleBase and RuleSet is transformed into a "set" relation in Alloy because of the "one to many" cardinality of association relations between these classes. The association relation between RuleBase and Vocabulary is transformed to "lone" in Alloy because of the "one to one or none" relation between these classes. The last row of Figure 9 is the translation of OCL presentation of composition relation as presented in Figure 7.

Table 5 - PML to Alloy transformation

PML	**Alloy**
Composition/Aggregation/Generalization (translated to OCL)	Fact (Predicate)
Class	Signature
Relation	Relation




```
sig RuleBase{
vocabulary:lone Vocabulary,
ruleset:set RuleSet,
externalvocabulary:set ExternalVocabulary}
```

```
pred Vocabulary_voca1[]{
all p: Vocabulary | some a: RuleSet | p in a.rp5
}
pred Vocabulary_voca2[]{
all p: Vocabulary | all a1, a2: RuleSet | (
(p in a1.rp5) && (p in a2.rp5))  =>  ( a1 = a2 ) }
```

Figure 9 – Transformation of part of PML model in Alloy

After having PML meta-model transformed into Alloy, the next step is to integrate concrete policies into the Alloy model. Having concrete policies in Alloy gives us a chance to analyze the concrete policies and handle policy conflicts. The main question to answer here is how to model concrete policies in Alloy. Modelling of

concrete policies and PML meta-model within the one Alloy model creates a framework under which numerous policy conflict detection approaches can be designed and tested.

We have systematically examined feasible approaches in Alloy for modelling concrete policies (Section 5). For each modelling approach and the structure of concrete policies modelled in Alloy, a conflict detection approach is designed and tested. The first approach for modelling concrete policies is to model them in a similar way meta-model was modelled. The first modelling approach is followed by three other approaches for modelling concrete policies, each of which tries to answer the question of how to detect conflicts in Alloy effectively.

In all of the conflict detection approaches, Alloy Assertion has been used. Alloy Assertion is a way to utilize the “Checking” method of Alloy. As stated earlier (in Section 2.7), “Checking” in Alloy is a method to check the model in order to find counterexamples.

Each approach described in the following section is followed by a discussion of this approach. The general outcomes will be summarized in Section 6.

5 Modelling of Policies in Alloy

In this section, different modelling approaches will be introduced. These modelling approaches are part of the whole methodology, as described in Section 4.3. The approaches used for modelling policies will affect the conflict detection methods directly. Thus, a unique conflict detection method is required for each approach. Each approach has its advantages and disadvantages, described in detail in its corresponding section.

We introduce an example that we use in modelling approaches in Section 5.1 and Section 5.2. Section 5.1 will introduce this sample policy-based example and the policies within this system. In Section 5.2, we will introduce motivation example for this sample policy, its UML presentation and minimization used in modelling approaches.

Throughout this section, we are referring to PML meta-model modelled in Alloy as PML meta-model, concrete policies modelled in Alloy as concrete policies. When we are talking about model, we mean PML meta-model and/or concrete policies modelled in Alloy according to the modelling approach used[17].

5.1 Sample Policy Based System

For testing different approaches, we use an example in Health domain. This example assists us to demonstrate the process of detecting conflicts among policies. This example has been inspired by work of Kaviani et. al. [17]. We do not intend to model a complete Health system but only to present a small part of a Health system in order to test presented modelling approaches. Access to resources is determined by policies within

[17] In third modeling approach, the model refers to concrete policies modelled in Alloy while in other approaches model refers to PML meta-model and concrete policies modelled in Alloy.

the system. Each user in this system has a role. “Nurse”, “Patient”, “Employee” and “Doctor” are four different roles within the system.

The scenario, we are considering, is a patient visiting a Health Organization. In this Health Organization it is needed for a Nurse to ask for patient’s information if the patient is new to that Health Organization. A Nurse is not obliged to fill in any information if a patient is in an emergency situation. A Doctor is permitted access to patient’s Health Record. When a patient is visited by a Doctor, Doctor can add information to the patient’s Health Record. In this system, Doctor and Nurse are sub-classed (extended in Alloy terms) from an Employee class (signature). Employee in the system is prohibited from accessing a patient’s Health Record. Patients’ Health Record information is being saved in an entity within the system called ElectronicHealthRecord.

The following policies describe our sample Health Organization’s policies. The first policy (P1) will be expressed in more detail in Section 5.2. Definition of other policies, similar to the P1 policy can be found in Appendix 3 (Section 10).

Policies in our sample policy-based system are as follow:

- P1 (Permission / A+): A Doctor is permitted to access[18] ElectronicHealthRecord.
- P2 (Prohibition / A-): An Employee is prohibited from accessing ElectronicHealthRecord.

[18] This access can be any of update, view or delete actions. Any of these actions are abstracted as an access action in our examples. We stick with the “access” action in the policies in order for our model not to get too complicated.

- P3 (Positive Obligation / O+): A Doctor is obliged to access ElectronicHealthRecord of a patient, after Patient's visit to the Doctor.
- P4 (Positive Obligation / O+): A Nurse is obliged to access ElectronicHealthRecord of a new patient at his/her first visit.
- P5 (Negative Obligation / O-): A Nurse is obliged not to access ElectronicHealthRecord of a patient in an emergency.

5.2 Policy Example

Similar to the Alloy's small scope concept[19], throughout this thesis a similar insight is utilized. The concept used, is to use a small model of the system and try different conflict detection approaches on it.

In this Section, we will focus on one of the policies introduced in Section 5.1 as an example for all other policies. This policy will be presented in PML (through its UML notation) and also in one major policy language notation (XACML).

One of the policies (P1), introduced in Section 5.1, has originated from a policy example presented in [17]. This sample policy (P1) has been altered from the original policy in [17] in order to omit the unnecessary elements. We have tried to keep only the essential elements involved in a policy, as defined in PML. This step has been accomplished in order to have a policy in PML with a minimum number of elements to gain more productivity without losing any semantics or functionality of a policy in PML. Figure 10 is the PML presentation of the original policy in UML notation. The example

[19] Small scope concept in Alloy: if there is a flaw in the system, it can be found by checking small scopes of the system, i.e. considering a small number of instances. For more information please refer to [20]

presented in the [17] had some functionality that can be eliminated without affecting the goal we are pursuing, a policy conflict detection. For example, there is a SendMailAction in the original example that sends an email when Doctor accesses a Health Record. The essential classes for a policy in this example model include Doctor, Permission, AccessElectronicHealthRecord and ElectronicHealthRecord classes. Therefore, we are not going to consider other classes in the modelling approaches.

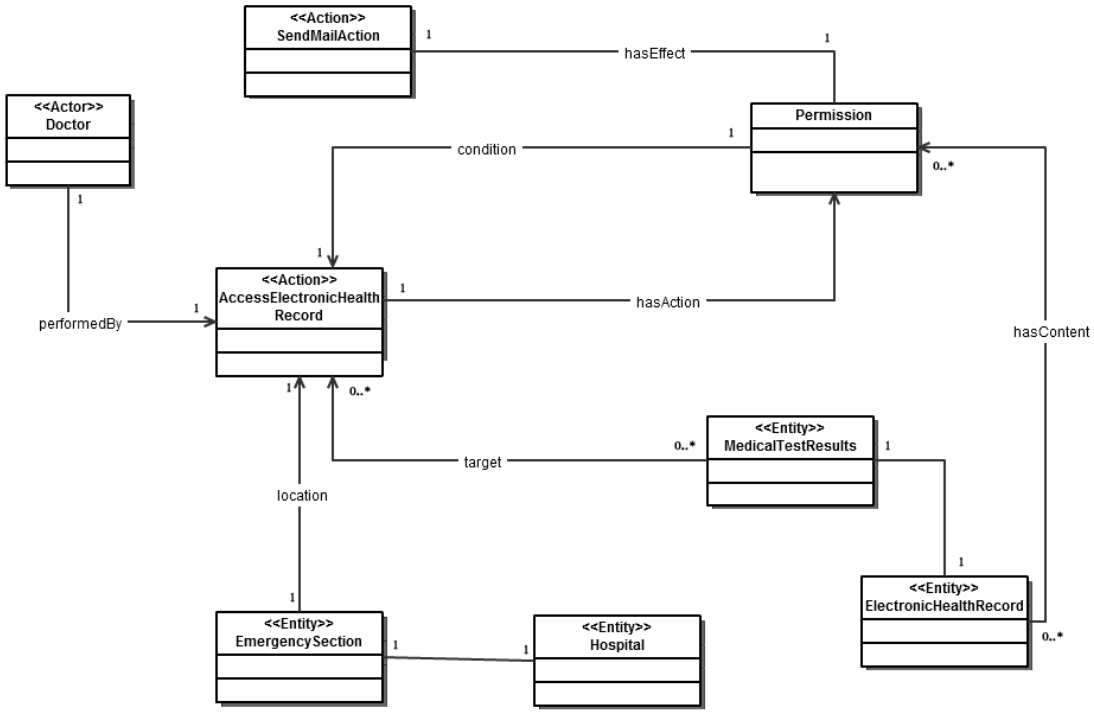


Figure 10 - Original policy example (Kaviani et. al [17]), UML presentation

The sample policy (P1) used in this section presents a policy that grants permission to access a resource to a specific role in the system. In this example,

“ElectronicHealthRecord” is a resource in the system and contains information about “Patients”.

Figure 11 is the presentation of sample policy P1. Comparing Figure 10 and Figure 11 reveals the minimization that took place in order to make this policy smaller. One of Doctor (as Actor), AccessElectronicHealthRecord (as Action), Permission and ElectronicHealthRecord (as Entity) class exist in the sample policy as the essential elements of a Permission policy in PML.

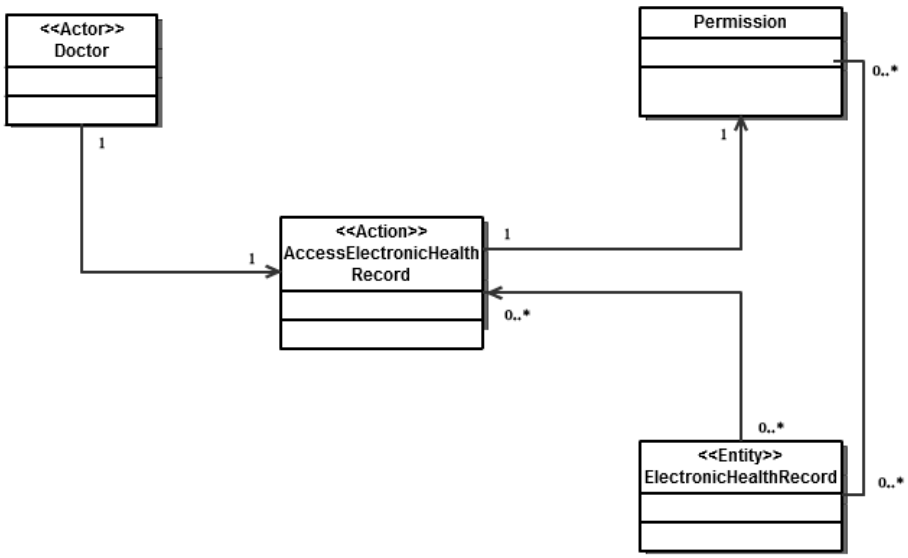


Figure 11 - Sample Policy, used in this section

Figure 12 is the representation of our sample policy in XACML notation, for illustration purposes only. The policy is presented in XACML since it is one of the most used and understood policy languages. Another reason is that Although PML has been presented its interchangeable capability to and from policy languages but the finalized

syntax of PML language (in terms of R2ML notations) is not available yet. Not having the XML presentation of PML to rely on is not affecting our research. Since, in the whole process of conflict detection, the relations between the classes (in UML) or signatures (in Alloy) is solely relied on. Different combinations of relations are the main part of this thesis. These relations can be easily defined having the PML model (via its UML presentation). Using the UML presentation of the policy in PML with the use of UML2Alloy will easily result in the Alloy representation of the concrete policies. Detail steps of the process have been described in Section 4.3.

Policy 1: A Doctor can access Health Record information.

```
<?xml version="1.0" encoding="UTF-8"?>
<Policy xmlns="urn:oasis:names:tc:xacml:2.0:policy:schema:os"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:oasis:names:tc:xacml:2.0:policy:schema:os
http://docs.oasis-open.org/xacml/access_control-xacml-2.0-policy-schema-os.xsd"
xmlns:md="http:www.med.example.com/schemas/record.xsd"
PolicyId="urn:oasis:names:tc:xacml:2.0:example:policyid:4"
RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining1613
algorithm:deny-overrides">
<PolicyDefaults>
    <XPathVersion>http://www.w3.org/TR/1999/Rec-xpath-
    </XPathVersion>
</PolicyDefaults>
<Target/>
<Rule RuleId="urn:oasis:names:tc:xacml:2.0:example:ruleid:4" Effect="Permit">
<Description>
A doctor shall be permitted to read or write medical elements of a patient record in the http://www.med.example.com/records.xsd namespace.
</Description>
<Target>
    <Subjects>
        <Subject>
        <SubjectMatch
        MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
            doctor
            </AttributeValue>
            <SubjectAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:2.0:example:attribute:role"
            DataType="http://www.w3.org/2001/XMLSchema#string"/>
        </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">
                urn:example:med:schemas:record
                </AttributeValue>
                <ResourceAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:resource:target-namespace"
                DataType="http://www.w3.org/2001/XMLSchema#string"/>
            </ResourceMatch>
            <ResourceMatch
                MatchId="urn:oasis:names:tc:xacml:1.0:function:xpath-node-match">
                <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
                /md:record/md:medical
                </AttributeValue>
                <ResourceAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:resource:xpath"
                DataType="http://www.w3.org/2001/XMLSchema#string"/>
            </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
                AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"
                DataType="http://www.w3.org/2001/XMLSchema#string"/>
            </ActionMatch>
        </Action>
        <Action>
                <ActionMatch
                MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
                <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
                write
                </AttributeValue>
                <ActionAttributeDesignator
                AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"
                DataType="http://www.w3.org/2001/XMLSchema#string"/>
                </ActionMatch>
        </Action>
    </Actions>
</Target>
</Rule>
</Policy>
```

Figure 12 **– Sample Policy, presented in XACML**

5.3 Modelling Concrete Policies

In the first and second modelling approaches, concrete policies are modelled in the same way the PML meta-model was modelled in Section 4.3. The modelling was done through the UML profile of PML and by using UML2Alloy tool, considering some special alterations in the model (i.e. using OCL as described in Section 4.3). After

transformation of the concrete policy into Alloy, the connection to the PML meta-model is created to check the syntax of concrete policies against the PML meta-model. Extension relations (in Alloy) can be used to connect concrete policy to meta-model. Therefore, each concrete policy would be considered a child of the Policy signature in Alloy. The extension relations connect each signature (in Alloy model) with its corresponding signature in the PML meta-model. Figure 13 presents the sample concrete policy in Alloy. In this example, Permission1 is the concrete policy while Permission is the permission class in PML meta-model. The extension relation between these classes was created using the "extends" keyword. For instance, ElectronicHealthRecord is a concrete class in the system while Entity is the class in PML meta-model and they are related using "extends" keyword in Alloy.

```
sig Permission1 extends Permission {
accesselectronichealthrecord:one AccessElectronicHealthRecord}

sig Doctor extends Actor {
accesselectronichealthrecord:one AccessElectronicHealthRecord}

sig AccessElectronicHealthRecord extends Action {
permission:one Permission}

sig ElectronicHealthRecord extends Entity {
accesselectronichealthrecord:set AccessElectronicHealthRecord,
permission:set Permission}
```

Figure 13 - Alloy Presentation of Sample Concrete Policy

Alloy Analyzer, the tool that helps to investigate Alloy models, utilizes the extension relation in Alloy. The extension relationships between concrete policies and

PML meta-model will be used in the process of checking for policy conflicts, which will be described later on in different modelling approaches.

We have used the extension relation in Alloy model to connect concrete policy signature and policy signature. Having extension relation used in Alloy, Alloy Analyzer is able to “Check” assertions written for parent class (policy signature) as well as child classes (concrete policy signature). As an example in the original policy (Figure 10), let us consider “Hospital” signature. Hospital is related to EmergencySection and as it can be seen, Hospital signature is extending the EmergencySection Entity. This extension makes EmergencySection a Hospital. Thus, Alloy Analyzer will check EmergencySection for any assertions written for Entity of Hospital as well.

```
fact Asso_Action_action_entity_Entity { Action <: entity in  ( Action) set->set (
Entity) && Entity <: action in  ( Entity) set->set ( Action) }
```

Figure 14 - Sample Fact (in Alloy)

UML2Alloy was used to transform PML meta-model into Alloy. Thereon, looking into the transformed PML meta-model in Alloy, we will find some facts generated. The generated Facts are fundamentally about relationship between classes, cardinalities, hierarchies etc. As an example, a fact is presented in Figure 14. This fact forces the cardinality of relation between Action and Entity to be multi to multi, meaning that both signatures (classes in UML) can have zero to infinity number of relations to other signature (other class in UML). Similar facts are also being checked for Hospital as well as Entity.

5.4 First Modelling Approach

In the first modelling approach, we have used concrete policies and PML meta-model in the Alloy model. PML meta-model was modelled as described in Section 4.3 and concrete policies were modelled using the method described in section 5.3. Then, we used Alloy Analyzer to Run (c.f. Section 11.4) the model (i.e. PML meta-model and concrete policies) and analyze for its consistency. As there is no assertion used in this step, analysis does not intend to check for policy conflicts. As it turns out, there was a problem identified as a result of analysis of the model, which holds us back from continuing this approach.

5.4.1 Outcome

The problem with this modelling approach was the generation of relations that might lead to generation of unwanted policies. The origin of this problem was related to the expansion of various signatures in the Alloy model. Using Alloy Analyzer on the model, it tries to generate as many possible relations in a model as it can. Hence, it might generate unwanted policies.

Alloy Analyzer does not always generate same instance when one uses it to analyze an Alloy model. Based on the Alloy model and based on the definition of signatures and relations in the model, Alloy Analyzer generates a portion of a model as instances of the model. Not each time the generated instance of the model is the same. However, the generated instance is always within the definitions of signatures and relations the Alloy model. Instance generation is one of features of Alloy, which benefits

researchers to investigate Alloy models from different points of view by generating different instances from their model.

Considering this feature of Alloy on our model, it can result in generation of possible unwanted policies, generated from the policy signatures in the PML meta-model. Generation of additional policies is not desirable. As an example, let us consider a simple case to present how these unwanted policies can alter behaviour of a system and our policy conflict detection method. It is likely that one of these generated policies will be exactly a policy conflicting with one of the policies within the system.

Because of this behaviour of the Alloy Analyzer, there is no point to design a conflict detection method. Even if a conflict detection method in Alloy were available, it would not be much of a help to detect conflicts in the system; since conflicts could be detected among policies that, some of them, are generated by Alloy Analyzer.

Let us clarify it more through an example. Let us consider a sample policy permitting Doctor to access Health records (Permission policy). In contrast with this policy, a policy can be generated from the Prohibition signature in the PML meta-model that prohibits Doctor to access the Health Record. This possible generated policy can be seen in Figure 15. This generated policy will end up in a conflict with one of the concrete policies defined within the system, as one is permitting and the other one is prohibiting access to one specific resource. Therefore, any policy conflict detection approach will fail as unwanted policies are being generated throughout the analysis of the model. In order to be able to detect conflict between policies in Alloy, first we need to resolve this problem of unwanted generated policies.

```
sig Prohibtion1 extends Prohibition{
accesselectronichealthrecord:one AccessElectronicHealthRecord}

sig Doctor extends Actor{
accesselectronichealthrecord:one AccessElectronicHealthRecord}

sig AccessElectronicHealthRecord extends Action{
permission:one Permission}

sig ElectronicHealthRecord extends Entity{
accesselectronichealthrecord:set AccessElectronicHealthRecord,
prohibition:set Prohibition}
```

Figure 15 - Sample generated policy

In brief, the following advantages and drawbacks of this approach can be named.

Advantages:

- A systematic way to model concrete policies along with its meta-model in Alloy has been introduced.

Disadvantages:

- Generation of unwanted concrete policies can result in indefinite number of conflicts between policies.

5.5 Second Modelling Approach

In the second modelling approach, we attempt to limit the expansion capability of Alloy Analyzer. The limitation was set up to affect generation of instances in the Alloy model (i.e. signature elements). As stated in the first modelling approach (Section 5.4),

generation of new policies is undesirable. Considering domain manager's point of view, generating new policies from the PML meta-model signatures not only is undesirable but also may change the behaviour of the system as well. The only policies needed to be analyzed are the concrete policies defined within the system.

Concrete policies are modelled in a similar way that we have modelled PML meta-model from its UML profile into the Alloy model. "Policy" signature (in Alloy model) is parent of all concrete policies modelled in the Alloy model. In other terms, all concrete policies are children of Policy signature.

Alloy Analyzer is capable of limiting the number of instances it generates (for a specified signature) as it expands the model. Using this capability, the expansion of policy class (meta-model) and its children (concrete policies) was limited.

The limitation on expansion of the model can be achieved by using the "for but" keyword after the "run" command in Alloy. By specifying the number of desired instances of signatures, one can tell Alloy to generate only specific number of instances from the model. Figure 16 shows the difference this keyword makes. The top code is the code used to generate Alloy model without any limitations (e.g. first modelling approach) while the code restricting generation of signatures in Alloy model is presented in the bottom of the figure (e.g. second modelling approach). The generation is limited to one signature for the Permission1, Doctor, Nurse, ElectronicHealthRecord, and AccessElectronicHealthRecord signatures. Figure 17 visually shows the difference this limitation makes. On the top, a sample run of part of Alloy model without any limitations applied, can be seen while on the bottom of Figure 17, the model can be seen with the limitation applied.

run {} for 4
run {} for 4 but 4 int 1 Permission1, 1 Doctor, 1 Nurse, 1 ElectronicHealthRecord, 1 AccessElectronicHealthRecord

Figure 16 - Limiting Generation of signatures (in Alloy)

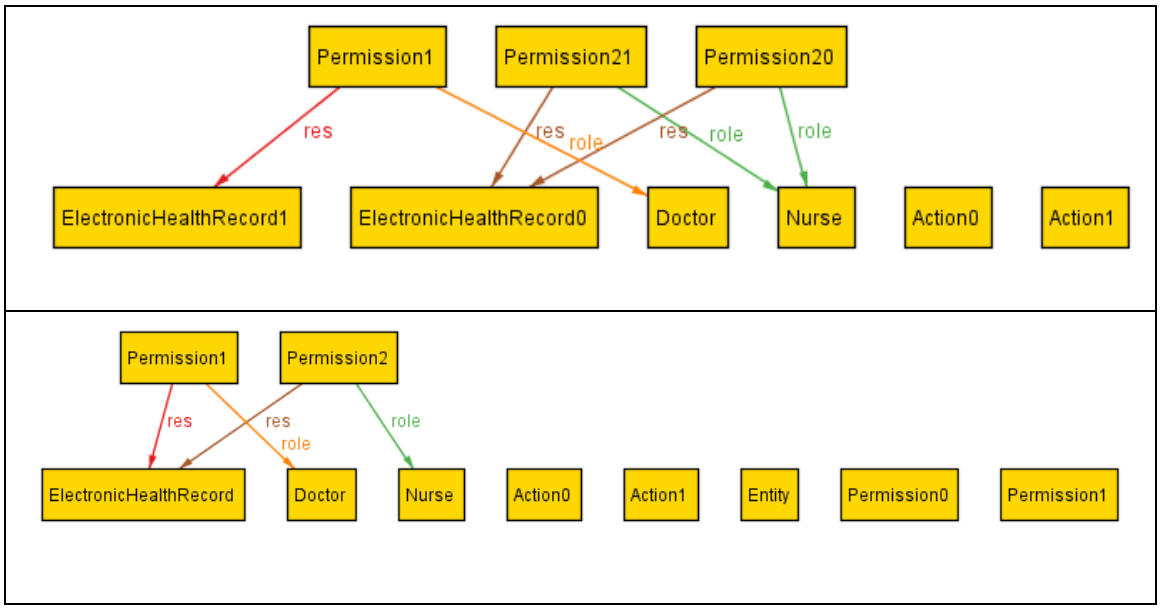


Figure 17 - Alloy model - Top without any limitation - Bottom limitation applied

For the purpose of testing capability of this approach to detect policy conflicts, another policy along with the example policy has been used. The original example policy was to grant/deny access to Doctor over HealthRecord. The other policy used here will be similar to this policy but only with one difference. The Actor of the policy will be Nurse in this case. Considering the top figure in Figure 17, Permission1 is the policy permitting/denying a Doctor to access ElectronicHealthRecord while Permission2 is the policy permitting/denying a Nurse to access ElectronicHealthRecord. As it can be seen in

lower part of Figure 17, this limitation restricted generation of new policies and only Permission1 and Permission2 policies were generated. However, not using the limitation in Alloy model would result in the simulation shown in the upper part of Figure 17. Permission21 and Permission20[20] are the same policy generated from Permission2. The last two policies were generated by Alloy (in the top) are duplicate policies in our Alloy model while there was only one instance of them in the system we wanted to model. However, the only generated policies in the bottom part are the Permission1 and Permission2 policies as required.

5.5.1 Conflict Detection

In order to implement policy conflict detection approach, we used assertions in Alloy. Assertion is a mechanism in Alloy to find counterexample(s). As mentioned in Section 2.7, Alloy investigates models in two different ways and Assertion is to "Check" the model in Alloy.

Conflicts among policies may take place because of different reasons and in different situations. Complete discussion over occurrence of conflicts among policies can be found in Section 2.5. However, as the first step of detecting conflicts amongst policies, we selected a Prohibition/Permission conflict scenario between two policies. We simply check if a role (here Doctor) has prohibition and permission access to a resource (i.e. HealthRecord) at the same time (modality conflicts). Being granted permission over a resource while being prohibited over that specific resource is a type of conflict among various conflict types described earlier in Section 2.5.

[20] Alloy adds a numeric value to the end of signatures' names while generating instances from the model.

Assertions were used at the level of the PML meta-model rather than concrete policies, namely Policy signature. Using assertion at the level of Policy signature was inspired by the structure of the model, in which both PML meta-model and concrete policies are present. Each element of concrete policy was connected to its specific element in PML meta-model. Having assertion at the level of Policy signature, which is a parent of all other concrete policies, will propagate “checking” for all of the concrete policies (policy signature in Alloy). Alloy would propagate the assertion to the children of Policy signature that is concrete policies. This is a desired situation in a conflict detection approach, to write a constraint for a class of policies and being able to check other derived signatures (concrete policies) against it.

The assertion used for conflict detection in this modelling approach checks a simple fact. The fact is for any role within the system, both Permission and Prohibition policies on a same resource and the same action should not be present at the same time. Figure 18 shows the assertion used to detect the conflict of having Permission and Prohibition policy for the same role. In this conflict detection approach, assertion only checks for conflicts based on the role variable. Other variables in a policy (i.e. action and resource) are not considered here for detecting conflicts.

In general, a conflict between two policies, where one is Prohibition and the other one is Permission, needs to be checked for the role, action and resource. If for a specific role, a specific action has both Permission and Prohibition policies present in the system for the specific resource, then there is a conflict between these two policies. However, as the first step we only check for conflicts for roles, which immediately lead into a problem with this approach. Considering adding more variables to the equation will not resolve it

but makes the matter more complicated. The analysis of the Alloy model using the assertion shown in Figure 18 resulted in the system error and no conflicts have been detected. The detail information on the outcome and the problem is given in the next Section.

```
assert nop{ all p: Prohibition ,  q: Permission | q->role = p->role }
```

Figure 18 - Conflict Detection (Assertion in Alloy)

5.5.2 Outcome

The issue with the "Second modelling approach" was based on using of assertions. Looking for counterexamples in a model, one should use assertions in Alloy. However, assertions used in this modelling approach resulted in an error. We should discuss what could be done in order to solve this issue.

In this modelling approach, assertions were used at the level of Policy signature and not at the level of concrete policies. Using assertions at the level of Policy signature, forces Alloy Analyzer into a situation that could not resolve a relation among different available relations. As expected, what happens in this situation is that Alloy Analyzer tries to check the assertion both at the level of Policy signature and at the level of concrete policies as well**.** In this case, when Alloy Analyzer encounters a relation, which exists in the policy signature (i.e. Permission) and also in the concrete policy signature (i.e. Permission1), it cannot make a distinction between them. This is because concrete policy signatures are policy signatures in the Alloy model. This problem is rooted in having the same name for the relation in the policy and concrete policy signatures. Using

the same name is inevitable and cannot be avoided. This situation originates in extension of concrete policies from the PML meta-model's Policy signature, the extension that makes the connection between policy and concrete policy possible. One might say simple change of relation name would solve the problem. However, one should consider the generation process and the assertion asserted at the level of Policy signature.

sig Permission{
role:one Actor,
}
	sig Permission1 extends Permission{
	role:one Doctor,
	}

Figure 19 – Permission Signature (Policy) and Concrete Policy (Permission1)

As an example, a part of concrete policy and part of policy signature is presented in Figure 19. As it can be observed, both policy and concrete policy signatures have a relation called role**.** Alloy Analyzer tries to run the assertion for the concrete policy as well as the policy signature. When Alloy Analyzer attempts to check the q->role (role of the user), the problem rose. Alloy Analyzer could not decide which "role" we are referring to thus it throws an error[21]**.** The assertion, which is written for the policy signature, will not be able to resolve the reference. The reason is that both parent and child class (concrete policy and abstract policy signature) are considered as policy signature and both have a relation called role. As a summary of advantages and drawbacks of this approach, following can be named:

Advantages:

[21] Alloy is not supporting overwriting of relations. Overwriting is a concept that has been utilized in Object Oriented paradigm.

- Compared to the previous modelling approach, a general conflict detection approach was introduced for the first time here, although not successful.

Drawbacks:

- Alloy Analyzer got into problem with this model. The reason was usage of assertion at the level of policy class, plus having relations with the same name in policy and concrete policies classes.

5.6 Third Modelling Approach

The only way to investigate an Alloy model for counterexamples is to use assertions. Third modelling approach is designed in order to solve the problem occurred in the previous modelling approach. In the previous modelling approach, we attempted to detect policy conflicts by using assertions at the level of policy signature, which was the main source of the problem. In the third modelling approach, we attempt to alter the level of assertion with respect to the inheritance hierarchy of PML meta-model. In this modelling approach, we attempted to use assertion not at the level of policy signature but at the level of concrete policies.

Previous modelling approaches (c.f. Sections 5.4 and 5.5) use the assertion at the level of policy signature, so each time a concrete policy inserted into Alloy model, it automatically will be checked against conflict detection as each concrete policy is a policy itself. Therefore, Alloy Analyzer treats those concrete policies as policies. In this modelling approach, this is not the case and each concrete policy would be treated on its own and not through its parent's signature (i.e. policy signature). In this modelling

approach, there is no extension from parent signature (Policy signature) available for each child signature (Concrete Policy).

Having no extensions in modelling of concrete policies, will let us to alter modelling of concrete policies in more alternative ways. Having extensions in Alloy model, we are forced to follow the exact structure of a policy defined in PML meta-model in the Alloy model, since each signature is needed to be related to an element of meta-model. Otherwise, there will be no point in having PML meta-model and concrete policies in an Alloy model.

The modelled policies (i.e. P1 to P5) are shown in Figure 20. Since there is no general assertion (as in Second modelling approach) available for each concrete policy, we need to write assertions in this modelling approach. We will discuss automatic generation of these assertions in Section 5.8.

```
// Policy P1: Doctor is permitted to access the Resource
sig Permission1 {
role:one Doctor,
res: one ElectronicHealthRecord,
act: one Action}

// Policy P2: Employee is prohibited to access the resource
sig Prohibition1 {
role:one Employee,
res: one ElectronicHealthRecord,
act: one Action}

// Policy P3: Doctor is obliged to access the resource (after visiting a patient)
sig ObligationPlus1 {
role:one Doctor,
res: one ElectronicHealthRecord,
act: one Action}

// Policy P4: Nurse is obliged to access the resource (after first visit of a new patient)
sig ObligationPlus2 {
role:one Nurse,
res: one ElectronicHealthRecord,
act: one Action}

// Policy P5: Nurse is obliged not to access the resource (if it is an emergency situation)
sig ObligationMinus1 {
role:one Nurse,
res: one ElectronicHealthRecord,
act: one Action}
```

Figure 20 - Modelled Policies

The assertions in this modelling approach are identical to the assertions in the previous approach (Section 5.5) but with only one difference that they are being applied at the level of concrete policies and not at the level of policy signatures. Figure 21 shows the difference between these assertions. The top assertion is the assertion used in the

previous approach. The top assertion was used at the level of policy signature. Assertion at the bottom of Figure 21 presents the assertion used in the current approach. As it can be seen, the only difference from the previous approach is that the assertion is written for concrete policies “Prohibition1” and all “Permission1” while in the previous approach it was written for PML meta-model signatures, i.e. “Permission” and “Prohibition”.

```
assert nop{ all p: Prohibition ,  q: Permission | q->role = p->role }
assert nop{ all p: Prohibition1 ,  q: Permission1 | q->role = p->role }
```

Figure 21 – Assertion at the level of Policy (Top) and at the level of Concrete Policy (Bottom)

To detect different types of conflicts (as stated in Section 2.5.1), we need to check three different types of conflicts: A+/A- , O+/O-, O+/A-. In the following sections, we will describe steps taken for detecting conflicts in each conflict type.

5.6.1 Permission/Prohibition (A+/A-)

The assertion presented in bottom part of Figure 21 is an assertion that checks for a Permission/Prohibition type of conflict between two concrete policies. We use the assertion for the modelled policies in Alloy, as presented in Figure 20. Policy P1 grants access to a Doctor to access ElectronicHealthRecord while P2 prohibits access of any employee in the system to ElectronicHealthRecord. Doctor, being an Employee by definition, is granted and denied access to ElectronicHealthRecord at the same time. The assertion presented in the bottom part of Figure 21 tries to find this conflict by checking if a same role has both permission/prohibition policies associated with it, which permit/deny access to a resource at the same time.

The top part of Figure 22 demonstrates the result of Alloy Checking operation with this assertion. As it can be seen, the assertion succeeded in finding the counterexample we were looking for. The counterexample shows that there was a conflict and it was detected. The bottom part of Figure 22 shows one such counterexample generated by Alloy Analyzer. As it can be seen, a Permission policy (Permission1) is relating Doctor1 to Action while at the same time a Prohibition policy (Prohibition1) relates them.

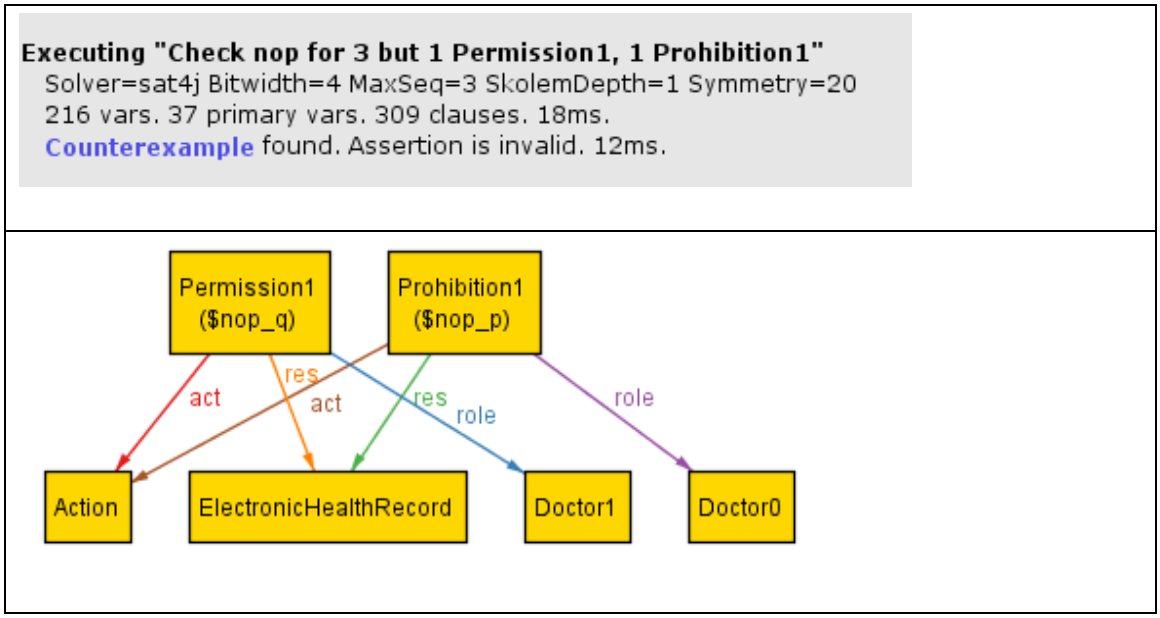


Figure 22 -Assertion successful in finding counterexample

5.6.2 Positive Obligation / Negative Obligation (O+/O-)

The assertion presented in Figure 23 is an assertion that checks for a Positive Obligation / Negative Obligation type of conflict between policies.

```
assert nop3{ all p: ObligationMinus1, o: ObligationPlus2 | o.role = p.role }
```

Figure 23 - Assertion for O+/O-

As stated earlier in the sample policy based example, policy P4 obliges a Nurse to access ElectronicHealthRecord upon visit of a new patient and policy P5 obliges Nurse not to access ElectronicHealthRecord in an emergency situation. In a case in which a new patient visits in emergency situation, these policies will end up in conflict. The assertion presented in Figure 23 tries to find this conflict by checking if a same role has both Positive Obligation/Negative Obligation policies associated with it on a same action to a resource at the same time.

The top part of Figure 24 shows the result of Checking the assertion. As it can be seen, the assertion succeeded in finding the counterexample we were looking for. The counterexample demonstrates that there was a conflict and it was detected. The bottom part of Figure 24 shows one counterexample generated by Alloy Analyzer. As it can be seen, a Negative Obligation policy (ObligationMinus1) is relating Nurse1 to Action while at the same time a Positive Obligation policy (ObligationPlus2) relates them.

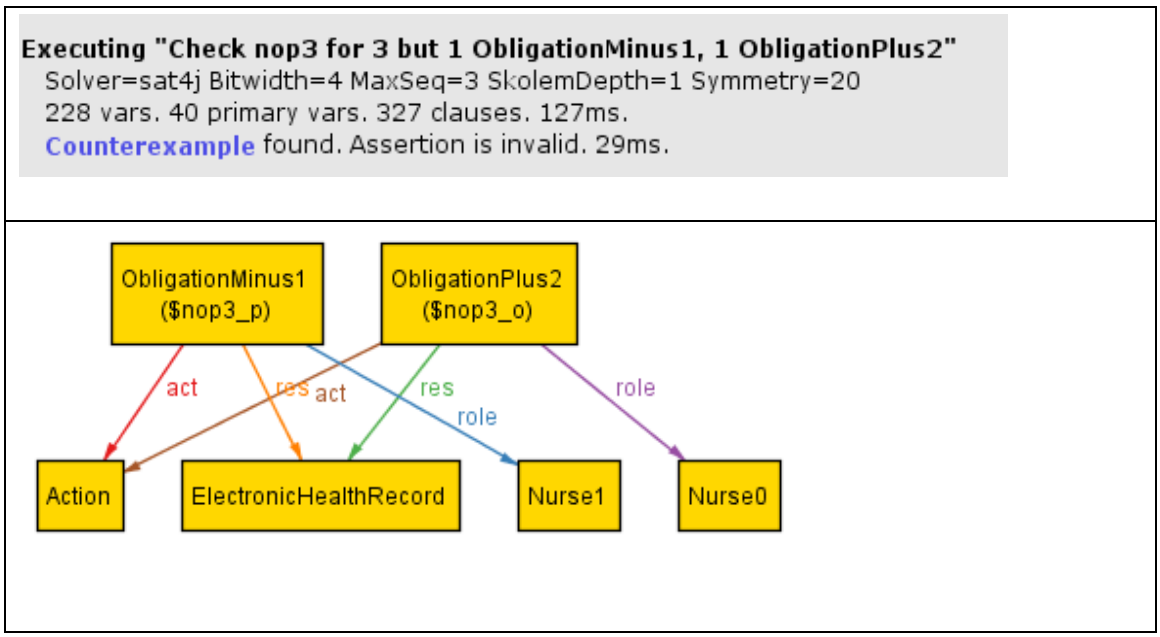


Figure 24 - Assertion successful in finding counterexample

5.6.3 Positive Obligation / Prohibition (O+/A-)

The assertion presented in Figure 23 is an assertion that checks for a Positive Obligation / Prohibition type of conflict between policies.

```
assert nop2{ all p: Prohibition1 ,  o: ObligationPlus1 | o.role = p.role }
```

Figure 25 - Assertion for O+/A-

In our example, Policy P3 obliged a Doctor to access ElectronicHealthRecord when a Doctor visits a patient while P1 Prohibited access to ElectronicHealthRecord to an Employee. The assertion presented in Figure 25 tries to find this conflict by checking if a same role has Positive Obligation/prohibition policies on an action to a resource at the same time.

The top part of Figure 26 demonstrates the result of Checking the assertion. As it can be seen, the assertion succeeded in finding the counterexample we were looking for. The counterexample shows that there was a conflict and it was detected. The bottom part of Figure 26 shows one counterexample generated by Alloy Analyzer. As it can be seen, a Positive Obligation policy (i.e. ObligationPlus1) is relating Doctor0 to Action while at the same time a Prohibition policy (i.e. Prohibition1) relates them.

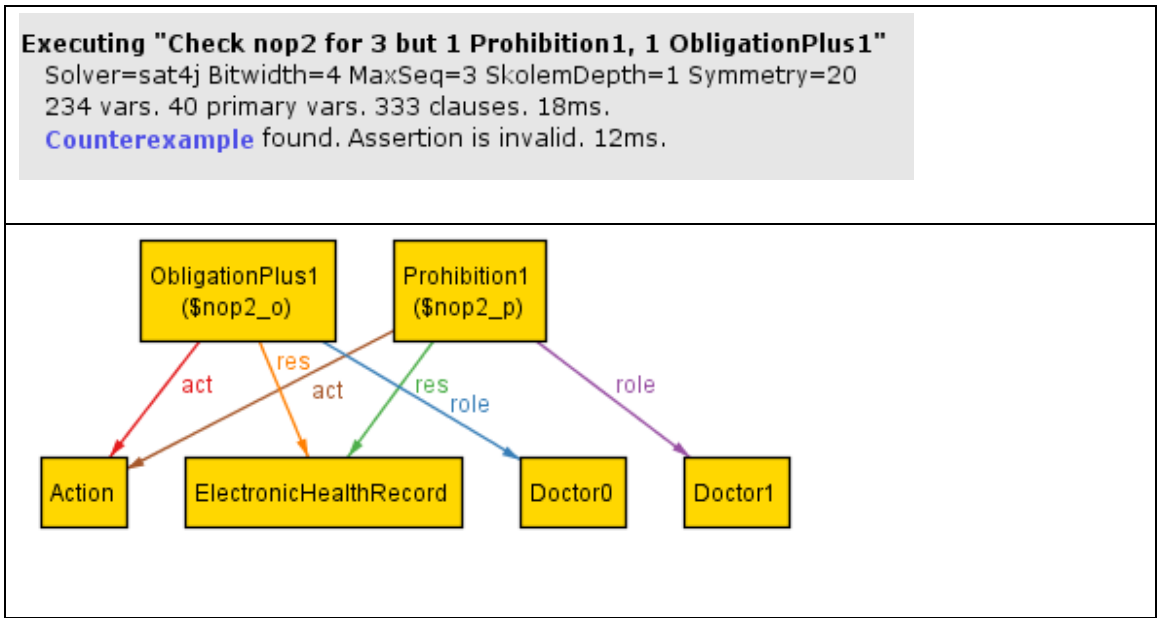


Figure 26 - Assertion successful in finding counterexample

5.6.4 Outcome

In this approach, we have shown the feasibility of detecting conflicts among policies using Alloy. In detected conflicts (counterexamples) that have been visually presented (Figure 22, Figure 24 and Figure 26), more than one Doctor (i.e. Doctor0 and Doctor1) can be seen in the figure. However, in our model we do not have more than one

Doctor (or Nurse). These generated Doctors (Doctor1 and Doctor0) should be considered as Doctor signature in the Alloy model. These signatures are instances of the Doctor signature generated by Alloy Analyzer. Therefore, when the assertion found instances of Doctor0 and Doctor1 in the counterexample within the Alloy model, it actually found a policy conlflict for Doctor role in the system.

As it can be seen in Figure 22, the result of the conflict detection was successful, i.e. a counterexample was found using this approach. However, a drawback for this approach is losing the connection to the PML meta-model. As mentioned earlier and as shown in Figure 20**,** there is no connection to the PML meta-model. Using this approach, PML meta-model has not been used at all. Not extending signatures from the meta-model is one of biggest drawbacks here. Not having this option, one cannot check if the policy is syntactically correct or not.

As a summary of advantages and drawbacks of this approach, following can be named:

Advantages:

- This approach has shown the proper modelling for concrete policies in order to detect conflicts using Alloy.

Drawbacks:

- As a side drawback, in this approach, connection to the PML meta-model was lost. This loss prevented us from syntactically checking the correctness of concrete policies.

5.7 Fourth Modelling Approach

Although in previous modelling approach, we have showed a proper way to model policies in Alloy, but we want to try all possible ways to model concrete policies in Alloy. In this approach, modelling of policies will be experimented by mainly using relations to model concrete policies in Alloy.

In previous approaches, each policy was modelled as a signature in the Alloy model while in this modelling approach, policies will be modelled using relations. All policies are modelled within one signature.

In this modelling approach, we define a policy signature (as shown in Figure 27). This policy signature contains relations that define concrete policies in the system. In this approach, each policy is treated as an instance of a policy signature. The different values of this signature (i.e. policy signature) define concrete policies. Figure 27 shows the structure used to model policies with this approach in Alloy. The policy signature has relations including PolicyType, Actor, Action and Entity. Actor, Action and Entity are the signature models for Actor, Action and Resource in Policies. PolicyType is expressing different types of the policy including Permission, Prohibition, Positive Obligation and Negative Obligation. This signature with the relations will model all the concrete policies in the system.

```
sig PolicyType{ }
sig Permission extends PolicyType { }
sig Prohibition extends PolicyType{ }

sig policy{
p:PolicyType->Actor->Action->Entity
}
```

Figure 27 - Policy Model (based on relation in Alloy)

Figure 28 shows a sample policy modelled in this approach. As it can be seen, a policy is inserted into model as a fact. The combination of different facts, if inserted into the system, can create new policies within the system in this modelling approach. In this example, a Permission policy for a Doctor to Access ElectronicHealthRecord is defined.

```
fact{
Permission->Doctor->Access->ElectronicHealthRecord in policy.p
}
```

Figure 28 - **Sample policy (modelled using fourth modelling approach)**

5.7.1 Outcome

Having concrete policies modelled as a single signature and relations within it, we have faced one major issue that cannot be resolved. The problem we have faced was unwanted expansion of the model. The problem of expansion of model, resulting in generating unwanted policies, is similar to what we have encountered in the first modelling approach. The problem with previous modelling approaches (Section 5.4, 5.5 and 5.6) has been resolved using a limitation at the level of signatures. This limitation allowed us to restrict number of instances of signatures generated in the Alloy model.

However, since in this approach we are modelling policies using one signature with relations distinguishing between policies, no similar action can be used as there is no way of restricting Alloy Analyzer to generate relations.

As a summary of advantages and drawbacks of this approach, following can be named:

Advantages:

- In this approach, a different way of modelling policies in Alloy was investigated. We used the relations to model different policy types.
- All policies were modelled in one signature. It might be useful for analyzing a set of policies, since all policies are modelled in a unique signature.

Drawbacks:

- The major drawback was incapability of us to utilize Alloy Analyzer to limit the number of generated instances of policies, while modelling policies using relations.

5.8 Generation of Assertions

In the third modelling approach, different assertions for our policy-based example system have been presented. These assertions have helped us to detect conflicts among policies. In this Section, we discuss assertions needed to be asserted into a model to detect conflicts. We also present an automatic way to generate these assertions.

According to Section 2.5.1, three different policy conflict types can be thought of in policy-based systems. These conflicts are A+/A-, O+/A-, O+/O-. In order to detect any possible conflict within a system, one might check for any possible combination of these

types of policies within an Alloy model. In this Section, we present how to perform this task by adding required assertions. Accomplishing this task, we consider policies within the system have Actor, Policy Type, Action and Resource elements present in their definition. We refer to concrete policies within the system as Pol-1, Pol-2 ... Pol-n. Structure of a policy is presented in Table 6. The last column of Table 6 presents a sample policy called POL-1.

Table 6 - Policy Structure

<policy> ::= <policy-name><actor><policyType><action><resource> <policy-name> ::= string <actor> ::= string <policyType> ::= Permission | Prohibition | Positive Obligation | Negative Obligation <resource> ::= string <action> ::= string				
POL-1	Doctor	Is Permitted	To Access	ElectronicHealthRecord

We assume that the policies are categorized based on their policy types within the system into four different categories: Permission (A+), Prohibition (A-), Positive Obligation (O+), And Negative Obligation (O-). Based on these policy types, we introduce the assertions needed to be generated within the Alloy model. For Permission and Negative Obligation there is a need to add only one assertion in the model, since they only can be involved in one policy conflict type (i.e. A+/A- for Permission and O+/O- for Negative Obligation). For Prohibition and Positive Obligation we need to have two assertions as they can be involved in two type of policy conflicts (i.e. A+/A- and O+/A- for Prohibition and O+/A- and O+/O- for Positive Obligation).

Figure 29, Figure 30, Figure 31 and Figure 32 presents the assertion needed for Permission, Prohibition, Positive Obligation and Negative Obligation policies within the system accordingly.

Generate the following assertion for the Permission policy (Pol-j) and all Prohibition policies (Pol-k) within the system.

- assert Pol-j-Pol-k { all p1: Pol-j, p2: Pol-k | p1.role = p2.role }

Figure 29 - Assertion for Permission Policy Type

Generate the following assertion for the Prohibition policy (Pol-j) all Permission policies (Pol-k), and all Positive Obligation (Pol-l) within the system.

- assert Pol-j-Pol-k { all p1: Pol-j, p2: Pol-k | p1.role = p2.role }
- assert Pol-j-Pol-l { all p1: Pol-j, p2: Pol-l | p1.role = p2.role }

Figure 30 - Assertion for Prohibition Policy Type

Generate the following assertion for the Positive Obligation (Pol-j) policy and all Negative Obligation policies (Pol-k) within the system.

- assert Pol-j-Pol-k { all p1: Pol-j, p2: Pol-k | p1.role = p2.role }
- assert Pol-j-Pol-k { all p1: Pol-j, p2: Pol-k | p1.role = p2.role }

Figure 31 - Assertion for Positive Obligation Policy Type

Generate the following assertion for the newly scanned Negative Obligation policy (Pol-j) and any previously scanned Positive Obligation policies (Pol-k).

- assert Pol-j-Pol-k { all p1: Pol-j, p2: Pol-k | p1.role = p2.role }

Figure 32 - Assertion for Negative Obligation Policy Type

Assertions presented in the above figures are the required assertion for each type of policies in order to detect policy conflicts that involve the specific policy type. Generating all the assertions presented in above figures, one will find duplicate assertions are being generated for checking same policy conflict. For example, a Permission / Prohibition conflict needs only one assertion. However, it can be generated by using following steps presented in Figure 29 or Figure 30.

Covering all types of policy conflicts, we propose the following approach.

1. If policies are not sorted, sort them based on policy types.
2. For any Positive Obligation policy within the system, generate the assertions presented in Figure 31 and add these assertions to the Alloy model.
3. For any Permission policy within the system, generate the assertions presented in Figure 29 and add these assertions to the Alloy model.
4. Use the assertion above to find any possible conflicts (counterexamples) in Alloy model.

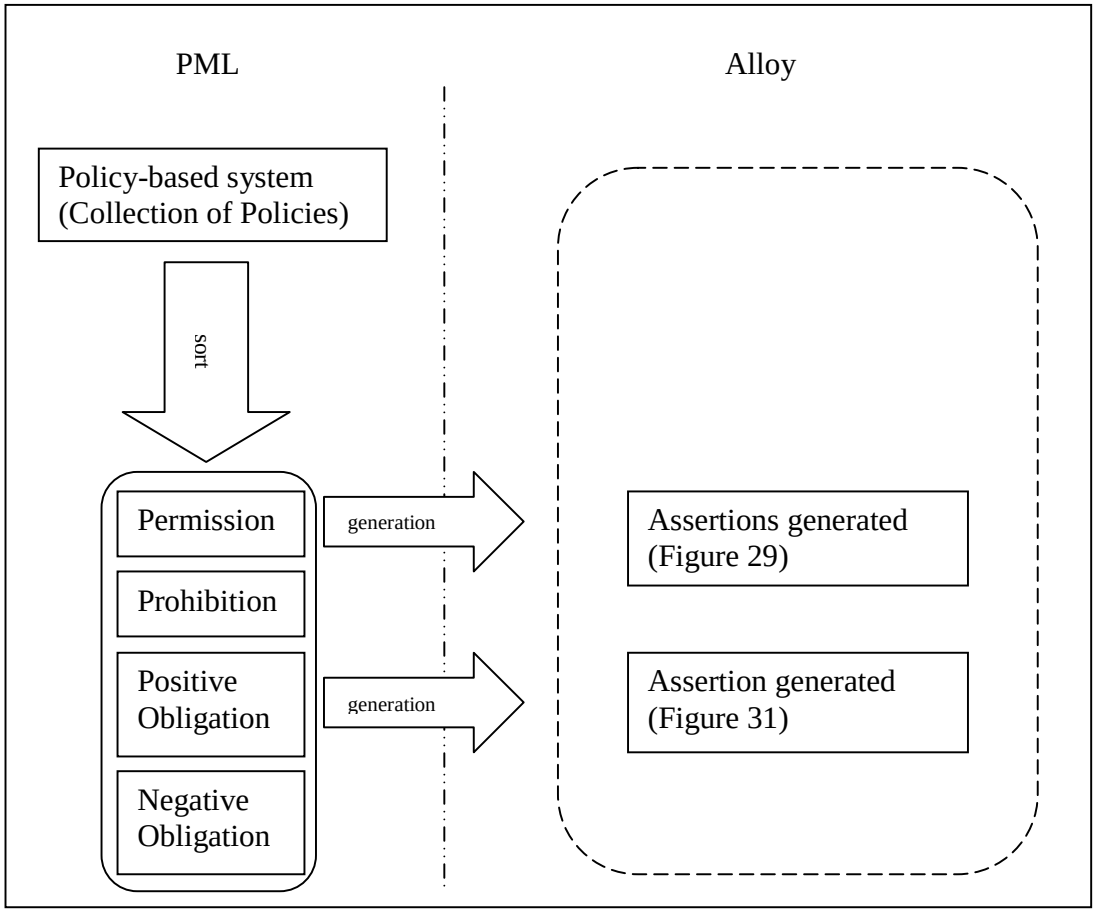


Figure 33 - Generation of Assertions

Figure 33 presents the steps of assertions generation visually. The generated assertions for Positive Obligation will cover O+/O- and O+/A- policy conflict type while assertions generated for Permission will cover the A+/A- policy conflict type. Another possible choice is to generate assertions for Prohibition and Negative Obligation. Assertions generated for Prohibition will cover the A-/O+ and A-/A+ types of policy conflicts while assertions generated for Negative Obligation will cover the O+/O- policy conflict type. Any combination used, one should only be considered to generate assertions for all types of policy conflict types. Steps in Figure 29 generates assertion for

A+/A- policy conflict type, Steps in Figure 30 generates assertion for A+/A- and A-/O+ policy conflict type, Steps in Figure 31 generates assertion for O+/O- and A-/O+ policy conflict type and Steps in Figure 32 generates assertion for O-/O+ policy conflict type. Having all types of policy conflicts covered (as mentioned in Section 2.5.1), a combination of different assertion generation is needed to be selected to cover all means combination of policy conflicts, i.e. O+/O-, A+/A- and O+/A- policy combinations.

6 Outcome and Discussion

In this thesis, we have used a policy language that has been conformant to the principles of MDE. The language of our choice to present policies was PML. Our hypothesis was to use PML meta-model would help us in detecting conflicts at the level of concrete policoes. Meta-model of a language defines structure for that language, restrictions imposed on each element of that language, and other semantics of that language. However, as it has been shown, we were not successful in integrating usage of the meta-model into the viable conflict detection mechanism.

Several factors contributed to this unsuccessful integration. Factors such as structure of the model in Alloy, the way in which Alloy analyses it and relation of the PML meta-model and concrete policies in the Alloy model. The goal of using the meta-model lead us to systematically investigate different approaches for modelling policies. By eliminating and overcoming each approach's restriction(s), we have introduced a modelling approach that leads us to detect policy conflicts (i.e. the third modelling approach).

Different approaches for modelling policies in Alloy have been presented in Section 5. These modelling approaches were aimed to detect conflicts among policies.

The first two modelling approaches presented in this thesis, began with modelling of PML meta-model followed by modelling of concrete policies. As our first steps of modelling policies in Alloy, it seems the best way is to model policies in Alloy along with its meta-model. We also were aware that having PML meta-model in an Alloy

model would give us the ability to check the policies for their syntax in addition to the main goal of ours, policy conflict detection.

In the first and second modelling approaches (c.f. Sections 5.4 and 5.5), we have used the PML meta-model in the Alloy model. Permission and Prohibition policies (signatures) were modelled in Alloy as they were elements of the PML meta-model. They have been used to act as a parent signature for concrete policies. As mentioned earlier, each concrete policy was modelled as a child of the PML meta-model signature (e.g. Permission or Prohibition). The advantage of using this inheritance hierarchy as stated in Section 5.5.1 is being able to write an assertion for conflict detection at the PML meta-model (e.g. Permission or Prohibition signatures) and use it to check all the instances of concrete policy signatures derived from those signatures (i.e. concrete policies). Thus, we only needed to have one assertion for each type of conflicts in our Alloy model.

The techniques applied to model concrete policies in the first and second modelling approaches (Sections 5.4 and 5.5) are similar to the one used to model the policy meta-model. Namely, we have used the UML presentation of the policies and then translated them into the Alloy model with help of UML2Alloy tool, considering the restrictions we have for using this tool (as mentioned in Section 4.3). However, as an outcome of the first modelling approach, unwanted generation of policies in the Alloy model prevented us from taking any further steps to detect conflicts.

In the Second modelling approach (Section 5.5), an attempt was made to restrict generation of unwanted signatures and relations. This restriction helped us to generate only instances of concrete policies that exist in the system. This task was accomplished by using “for” command in Alloy (“for” can be used after “run” or “check” command).

This way we could tell Alloy how many number of instances should be generated for each signatures in the model. After solving the problem of generation of unwanted policies, we have applied conflict detection methods. For detecting conflicts, we used Alloy Assertions. However, using assertion in the second modelling approach (c.f. Section 5.5) caused an error. As mentioned in detail in Section 5.5.2, not being able to distinguish between different relations (relations defined in parent and child signatures) was the source of the problem.

To solve this problem, we have tried to alternate the approach by using assertion at the level of concrete policies (Third modelling approach, Section 5.6). Using assertions at the level of concrete policies will require generation of numerous conflict detection assertions, unlike the limited number of assertion anticipated to be written at the level of PML meta-model (if the second modelling approach has been successful).

Considering the fact that number of assertions is relative to the number of concrete policies within the system, an attempt to investigate the outcome of this approach was carried out in Section 5.6. It has been shown that this modelling approach succeeded in detecting policy conflicts. In this modelling approach, all types of possible conflicts (as discussed in Section 2.5.1) have been successfully detected. Later on, in Section 5.8 a complete method for creating the needed assertions for this modelling approach has been introduced.

In the first, second and fourth modelling approaches, we only used one or two policies to investigate outcome of the modelling approach. These two policies were sufficient to demonstrate the unfeasibility of these approaches since we were discussing the unsuccessful modelling approaches policies. However, in the third modelling

approach, since we needed to testify and demonstrate detection of all possible conflict types, we have used all the policies within our policy-based system as introduced in Section 5.1.

In the fourth modelling approach (Section 5.7), we examined using relations in modelling policies. In the other approaches, focus was to model each concrete policy as a unique signature. However, in the fourth modelling approach, the focus is based on modelling concrete policies using relations. As described and mentioned in Section 5.7, this modelling approach cannot be utilized to detect policy conflicts since no limitations can be put on the number of generated relations. Thus, we encountered the same problem as in the first modelling approach. However, this time no solution can be found to fix this problem.

It is also worth discussing that the fourth modelling approach could be altered in numerous ways. The alteration could be done regarding different combination of relations and signatures to present concrete policies. For example in the first modelling approach, all classes associated with concrete policies were modelled in a unique signature (Figure 27) while all concrete policies and associated classes have been modelled as one signature in the fourth modelling approach. From modelling each concrete policy with one signature to modelling all concrete policies in one signature, different combination of relations and signatures can be thought of. For example, one can think of having policy types as a signature and not as relations in the policy model. Figure 34 presents a sample way of modelling policies in this unique way.

```
Sig policyType {}
Sig permission extends policyType {}
Sig prohibition extends policyType {}
Sig policy {
Ptype: one policyTpe
P: Actor->Action->Entity}
```

Figure 34 - Alternative modelling approach

We have modelled concrete policies mainly based on signatures (in the first, second and the third modelling approaches) and mainly based on relation (in the fourth modelling approach). We have discussed any possible combinations of these approaches to see if we can find any better solution candidate for conflict detection problem using Alloy. However, for the sole purpose of this thesis, no combination of the signatures and relations (i.e. fourth modelling approach and alterations to it) will lead us to a conflict detection approach using Alloy. Having relations as part of modelling concrete policies in Alloy will result in a failure similar to the one we have come across in fourth modelling approach (Section 5.7.1). It is because that Alloy Analyzer extends relations in the model, and nothing can be done to restrict these expansions. Table 7 presents a comparison of all approaches. The summarization makes it more clear on which direction the research question lead us, what held the research back and if a successful conflict detection approach was implemented or not.

Table 7 - Comparison of Modelling Approaches

	Modelling Approach	**Positive Points**	**Negative Points**	**Conflict Detection**
First modelling approach	Based on signatures	Concrete policies extend meta-model allowing for syntax check	Generation of unwanted policies	Not applicable
Second modelling approach	Based on signatures	Capable of checking the syntax plus conflict detection	Alloy Analyzer error. Cannot determine reference value	Not successful
Third modelling approach	Based on signatures	Simple way of modelling sample policies	No PML meta-model connection	Successful
Fourth modelling approach	Based on relations	Unified representation of all policies	Generation of unwanted policies	Not applicable

To summarize the outcome of this thesis, a successful modelling approach was recognized as a suitable solution for detecting conflicts among policies. In the third modelling approach (Section 5.6), we have successfully managed to introduce an approach to detect conflicts among policies within a policy-based system. In this modelling approach, several assertions are needed to be added to our model. The generation of these assertions have been discussed in Section 5.8.

Different modelling approaches presented in this thesis, tried to systematically investigate different possible modelling ways to model concrete policies in Alloy and analyze them. The first and second modelling approaches were using the same method to model meta-model as the concrete policies. Having the meta-model and concrete policies in one model proved not to be a successful path to detect conflicts. Third modelling

approach succeeded in detecting conflicts among policies while it has not had the connection to the meta-model. We have discusses detail steps of third modelling approach in section [5.6].

One can consider using second modelling approaches in addition to the third modelling approach in two separate steps to both analyse the concrete policies and detect conflicts among them. Figure 35 presents the proposed method. In the fourth modelling approach, another possible way to model the policies in Alloy was examined. It was not successful since expansion problem of the model cannot be avoided.

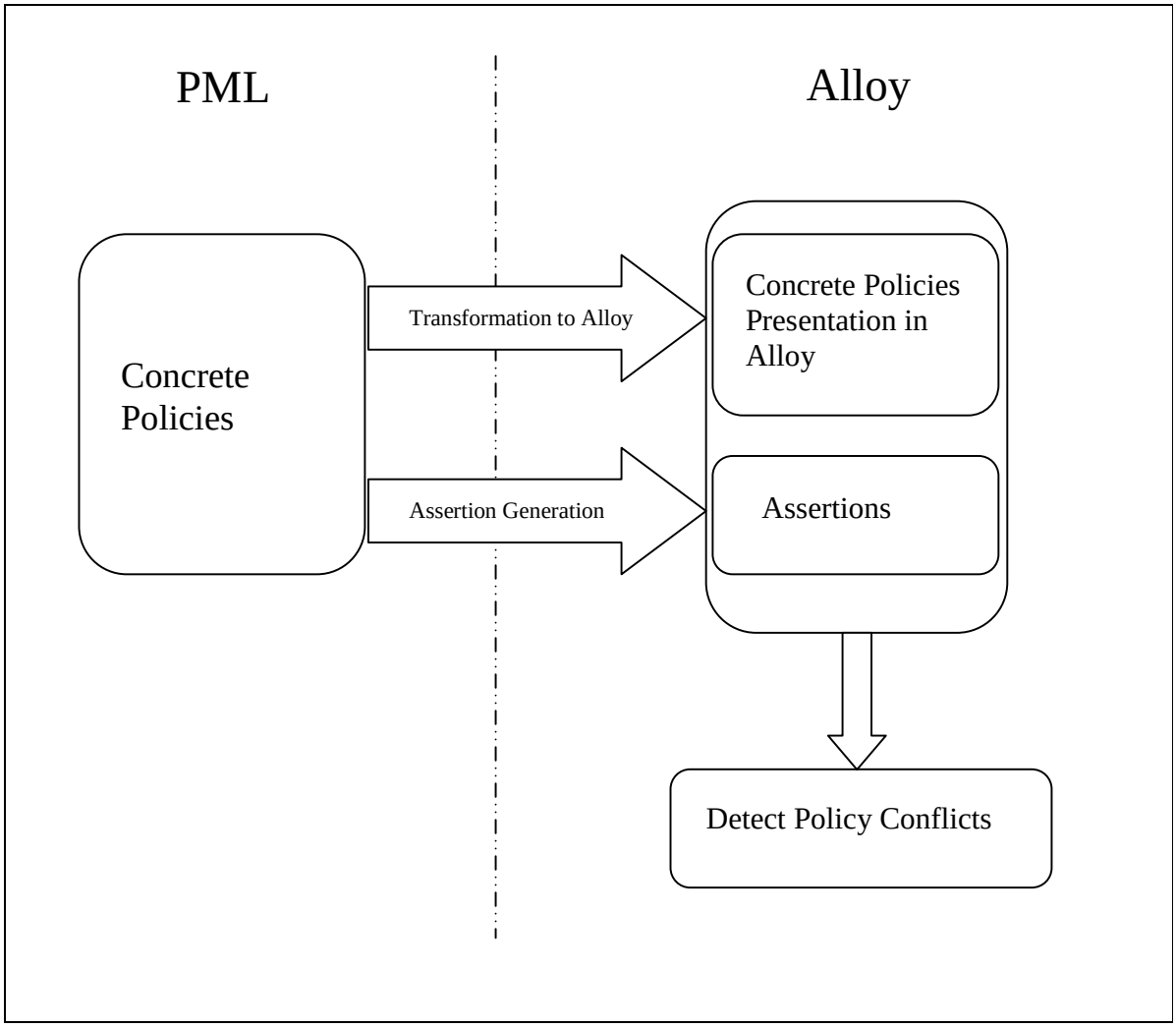


Figure 35 - Proposed Method

In this thesis, we tried to introduce a policy conflict detection method for PML with the help of Alloy. In Section 2.7, different practices of Alloy in different areas have been reviewed. The usage of Alloy varies from RBAC[22] modelling to XACML policy modelling. All of the approaches presented in Section 2.7 have used Alloy and Alloy Analyzer to model and analyze a system or part of it. In the following lines**,** we will

[22] Role Based Access Control

compare those approaches and this thesis's approach. Jacqueline et. al. in [21] used a similar approach to this thesis's approach in a sense that they were also pursuing a modelling of a MOF-complaint meta-model, which later on was expressed as UML and OCL. In their approach, authors only have modelled the meta-model and used Alloy to detect flaws in it. They basically have used the capability of Alloy to generate different instances of the model and then check them against the meta-model in Alloy. They have used the result of generated instances to correct their meta-model. Their approach can be compared to our second modelling approach (Section 5.5). However, in the second modelling approach, we not only have the meta-model present in our model but also the concrete policies within the Alloy model. In [33], authors used Alloy to model a JAAS framework and in [34] and [37] research were carried out to analyze web ontologies. In these studies, authors have not modelled the meta-model of a language ([33]), or have not used the modelled meta-model and instances simultaneously ([34] and [37]). They have provided an Alloy presentation of the system under study and have used Alloy Analyzer to analyse their systems.

Schaeffer et. al. in [22] used Alloy to analyze policy-based interactions. Although the definition and usage of policies are for a specific domain called "Self Managed Cell". However, we do compare their approach with the modelling approaches presented in this thesis. In their study, authors have succeeded to provide a solution for their need of resolving conflicts in their domain. However, at the final comments, it has been mentioned that they are not suggesting the proposed method as a general policy conflict detection method, but more as a method to "unambiguously specify the desired behaviour of interacting Self-Managed Cells". In addition to that, their approach was also meant to

be for a specific domain of “Self Managed Cell”, while the approach we are pursuing is a general approach that can be applied not only to a particular policy language but also we would like to have a complete general conflict detection approach as an outcome.

In literature review (Section 2.7), several studies have been reviewed about Role Based Access Control (RBAC). Researchers have used Alloy to model and analyze RBAC models. Since the concept of RBAC is close to the concept of access control policy, we have studied some of the works in that area. RBAC are used to assign access to resources is a system, similar to the access control policies.

Toachoodee et. al. in [23] used Alloy to analyze a RBAC model. The RBAC model they are analysing has been introduced in their paper for the first time. Although their approach is similar to the approach exercised in this thesis, deeper analysis helps us to distinguish the difference to our approach. In their approach, at the time of modelling of instances (called model transformation in the paper), they present all the constraints and limitations needed by using OCL constraints, simple relations and definition of signatures in Alloy. In addition, the definition of permission was defined to their specific need and in a specific domain. For example, definition of permission is defined by a person’s location and time. Based on those values a permission concept would be deduced in the system. However, a policy, in general, might include greater number of elements than location and time. We cannot assume definition of policy conflicts based on definite number of variables (i.e. location and time) in any domain. In another RBAC related research study using Alloy [24], Schaad and Moffett analysed different RBAC

models in regards to each other. They used Alloy to model and check upon concepts of RBAC[23] within their model.

The main task in their study was to test their proposed model against mentioned concepts in Alloy. Their approach is different from this thesis approach in a sense that they used Alloy to analyse their proposed model and test it against some concepts of RBAC. They have not used Alloy to generate or analyse any instances of their proposed model or the concepts of RBAC but only to analyse them in an Alloy model.

Alloy has also been used to model policy languages. Layouni et. al. in [25] used Alloy to model a policy language called APPEL (discussed in Section 2.7.2). In this approach, authors modelled the APPEL language completely in Alloy. They have modelled policies in Alloy and successfully have detected conflicts between them**.** However, their modelling approach is unique to the APPEL language and cannot be extended to other policy languages. They basically model each element of their language into Alloy in a specific way. However, in our approach we are pursuing a way of modelling a generic policy language's UML profile into Alloy. We are dealing with a more general approach than their approach.

Some studies on modelling of XACML policy language have also been reviewed in Section 2.7.2. Martin and Xie [26] proposed a way to generate policies for XACML policy language using a tool called Crig. Nevertheless, Authors suggested that they could try to use Alloy in their analysis instead of the tool they are already using. Zhang et. al. [27] proposed a way in which they can generate verified XACML policies. However, we

[23] The concepts checked are: Static Separation of Duty (SSoD), Dynamic Separation of Duty (SDSoD) and the Operational Separation of Duty (OpSoD), for the definition of these concepts please refer to [24].

are pursuing an approach to verify existing policies and detect conflict among them. We would like to be able to use any existing policies and then analyze them.

Hughes and Bultan in [28] used partial ordering and eliminated the need for using assertions but only facts in their model. The partial orderings used in their approach were translated into facts. Using those facts, they were able to detect conflicts. Nevertheless, no discussion about generation of different assertions and in what order they are generated can be found.

Table 8 - Comparison of different studies using Alloy

Alloy approaches	**Method used**	**Conflict Detection**	**Note**
A metamodel for the measurement of Object-Oriented systems: an analysis using Alloy [21]	Analyzing model presented in UML and OCL	Meta-model modelled	Detect flaws in the meta-model, used Alloy as a meta-model checker
Verification of Aspect-UML models using Alloy [33], a Combined approach to checking web ontologies [34]	Various modelling based on the research type	No meta-model modelling, modelling based on the specific language	Analyzed model using Alloy
Verification of policy-based self-managed cell interactions using Alloy [22]	policy-based interactions	Conflict detection defined based on the specified domain	Cannot generalize their approach to any domain
Ensuring spatio-temporal access control for real-world applications [23]	introducing a new RBAC model	Conflict detection by OCL	Definition of permission is local to their research area and cannot be generalized
Conflict detection in call control using First-Order Logic model checking [25]	APPEL policy language	No meta-model modelling, modelling based on the specific language	Cannot generalize the approach to our domain
Automated test generation for access control policies via Change-Impact Analysis [26], automated verification of access control policies [28]	XACML	No meta-model modelling, Modelling in their own way and for the specific research.	Different goals, generated verified XACML policies, also mentioning Alloy can be used as future work

Most of the approaches that use Alloy as a counterexample finder rely on Jackson's small scope hypothesis, which suggests that if a bug exists it will appear in small model of a system [20]. Also in most of the modelling approaches discussed in

Section 2.7, the modelled meta-model of the language in Alloy has not been used in the analysis process.

As discussed above in addition to section 2.7, in some approaches, every single element of the language (or system) was modelled in Alloy for analysis. Some innovative approaches (such as using partial ordering or defining concepts of conflict in a simpler way in their domain) have been used to model the system in Alloy and then analyze it. Table 8 presents a summary of some of the studies discussed in this thesis.

A policy conflict detection depends on the definition of the policy language and specific statements stated in its definition. However, none of the studies discussed in this section has introduced an approach to include meta-model of a language in their analysis. In this thesis, we attempted analysis of a system by introducing different modelling approaches in which PML meta-model are present.

We also like to discuss the static conflict detection method used in this thesis. In all policy languages, using a dynamic conflict detection method is quite achievable and it can be simply implemented. As stated earlier in Section 2.5.2, a simple way of monitoring outcome of different applicable policies (if more than one policy is applicable) can lead to the detection of policy conflicts. This can also be enhanced by providing a monitoring service for any changes in the policies, resources, interactions within the domain and other factors in a system. Nevertheless, the main goal of this thesis is not to testify the feasibility of dynamic conflict detection of policies using Alloy. The main concern is to determine various solutions, which can be offered by Alloy for detecting policy conflicts at the design time.

However, as easy the dynamic policy conflict detection method sounds, on the other hand the static conflict detection method is not straightforward and is needed to be explicitly designed and implemented for a system. Static conflict detection is used when a system is not functional and usually before implementation phase. At the time of analysing the system under static conflict detection method, actual data should not be assumed present in the system. Therefore, the static conflict detection method needs to explore various possible states that might happen in a system. Dynamic detection method has no obligation to perform this task. Some policy conflicts could be detected based on the states generated by static conflict detection method. Since these conflicts might not take place in the real world situation, the conflicts detected by static conflict detection method are usually considered as potential conflicts. One could conclude that design and implementation of a static conflict detection method is a demanding task, considering the generation of situations intensely related to the underlying domain.

We also want to discuss the “conditions” used in assertions to detect policy conflicts. As stated in Section 5.8, in the process of generation of assertions and in Section 5.6, the third modelling approach, we only used the “role” element of policies to check if they are conflicting with each other. We have not checked the “action” and “resource” in these conflicts although policies presented and modelled has both elements. We claim that we can use assertions in which not only “role” but also “action” and “resource” are present as well. Not having these elements in conditions checked at the modelling approach will not affect the result of the thesis. We just need to add two more constraints to be checked in each assertion; these constraints are to check “action” and “resource” as we do for the “role”. This follows the same rationale as we have used for

decreasing number of policy elements in Section 5.2. As an example, an assertion checking for “role”, “action” and “resource” between two Permission and Prohibition policy is presented in Figure 36.

```
assert Pol-j-Pol-k { all p1: Pol-j, p2: Pol-k | p1.role = p2.role && p1.action = p2.action
&& p1.resource = p2.resource }
```

Figure 36 - Assertion

We also like to discuss cases in which there is no policy of certain type present within a system. In all of the modelling approaches, we have assumed that at least one policy of each policy type exists. If no single policy from one policy type is present within a system, Alloy will generate a policy from the meta-model. This policy might affect our conflict detection method. In modelling approaches, if we encounter a situation in which one of the policy types does not have any representative concrete policy defined in the system, we will create a dummy policy for that policy type and then continue the analysis. Let us consider an example when there is no Prohibition policy within a system. In this case, the Alloy Analyzer would generate a policy instance from the meta-model class Prohibition. To avoid that we create a dummy prohibition policy as follow: ActorAA is prohibited to accessAA ResourceAA. We will make sure that the selected names for Actor, Action and Resource are not previously available within the system. This task can simply be done by checking the possible values for Actor, Action and Resource within a system. Accomplishing this step, we make sure this policy will not create any possible conflicts with other policies within the system

7 Conclusion

In this thesis, the goal was to investigate Alloy's capability to detect policy conflicts. In addition, we tested the feasibility of having semantics presented by the policy language meta-model as a part of this analysis. Wherever policies are dealt with, one should expect policy conflict to occur. Policy conflict detection is the first step in the detection / resolution process. Policy conflict detection is not a new area of research and various policy conflict detection methods are available. Typically, after detection of a conflict, system's manager will be notified about the conflict. Based on the associated policies and other factors within a system, manager can investigate and find out the cause of the conflict. Either the cause of the conflict should be dealt with or a resolution decision should be assigned to that specific conflict. There are two different types of conflict detection algorithms: One is design time conflict detection (which is called static conflict detection method) and the other one is runtime conflict detection (dynamic conflict detection method). Static conflict detection methods are used before deployment of a system. This type of conflict detection tries to eliminate possible conflicts among policies at the design time of a system. On the other hand, Dynamic conflict detection methods are used when the system is functional and users are using the system. Static conflict detection methods are useful to find conflicts between existing policies while Dynamic conflict detection methods are more useful when dealing with other causes of conflicts in the system.

.

The goal of this thesis is to investigate whether Alloy is an appropriate tool for the static conflict detection. This means, first we investigated whether we can use Alloy as a tool to model policies and second to detect conflicts among these modelled policies. Alloy is a language for model checking. Alloy comes with a tool called Alloy Analyzer, which is helpful in the process of checking the model by providing an environment for Alloy language and also by generating instances of the model. After having a model in Alloy, one can ask Alloy Analyzer to check the model to see if the model is consistent or not and if Alloy Analyzer can find any counterexample interfering with part (or whole) model or not. Alloy Analyzer's role in this thesis was to analyze the model, i.e. to find any inconsistencies in the model, which in case of policies means to find conflicts among policies

We also investigated usage of meta-model in analysis of policy conflicts. Presenting policies, we used a language called Policy Modelling Language (PML). PML is based on Model Driven Engineering principles, thus PML has a meta-model defined. We transferred the PML meta-model presentation in UML into an Alloy model. This transformation was done with the help of UML2Alloy. We systematically explored different ways to represent policies in Alloy. We have begun the modelling with the modelling of PML meta-model and concrete policies in a unique Alloy model (first and second modelling approaches). Then, we have presented a modelling approach which only contains the concrete policies in an Alloy model (third modelling approach) and finally we have presented an alternative modelling of concrete policies in Alloy using relations (fourth modelling approach). For each modelling approach, we attempted to

come up with a conflict detection mechanism. We have used Alloy's reasoning capability (namely Assertion in Alloy) for this task. Each of the modelling approaches has its own drawbacks and advantages.

In this thesis, we have shown that Alloy also can be used not only to model concrete policies in PML but also to help us find counterexamples of the model and lead us to detect conflicts among concrete policies.

Analysis utilized by using Alloy and PML was an innovative approach, which has not been studied before. Inspired by concept of MDE, in this thesis we have investigated analysis of PML meta-model along with its instances in a model. We have shown through different modelling approaches how to use the meta-model of PML in accordance with our goal of policy conflict detection. The outcome of this thesis confirms the previous similar researches done in this area, but introduced an innovative outcome, the possibility of analysing a system meta-model and its concrete instances simultaneously using Alloy.

Within the steps of reaching the goal of this thesis, we also have managed to provide a full presentation of PML meta-model in Alloy. This model can be counted as an alternative outcome of this thesis. The complete code of PML meta-model in Alloy can be found in [51] while part of that is included in Appendix 1 (Section 9).

8 Resources

1. Hassan, W., Logrippo, L.: Governance Policies for Privacy Access Control and their Interactions. In: Feature Interactions in Telecommunications and Software Systems VIII, ICFI'05, IOS Press, pp. 114-130. (2005)
2. Lupu, E., Sloman, M.: Conflict Analysis for Management Policies. In: Proceeding to the 7th international symposium on Integrated Network Management IM'97 (formerly known as ISINM), San-Diego (USA), Chapman & Hall, pp. 430-443. (1997)
3. Moffett, J., Sloman, S.: Policy conflict analysis in distributed system management. In: Journal of organizational computing, pp. 1-22. (1997)
4. Dunlop, N., Indulska, J., Raymond, K.: Dynamic conflict detection in policy-based management systems. In: Proceeding of International enterprise distributed object computing conference, IEEE Computer Society, pp 1-15. (2002)
5. Kamoda, H., Hayakawa, A., Yamaoka, M., Matsuda, S., Broda, K., Sloman, M.: Policy conflict analysis using Tabeleaux for On demand VPN network. In: Proceeding of Sixth international symposium on a world of wireless mobile and multimedia networks, pp. 565-569. (2005)
6. Syukur, E., Loke, S., Stanski, P.: Methods for Policy Conflict Detection and Resolution in Pervasive Computing Environments. In: Proceeding of Policy Management for Web workshop in conjunction with WWW2005 Conference, Chiba, Japan, pp. 10-14 (2005)
7. Uszok, A., Bradshaw, J. , Jeffers, R., Tate, A., Dalton, J.: Applying KAoS services to ensure policy compliance for semantic web services workflow composition and enactment. In: S. A. McIlraith, D. Plexousakis, F. van Harmelen (Eds.), The Semantic

Web—ISWC 2004, Proceedings of the Third International Semantic Web Conference, pp. 425-440. (2004)

8. Uszok, A., Bradshaw, J., Johnson, M., Jeffers, R., Tate, A., Dalton, J., Aitken, S.: KAoS policy management for semantic web services. IEEE Intelligent Systems, vol. 19, no. 4, pp. 32-41. (2004)
9. Bradshaw, J. M., Uszok, A., Jeffers, R., Suri, N., Hayes, P., Burstein, M., Acquisti, A., Benyo, B., Breedy, M., Carvalho, M., Diller, D., Johnson, M., Kulkarni, S., Lott, J., Sierhuis, M., Van Hoof, R.: Representation and reasoning for DAML-based policy and domain services in KAoS and Nomads. In: Proceedings of the Autonomous Agents and Multi-Agent Systems Conference (AAMAS 2003). ACM Press, pp. 835-842. (2003)
10. Kagal, L.: Rei: A Policy Language for the Me-Centric Project. Hewlett Packard Labs Technical Report. (2002)
11. Kaviani, N.: Web Rules to Interchange Policies. In: Master of Science Thesis in the School of Interactive Arts and Technology, Simon Fraser University. (2007)
12. Uszok, A., Bradshaw, J., Jeffers, R., Suri, N., Hayes, P., Breedy, M., Bunch, L., Johnson, M., Kulkarni, S., Lott, J.: KAoS Policy and Domain Services: Toward a Description-Logic Approach to Policy Representation, Deconfliction, and Enforcement. In: Proceedings of 4th IEEE Workshop on Policies for Networks and Distributed Systems (Policy 2003), IEEE, pp. 93-96. (2003)
13. Damianou, N., Dulay, N., Lupu, E., Sloman, M.: The Ponder Policy Specification Language Proc. In: Proceeding of workshop on Policies for Distributed Systems and Networks, Bristol, UK, pp. 18-39. (2001)

14. Moses, T.: eXtensible Access Control Markup Language (XACML) Version 2.0. OASIS Standard (2005)
15. Anastasakis, K.: UML2Alloy Reference Manual, UML2Alloy Version: 0.52 [Online] available at http://www.cs.bham.ac.uk/~bxb/UML2Alloy/files/uml2alloy_manual.pdf (retrieved 01/09/2009)
16. Gogolla, M., Richters, M.: Transformation rules for UML class diagrams. In: Selected papers from the First International Workshop on The Unified Modeling Language UML. pp. 92–106. (1999)
17. Kaviani, N., Gasevic, D., Milanovic, M., Hatala, M.: Model-Driven Engineering of a General Policy Modeling Language. In: Proceedings of the 9th IEEE Workshop on Policies for Distributed Systems and Networks (POLICY 2008), pp.101-104. (2008)
18. Wagner, G., Giurca, A., Lukichev, S.: A General Markup Framework for Integrity and Derivation Rules. Dagstuhl Seminar Proceedings, Principles and Practices of Semantic Web Reasoning (2006). [Online] Available at: http://drops.dagstuhl.de/opus/volltexte/2006/479 (retrieved 01/09/2009)
19. Giurca, A. Wagner, G.: Rule Modeling and Interchange. In: Ninth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2007), pp.485-491. (2007)
20. Jackson, D.: Software Abstractions, Logic, Language, and Analysis, MIT Press, ISBN-10: 0-262-10114-9. (2006)
21. McQuillan, J., Power, J.: A Metamodel for the Measurement of Object-Oriented Systems: An Analysis using Alloy. In: Proceeding of International Conference on Software Testing, Verification, and Validation, pp. 288-297. (2008)

22. Schaeffer-Filho, A., Lupu, E., Sloman, M., Eisenbach S.: Verification of Policy-Based Self-Managed Cell Interactions Using Alloy. In: Proceedings of the 2009 IEEE International Symposium on Policies for Distributed Systems and Networks. pp. 37-40. (2009)
23. Toahchoodee, M., Ray, I. Anastasakis, K., Georg, G., Bordbar, B.: Ensuring Spatio-Temporal Access Control for Real-World Applications. In: Proceedings of the 14th ACM symposium on Access control models and technologies. pp. 13-22. (2009)
24. Schaad, A., Moffett, J.: A Lightweight Approach to Specification and Analysis of Rolebased Access Control Extensions. In: Proceedings of the seventh ACM symposium on Access control models and technologies. pp. 13-22. (2002)
25. Layouni, a., Logrippo, L., Turner K.: Conflict Detection in Call Control Using First-Order Logic Model Checking. In: Proceeding of 9th. Feature Interactions in Telecommunications and Software Systems. pp. 66-82. (2007)
26. Martin, E., Xie, T.: Automated Test Generation for Access Control Policies via Change-Impact Analysis. In: Proceeding of Third International Workshop on Software Engineering for Secure Systems (SESS'07). pp. 1-5. (2007)
27. Zhang, N., Ryan, M., Guelev, D.: Synthesising Verified Access Control Systems in XACML. In: Proceedings of the 2004 ACM workshop on Formal methods in security engineering. pp. 56-65. (2004)
28. Hughes, G., Bultan, T..: Automated verification of access control policies. Technical Report, Department of Computer Science, University of California, Santa Barbara. [Online] Available at http://www.cs.ucsb.edu/~bultan/publications/tech-report04.pdf (2004) (retrieved 01/09/2009)

29. Lin, A., Bond, M., Clulow, J.: Modeling Partial Attacks with Alloy. In: Proceeding of Security Protocols Workshop (SPW'07). pp. 1-15. (2007)
30. Filho, F., Brito, P., Rubira, C.: A Framework for Analyzing Exception Flow in Software Architectures. In: SIGSOFT Softw. Eng. Notes, Volume 30, Number 4. pp. 1-5. (2005)
31. Nakajima, S., Tamai, T.: Formal Specification and Analysis of JAAS Framework. In: Proceedings of the 2006 international workshop on Software engineering for secure systems. pp. 59-64. (2006)
32. Shaffer, A., Auguston, A., Irvine, C., Levin, T.: A Security Domain Model to Assess Software for Exploitable Covert Channels. pp. 45-56. (2008)
33. Mostefaoui, F., Vachon, J.: Verification of Aspect-UML models using Alloy. In: Proceedings of the 10th international workshop on Aspect-oriented modeling. pp. 41-48. (2007)
34. Dong, J., Lee, C., Lee, H., Li, Y., Wang, H.: A Combined Approach to Checking Web Ontologies. In: Proceedings of the 13th international conference on World Wide Web. pp. 714-722. (2004)
35. Hu, H., Ahn, G.: Enabling Verification and Conformance Testing for Access Control Model. In: Proceedings of the 13th ACM symposium on Access control models and technologies. pp. 195-204. (2008)
36. Kolovski, V., Hendler, J., Parsia, B.: Analyzing Web Access Control Policies. In: Proceedings of the 16th international conference on World Wide Web. pp. 677-686. (2007)
37. Wang H., Dong J., Sun J., Sun J.: Reasoning support for semantic web ontology family languages using Alloy. In: International Journal of Multiagent and Grid Systems,

Special issue on Agent-Oriented Software Development Methodologies. Volume 2, Issue 4. pp. 455-471. (2006)

38. Armac I., Kirchhof M., Manolescu L.: Modeling and Analysis of Functionality in eHome systems: Dynamic Rule-based Conflict Detection. In: Proceeding of the 13th Annual IEEE international symposium and workshop on Engineering and Computer Based Systems, pp. 219-228. (2006)
39. Bezivin J.: On the Unification Power of Models. In: Software and System Modeling, Volume 4, Issue 2, pp. 171-188. (2005)
40. URML: a UML-Based Rule Modeling Language. [Online] Available: http://oxygen.informatik.tu-cottbus.de/rewerse-i1/?q=node/7 (retrieved 01/09/2009)
41. Warmer J., Kleppe A.: The object constraint language: precise modeling with UML. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA. (1998)
42. Noy N., McGuinness D.: Ontology Development 101: A Guide to Creating Your First Ontology. Technical Report KSL-01-05, Stanford Knowledge Systems Laboratory. (2001)
43. Ribarić M., Sheidaei S., Gašević D., Milanović M., Giurca A., Lukichev S., Wagner G.: Modeling of Web Services using URML, R2ML and model transformations.: In Giurca, A., Gašević, D., Taveter, K., (Eds.). Handbook of Research on Emerging Rule-Based Languages and Technologies: Open Solutions and Approaches, IGI Publishing. pp. 422-446. (2009)
44. ArgoUML: A UML modeling tool. [Online] Available: http://argouml.tigris.org/
45. Hunter D., et al.: Beginning XML. Wiley, ISBN: 978-0-470-11487-2. (2007)

46. Lupu E., Sloman M.: Conflicts in policy-based distributed systems management. In: IEEE transactions of software engineering, Volume 25, Number 6, pp. 852-869. (1999)
47. JTP- Java Theorem Prover. [Online] Available at http://www.ksl.stanford.edu/software/JTP/ (retrieved 01/09/2009)
48. R2ML to JBoss Rules. [Online] Available at http://oxygen.informatik.tu-cottbus.de/translator/R2MLtoJBossRules/ (retrieved 01/09/2009)
49. R2ML to OCL. [Online] Available at http://oxygen.informatik.tu-cottbus.de/translator/R2MLtoOCL/ (retrieved 01/09/2009)
50. Dean M., Schreiber G., et al.: OWL Web Ontology Language Reference. W3C Recommendation 10 [Online] Available at http://www.w3.org/TR/owl-ref/ (retrieved 01/09/2009)
51. PML Alloy model. [Online] Available at http://sheidaei.com/shahin/pml/ (retrieved 01/09/2009)
52. Zhang, N., Ryan, M., Guelev, D.: Synthesising verified access control systems in XACML. In: the 2004 ACM Workshop on Formal Methods in Security Engineering, Washington DC, USA, ACM Press. pp. 56-65. (2004)
53. Gaševic, D., Djuric, D. Devedžic, V.: Model Driven Engineering in Model Driven Engineering and Ontology Development. Springer Berlin Heidelberg, 2009, ISBN: 978-3-642-00281-6
54. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification , Version 1.0 [Online] Available at http://www.omg.org/spec/QVT/1.0/PDF

55. Kelsen, P., Ma, Q.: A Lightweight Approach for Defining the Formal Semantics of a Modeling Language. In: Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems, pp. 690-704 (2008)
56. XMI Mapping Specification, v2.1.1, 2007, [Online] Available at http://www.omg.org/spec/XMI/2.1/PDF (retrieved 01/11/2009)
57. Lassila, O., Swick, R., et al.: Resource Description Framework (RDF) Model and Syntax Specification (1999)

9 Appendix 1 – PML Meta-model Representation in Alloy

9.1 Definition of Signatures

Note: Only first three pages of the listing are presented here for the demonstration purposes. Complete Alloy representation of PML meta-model is available from [51].

```
module untitledModel
 sig RuleBase{
        vocabulary:lone Vocabulary,
        ruleset:set RuleSet,
        externalvocabulary:set ExternalVocabulary}
 sig Vocabulary{
        vocabularyentery:some VocabularyEntery}
 sig RuleSet{
        rulebase:one RuleBase,
        variable:set Variable,
        vocabulary:lone Vocabulary,
        externalvocabulary:one ExternalVocabulary}
 sig ExternalVocabulary{
}
 sig ExternalVocabularyLanguage{
}
 sig Variable{
        ruleset:set RuleSet,
```

```
        objectvariable:lone ObjectVariable,
        genericvariable:lone GenericVariable,
        datavariable:one DataVariable}
sig IntegrityRuleSet extends RuleSet{
        integrityrule:set IntegrityRule}
sig DerivationRuleSet extends RuleSet{
        derivationrule:set DerivationRule}
sig ReactionRuleSet extends RuleSet{
        reactionrule:set ReactionRule}
sig ProductionRuleSet extends RuleSet{
        productionrule:set ProductionRule}
sig ReactionRule{
        andornafnegformula:some AndOrNafNegFormula,
        andornafnegformula:set AndOrNafNegFormula,
        eventexpression:one EventExpression,
        eventexpression:one EventExpression}
sig ProductionRule{
        andornafnegformula:some AndOrNafNegFormula,
        programactionexpression:some ProgramActionExpression,
        andornafnegformula:set AndOrNafNegFormula}
sig DerivationRule{
        andornafnegformula:some AndOrNafNegFormula,
        literalconjunction:one LiteralConjunction,
```

```
        objectdescriptionatom:one ObjectDescriptionAtom,
l       ogicalformula:one LogicalFormula}
sig IntegrityRule{
        logicalformula:one LogicalFormula}
sig AlethicIntegrityRule extends IntegrityRule{
}
sig DeonticIntegrityRule extends IntegrityRule{
}
sig LogicalFormula{
        implication:one Implication,
        derivationrule:one DerivationRule}
sig AndOrNafNegFormula{
        disjunction:lone Disjunction,
        conjunction:lone Conjunction,
        atom:lone Atom}
sig LiteralConjunction{
        atom:some Atom}
sig ProgramActionExpression extends AtomicEventExpression{
}
sig EventExpression{
        obligationordispensation:one ObligationOrDispensation}
sig Conjunction extends LogicalFormula{
        logicalformula:set LogicalFormula,
```

andornafnegformula:set AndOrNafNegFormula,

andornafnegformula:one AndOrNafNegFormula}

9.2 Definition of Facts

Note: Only first three pages of the listing are presented here for the demonstration purposes. Complete Alloy representation of PML meta-model is available from [51].

fact Asso_Vocabulary_vocabulary_rulebase_RuleBase { RuleBase <: vocabulary in (RuleBase) one->lone (Vocabulary) }

fact Asso_RuleSet_ruleset_rulebase_RuleBase { RuleSet <: rulebase in (RuleSet) set->one (RuleBase) && RuleBase <: ruleset in (RuleBase) one->set (RuleSet) }

fact Asso_RuleBase_rulebase_externalvocabulary_ExternalVocabulary { RuleBase <: externalvocabulary in (RuleBase) one->set (ExternalVocabulary) }

fact Asso_Variable_variable_ruleset_RuleSet { Variable <: ruleset in (Variable) set->set (RuleSet) && RuleSet <: variable in (RuleSet) set->set (Variable) }

fact Asso_RuleSet_ruleset_vocabulary_Vocabulary { RuleSet <: vocabulary in (RuleSet) one->lone (Vocabulary) }

fact Asso_IntegrityRuleSet_integrityruleset_integrityrule_IntegrityRule { IntegrityRuleSet <: integrityrule in (IntegrityRuleSet) set->set (IntegrityRule) }

fact Asso_DerivationRuleSet_derivationruleset_derivationrule_DerivationRule { DerivationRuleSet <: derivationrule in (DerivationRuleSet) set->set (DerivationRule) }

fact Asso_ProductionRuleSet_productionruleset_productionrule_ProductionRule { ProductionRuleSet <: productionrule in (ProductionRuleSet) set->set (ProductionRule) }

```
fact Asso_ReactionRuleSet_reactionruleset_reactionrule_ReactionRule {
ReactionRuleSet <: reactionrule in  ( ReactionRuleSet) set->set ( ReactionRule) }
fact Asso_RuleSet_ruleset_externalvocabulary_ExternalVocabulary { RuleSet <:
externalvocabulary in  ( RuleSet) set->one ( ExternalVocabulary) }
fact Asso_IntegrityRule_integrityrule_logicalformula_LogicalFormula { IntegrityRule <:
logicalformula in  ( IntegrityRule) set->one ( LogicalFormula) }
fact Asso_AndOrNafNegFormula_andornafnegformula_derivationrule_DerivationRule {
DerivationRule <: andornafnegformula in  ( DerivationRule) set->some (
AndOrNafNegFormula) }
fact Asso_DerivationRule_derivationrule_literalconjunction_LiteralConjunction {
DerivationRule <: literalconjunction in  ( DerivationRule) set->one ( LiteralConjunction)
}
fact Asso_AndOrNafNegFormula_andornafnegformula_productionrule_ProductionRule
{ ProductionRule <: andornafnegformula in  ( ProductionRule) set->some (
AndOrNafNegFormula) }
fact
Asso_ProductionRule_productionrule_programactionexpression_ProgramActionExpressi
on { ProductionRule <: programactionexpression in  ( ProductionRule) set->some (
ProgramActionExpression) }
fact Asso_ProductionRule_productionrule_andornafnegformula_AndOrNafNegFormula
{ ProductionRule <: andornafnegformula in  ( ProductionRule) set->set (
AndOrNafNegFormula) }
```

```
fact Asso_ReactionRule_reactionrule_andornafnegformula_AndOrNafNegFormula {
ReactionRule <: andornafnegformula in  ( ReactionRule) one->some (
AndOrNafNegFormula) }
fact Asso_AndOrNafNegFormula_andornafnegformula_reactionrule_ReactionRule {
ReactionRule <: andornafnegformula in  ( ReactionRule) one->set (
AndOrNafNegFormula) }
fact Asso_ReactionRule_reactionrule_eventexpression_EventExpression { ReactionRule
<: eventexpression in  ( ReactionRule) one->one ( EventExpression) }
fact Asso_EventExpression_eventexpression_reactionrule_ReactionRule { ReactionRule
<: eventexpression in  ( ReactionRule) one->one ( EventExpression) }
fact Asso_Conjunction_conjunction_logicalformula_LogicalFormula { Conjunction <:
logicalformula in  ( Conjunction) one->set ( LogicalFormula) }
```

9.3 Definitions of Predicates

Note: Only first three pages of the listing are presented here for the demonstration purposes. Complete Alloy representation of PML meta-model is available from [51].

```
pred Vocabulary_voca1[]{
        all p: Vocabulary | some a: RuleSet | p in a.rp5
}
pred Vocabulary_voca2[]{
        all p: Vocabulary | all a1, a2: RuleSet | (
        (p in a1.rp5) &&(p in a2.rp5))
         =>
```

```
        (a1 = a2)
}
pred Vocabulary_voca21[]{
        all p: Vocabulary | some a: RuleBase | p in a.rp32
}
pred Vocabulary_voca22[]{
        all p: Vocabulary | all a1, a2: RuleBase | (
        (p in a1.rp32) &&(p in a2.rp32))
         =>
         (a1 = a2)
}
pred RuleSet_rule1[]{
        all p: RuleSet | some a: RuleBase | p in a.rp31
}
pred RuleSet_rule2[]{
        all p: RuleSet | all a1, a2: RuleBase | (
        (p in a1.rp31) &&(p in a2.rp31))
         =>
         (a1 = a2)
}
pred ReactionRule_reac1[]{
        all p: ReactionRule | some a: ReactionRuleSet | p in a.rp4
}
```

```
pred ReactionRule_reac2[]{
        all p: ReactionRule | all a1, a2: ReactionRuleSet | (
        (p in a1.rp4) &&(p in a2.rp4))
         =>
         (a1 = a2)
}
pred ProductionRule_prod1[]{
        all p: ProductionRule | some a: ProductionRuleSet | p in a.rp3
}
pred ProductionRule_prod2[]{
        all p: ProductionRule | all a1, a2: ProductionRuleSet | (
        (p in a1.rp3) &&(p in a2.rp3))
         =>
         (a1 = a2)
}
```

10 Appendix 3 – Health Domain Policies

This section presents policies introduced in Section 5.1 in their UML presentation.

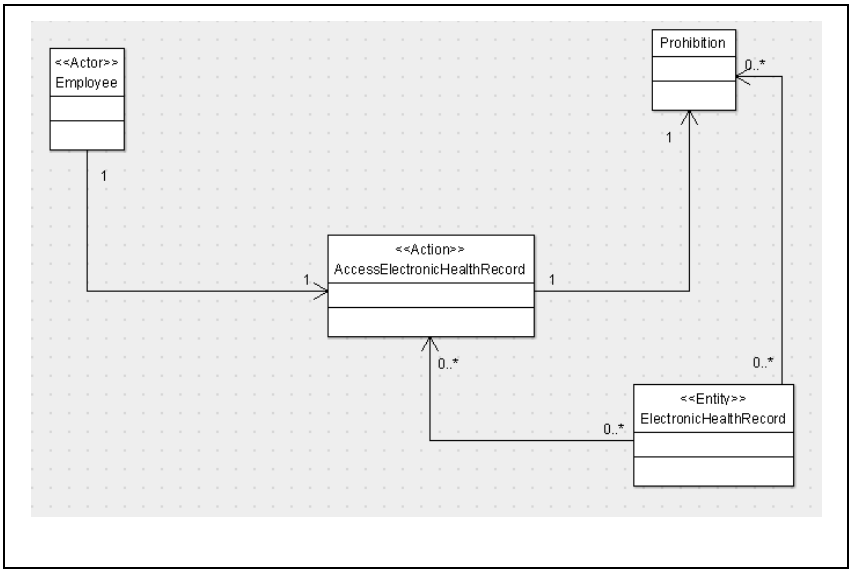


Figure 37 - UML presentation of Pol2

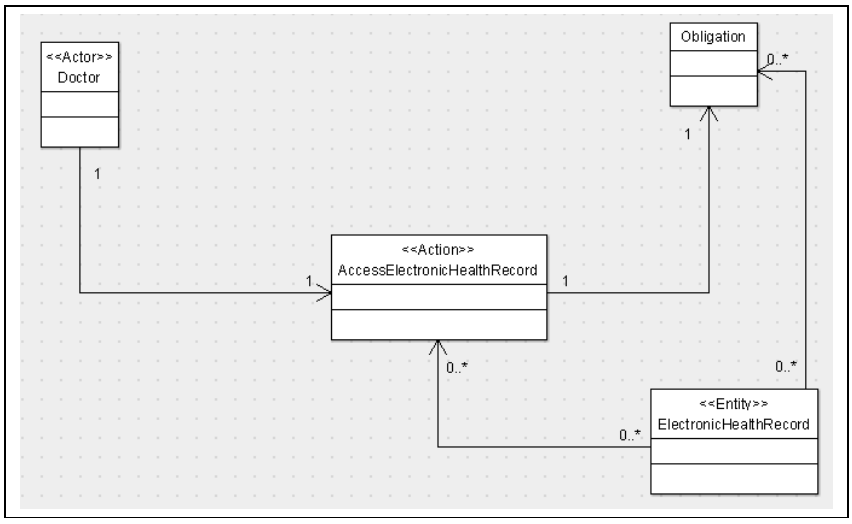


Figure 38 - UML presentation of Pol3

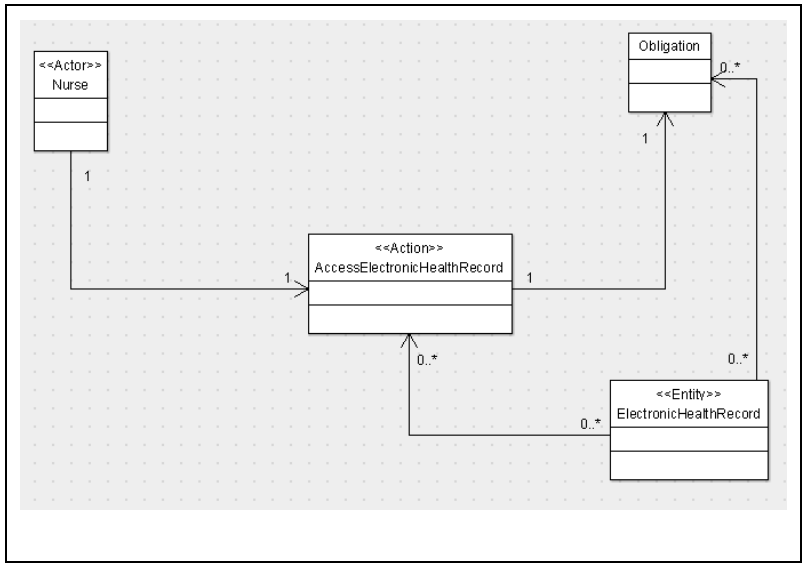


Figure 39 - UML presentation of Pol4

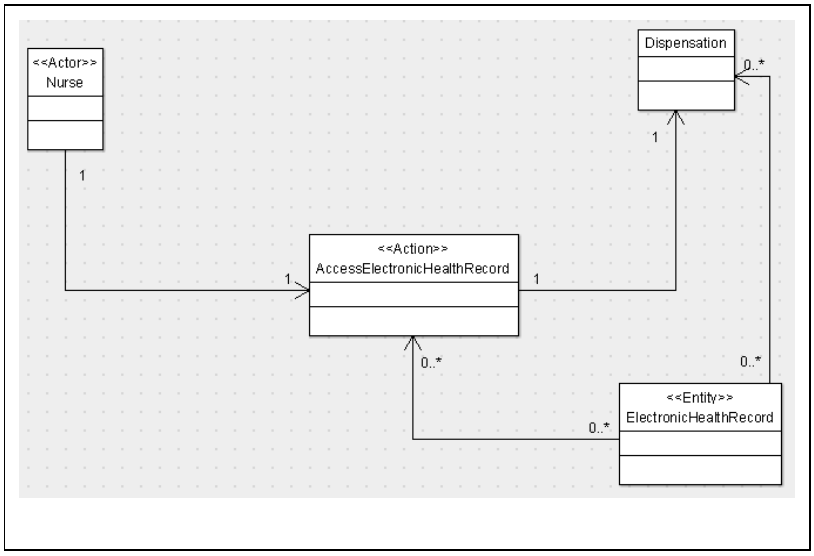


Figure 40 - UML presentation of Pol5

11 Appendix 2 – Alloy

An Alloy model consists of a number of signatures (sig in Alloy) model. Within and among these signatures, one can define relations that relate these signatures to one another. Alloy language is based on relations. Everything would finally translate into relations and passed to the SAT solver from Alloy Analyzer. Presenting this relations, building blocks of Alloy language are Signatures. However, for the complete definition, it would be best to study [20].

The essential constructs of Alloy are as follows:

- Signature describes the properties of a set of entity objects. It introduces a given type, which consists of a collection of relations (called fields) and a set of predicates representing the constraints on the fields. A signature may extend fields and constraints from another signature. Signature is expressed as ‘sig’ in Alloy.
- Fact is a constraint on relations and objects that is always true within the specification. It is a formula that takes no arguments and does not need to be invoked explicitly. Fact is expressed as ‘fact’ in Alloy.
- Predicate is a template for a parameterized constraint. It can be applied elsewhere by instantiating the parameters. A predicate is always evaluated to either true or false. Predicate is expressed as ‘pred’ in Alloy.
- Function is a template for a parameterized expression. It can be applied elsewhere by instantiating the parameters. A function evaluates to a value. Function is expressed as ‘fun’ in Alloy.

- Assertion is a constraint that is intended to follow from the facts in a model. It is a formula whose correctness needs to be checked, assuming the facts in the model. Assertion is expressed as ‘assert’ in Alloy.

11.1 Signature

A signature is a definition for a set of atoms. In the following lines, first, we define a signature called “MySig” and “MyAtt” signatures. The declaration for signature is shown in Figure 41.

```
sig MySig {}
sig MyAtt {
myrelation: one MySig
}
```

Figure 41 - Signature definition in Alloy

Signatures, usually contains relations. Signature “MyAtt” is a signature that has a relation called “myrelation” which relates “MyAtt” to one “MySig”. They also can be extended from other previously defined signatures. Figure 42 defines a signature called “MyAttExt”. It has been extended from “MyAtt” signature. Therefore, it inherits the relation “myrelation” to the signature “MySig” from “MyAtt”.

```
Sig MyAttExt extends MyAtt {}
```

Figure 42 - Extension and Relations in Alloy

Signatures can be defined as abstract. Abstract signatures cannot be instantiated from within a model (by Alloy Analyzer), however they can be further extended. Figure 43 presents definition of an abstract signature “AbSignature” and shows how it can be extended to “notAb”.

```
abstract sig AbSignature {}
sig notAb extends AbSignature {}
```

Figure 43 - Abstract signatures in Alloy

11.2 Operators

Alloy’s operators can be categorized into three classes: set operators, logical operators and the relational operators. The standard set and logical operators presented in Table 9.

Table 9 - Alloy set and logical Operations

Symbol	definition
+	union
-	difference
&	intersection
In	subset
=	Equality
!	negation
&&	conjunction
||	disjunction

Relational operators supported in Alloy can be named as product, join, transitive closure, reflexive-transitive closure, transpose, domain restriction, range restriction and relational override.

11.3 Functions / Facts

Alloy includes concept of functions and facts. These concepts help Alloy to force constraints on the model in different ways.

Facts are used for the constraints that are assumed always to hold true in the model. Functions (like functions in any programming language) are used to apply constraints on selected signatures and relations. This selection is by telling Alloy to use the function on what combination of signatures and relations. It is like passing arguments in programming languages such as C++ or JAVA.

```
Fact
fact{
Permission->Doctor->Access->ElectronicHealthRecord in policy.p
}
Function
fun redLights (s: LightState): set Light {s.color.Red}
Predicate
pred mostlyRed (s: LightState, j: Junction) {
lone j.lights – redLights(s)
}
```

Figure 44 - Facts, Predicates, Functions and Assertions in Alloy

Figure 44 presents an example of fact, predicate and function in Alloy. The fact presented here inserts a combination of instances into p relation of a policy signature. Function and Predicate example are originated from [20] where a model is discussed for cross road lights in a junction. The presented predicate, with the use of presented function, constraints the junction so that all lights but one at most is showing red.

11.4 Run / Check

```
Assertion
assert nop{ all p: Prohibition ,  q: Permission | q->role = p->role }
Predicate
pred mostlyRed (s: LightState, j: Junction) {
lone j.lights – redLights(s)
        }
```

Figure 45 - Assertion – Predicate

The purpose of modelling in Alloy (or any language) is to be able to analyze that model. In Alloy, two different commands (Run and Check) are used to execute the

analyze process on a model. A Run command tells Alloy Analyzer to search for an instance for a specific Predicate. Alloy tries to find a instance of the model to hold all the constraints and reports it to the user. The Check command tries to search for a counterexample of any Assertion in the model.

An assertion in the Alloy is a constraint that is intended to follow from the facts in the model. The Alloy Analyzer checks assertions. If assertions do not hold, it will be reported as a result of analysis. The assertions are mostly used to find possible counterexamples within the model.

Predicates are like assertions. The only difference is that Alloy Analyzer tries to find an instance within the model that satisfies all the constraints within the model.

A scope is an element that bounds the size of instance or the counterexample that Alloy Analyzer tries to generate. There is a default scope set for Alloy Analyzer. If no definite scope is set then the default one is used. In the Figure 46, an example showing usage of Run and Check with the scope specified.

```
Run
run SamplePred for 4 but 4 int, 1 Permission1, 1 Doctor

Check
check SampleAssertion for 3 int, 1 Permission, 1 Doctor
```

Figure 46 - Scope, Run and Check in Alloy