STRUCTURAL DOMAIN MODELLING FOR POLICY LANGUAGE SPECIALIZATION WITH CONFLICT ANALYSIS

Thesis submitted for the degree of
Doctor of Philosophy
at the University of Leicester

by
Zohra Ahsan Khowaja
Department of Computer Science
University of Leicester

8th June 2012

Declaration

The content of this submission was undertaken in the Department of Computer Science, University of Leicester, and supervised by Dr. Stephan Reiff-Marganiec during the period of registration. I hereby declare that the materials of this submission have not previously been published for a degree or diploma at any other university or institute. All the materials submitted is the result of my own research except as cited in the references. Research work presented in Chapter 3 has been previously published in [47].

Acknowledgements

First of all and foremost, I thank to my ALLAH almighty for giving me strength, will-power, patience against many odds and fulfilling my prayers, ALHAMDULILLAH.

This dissertation would not have been possible without the guidance and the help of several individuals who contributed and extended their valuable assistance in completion of this study. First and foremost I offer my sincerest gratitude to my supervisor, Dr Stephan Reiff-Marganiec, who has supported me throughout my thesis with his patience and knowledge whilst allowing me the room to work in my own way. I would like to thanks to my second supervisor, Dr. Artur Boronat for his support and guidelines.

My parents deserve special thanks for their inseparable support and prayers. My Father, who has been not with me since very long, but his inspiration imparted me as he departed us (may Allah bless his soul), *YES, O baba, for me you are always here to see me where I am now*. The struggle my Mother made in raising us through many thick and less thin. During my stint of PhD, not to mention, *Amma* for absorbing my hurts, worries and scars as I deposited to you. Your prayers and words made me to stick with my commitment of PhD when I wanted to quit.

Special thanks to my Dearest Sister, who was always care, supported and loved me like a mother, whom we lost when my PhD was heading towards end. The most painful spell of my life (still I am to extent), when I used to force me to concentrate on my work and has been failed terribly many times in such struggle to forgeting the fact: her no more vibrant-existence in this world.

Many thanks to my younger sister for being supportive and caring, specially sharing my responsibilities in my absence.

Words fail me to express my appreciation to my husband and my beloved

daughter Zainab, for their endless love and understanding, through the duration of my studies. Without being them with me, it was not possible to complete my studies.

I would also like to thank you to all member of Department of Computer Science, University of Leicester, UK. Finally special thanks also goes to friends and fellows for their help and valuable suggestions.

Abstract

Policies are descriptive and provide information which can be used to modify the behaviour of a system without the need of recompilation and redeployment. They are usually written in a policy definition language which allows end users to specify their requirements, preferences and constraints. Policies are used in many software application areas: network management, telecommunications, security, and access control are some typical examples. Ponder, KAoS, Rein, XACML, and WSPL are examples of policy definition languages. These languages are usually targeted at a specific domain, hence there is a plethora of languages. APPEL (the Adaptable Programmable Policy Environment Language) [69] has followed a different approach: It is a generic policy description language conceived with a clear separation between core language and its specialization for concrete domains. So far, there has not been any formal method for the extension and domain specialization of the APPEL policy language.

Policy conflict can occur when a new or a modified policy is deployed in a policy server, which leads to unspecified behaviour. To make policy based systems conflict free it is necessary to detect and resolve conflicts before they occur, otherwise the intended behaviour of a policy cannot be guaranteed.

We introduce a structural modelling approach to specialize the policy language for different domains, implemented in the VIATRA2 graph transformation tool. This approach is applied to APPEL. Our method for conflict analysis is based on the modelling methodology. As conflicts depend on domain knowledge, it is sensible to use this knowledge for conflict analysis. The identified conflicting actions are then encoded in the ALLOY model checker that confirm the existence of actual and potential conflicts.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Policies are descriptive and provide information which can be used to modify the behaviour of a system without the need of recompilation and redeployment [54]. They are usually written in a policy definition language which allows end users to specify their requirements, preferences and constraints [40]. They are common in many software application areas: electronic commerce, network management, telecommunications, security, and access control are some typical examples. Ponder, KAoS, Rein, XACML, and WSPL are examples of policy definition languages.

Policy conflicts can occur when a new or a modified policy is deployed in a policy server, which leads to unspecified behaviour. To make policy based systems conflict free it is necessary to detect and resolve conflicts before they occur, otherwise the intended behaviour of a policy cannot be guaranteed. Conflicts are said to have occurred when two policies are activated at the same time (they are triggered and their conditions are satisfied), that lead to actions that are conflicting with each other and hence, yield an undesired system state [41]. Since policies are defined by different stakeholders, including system administrators and end users, the likelihood of a policy conflict occurring is high. The exact definition of

conflicting actions depends on the domain in which the policy is used. Policy conflict is a common problem in many areas, like policy based management systems, distributed system, access and resource control, security, QoS, STPOWLA (the Service Target Policy Workflow Approach) [39] and many other application areas where policies are used. Attempts have been made to deal with policy conflict in different domains such as QoS [18, 19], security, access and resource control in distributed systems and network management environments [25, 54, 25, 35], telecommunication [58, 68, 17] and policy based management system [27, 54, 28, 53].

Many existent policy definition languages are targeted to specific domains such as networking, access control, security, QoS etc. Developing these languages from scratch represents a large investment of effort and time. Also they might not be reusable for other application domains. Reuse is essential as it saves time and effort of developers. A solution to this problem are languages that are general purpose, so they can be reused for different applications domains when required. These languages require some mechanism for domain specializations, and this mechanism needs to be identified and developed. APPEL [69] is an example of a general policy description language, but so far there has been no formal method for domain specialization.

To make effective use of policies in the systems, it is necessary to provide a policy conflict analysis and resolution mechanism. When specializing a policy language for a new domain, it will be useful to provide a method for policy conflict analysis and resolution at the same time.

1.1 Research Challenges

1.1.1 Structural Policy Language Extension Method

The first research challenge identified and addressed in this thesis is the policy language extension mechanism. Policy languages have been used for a variety of applications in software systems, and usually each application has received its own language. Policy languages such as Ponder, KAoS, Rein, XACML, and WSPL are usually targeted to a specific domain, hence there is a plethora of languages. Typically, new policy languages (or language variants) are developed in a pragmatic way when a new domain is addressed. These new languages and frameworks have some general requirements such as expressiveness, simplicity, enforceability, scalability and analyzability in addition to being domain specific [78].

Proving all these requirements in new language at once is difficult. Moreover inventing ad-hoc languages can lead to chaos. APPEL (the Adaptable Programmable Policy Environment Language) [69] has followed a different approach: APPEL is a generic policy description language conceived with a clear separation between core language and its specialization for a concrete domain. APPEL has been proven useful in many application areas such as Internet Telephony, Home Care, Sensor Networks, and Workflow adaptation. However so far no formal or structured approach is available for extension and specialization of APPEL.

Developing these languages represents a large investment in time, both of designers as well as domain experts. In the light of this, it is desirable to reuse as much as possible. There are other areas of Software Engineering where similar problems occur, that is, systems which can share a common core and require specialization. Designing domain specific languages or a system is a complex task that involves many aspects, whether functional or not. Designing generic models

and then using them for specialization is a way to achieve the solution.

1.1.2 Policy Conflict Methodology

The second research challenge is how to deal with policy conflict while specializing a policy language for a new domain. Together with the first research challenge and based on its solution, it is required to address the problem of policy conflict at the same time when a new domain is being addressed.

Methods exist to deal with policy conflict in many areas. Research on policy conflicts has been conducted in telecommunication, access control, authorization, QoS management, security, network management and many other applications. Some existing approaches for policy conflicts are based on Model Driven Development. These techniques use ontologies, information models, UML models and OCL (object constraint language) to deal with conflicts. All these methods propose good solutions to deal with policy conflicts, but these are supported by methods such as information models, ontologies, etc. Availability of domain knowledge within the model for conflict analysis is central to our approach.

1.2 Research Aims and Objectives

1.2.1 Aims

- **Develop a methodology for domain specialization of a policy language**
  As we identify a very strong link between domain and policy languages, we aim to provide a methodology based on languages and domain models to generate policy definition languages. Specifically we demonstrate the methodology for the APPEL policy language and provide a tool for the implementation of the methodology.

- **Develop an efficient and automatic conflict analysis methodology**
  This aim includes the development of suitable methods for conflict analysis for specialized policy languages based on a conflict definition for the model. We manually analyse the conflicts, with tool support to confirm conflicts.

1.2.2 Objectives

- Concepts that are already defined in the form of models, for example core concepts of the policy language or domain concepts should be reused as much as possible, both in language generation as well as conflict analysis.
- The methodology for extension of a policy language should be clear and sufficiently generic so that it can be reused for similar problems.
- The conflict analysis methodology shall correctly analyse actual and potential conflicts.

1.3 Research Methodology and Main Contributions

We have used a modelling approach in our thesis. The policy language specialization is achieved using model compositions. A parametrized composition technique is used for this purpose. The composition of models is implemented using graph transformation technology. Model checking is used for confirmation of policy conflicts. In this thesis we have used the APPEL policy language and case studies for our research. The main contributions of this thesis are:

1.3.1 Modelling of Policy Language and Domains

Specialization usually represents a large amount of effort involving domain experts. Using UML this knowledge is often captured in well-defined models. The

aid of these models is to provide as much reuse as possible. MDD (Model Driven Development) describes a method where the definition of the models and meta-models is central and underlies support techniques and tools to work with these models. In this thesis, we develop meta models of the APPEL policy language and the STPOWLA domain. We modelled the APPEL core and the domain in independent models.

1.3.2 Parametrized Composition of Models

To extend and specialize the policy language for different domains, we introduce a structural modelling approach. We use parametrization approach [61], where model composition provides a way to combine models, and model parametrization allows the reuse of models in multiple contexts. We define transformation rules using the VIATRA2 graph transformation tool for the implementation of parametrization process.

1.3.3 Policy Conflict Analysis

We manually analyse the conflicts in extended policy language (core concepts of the APPEL policy language and specialization to the STPOLWA domain). There are two types of policies in the STPOWLA domain: Reconfiguration and Refinement. Conflict definition is based on understating of the domain and thus availability of domain knowledge. The domain knowledge is already captured in the models, we use this knowledge to analyse policies. There is a natural link between the model extension and conflict analysis methods – assuming we can transfer the models to a suitable analysis tool. We decided on ALLOY as it had already been used to analyse policy conflicts in [51, 42, 43, 66, 76, 70]. Additionally, automatic transformation of UML into ALLOY code is possible using UML2Alloy tool [9],

and there are methods such as [60], providing a mechanisms for manual transformation to obtain ALLOY models from UML models. The analysed policies are encoded into ALLOY to confirm the conflicts.

1.3.4 Home Care Domain Modelling

For evaluation of our methodology we develop a UML model for the Home Care domain. By applying the structural approach, we extend the APPEL policy language for the Home Care domain. When applying the structural approach to the Home Care domain, we identify and show that this specialization needs to do more than one level of specialization when implemented in new homes. We also analysis the policy conflicts in this domain.

1.4 Overview of Approaches and Thesis

Figure 1.1 shows how an approach allows for joining of models to gain specialized languages and further how these models are taken into the conflict analysis phase. The thesis structure follows the approach closely.

Chapter 1, addresses the problem, motivation, research challenges, research methodology, solutions and list of main contributions.

Chapter 2, presents the literature review and background on Policy Languages, Model Driven Techniques, and Tools.

Chapter 3, introduces the structural modelling approach for the policy language domain specialization. The approach is then applied on APPEL policy definition language for domain specialization.

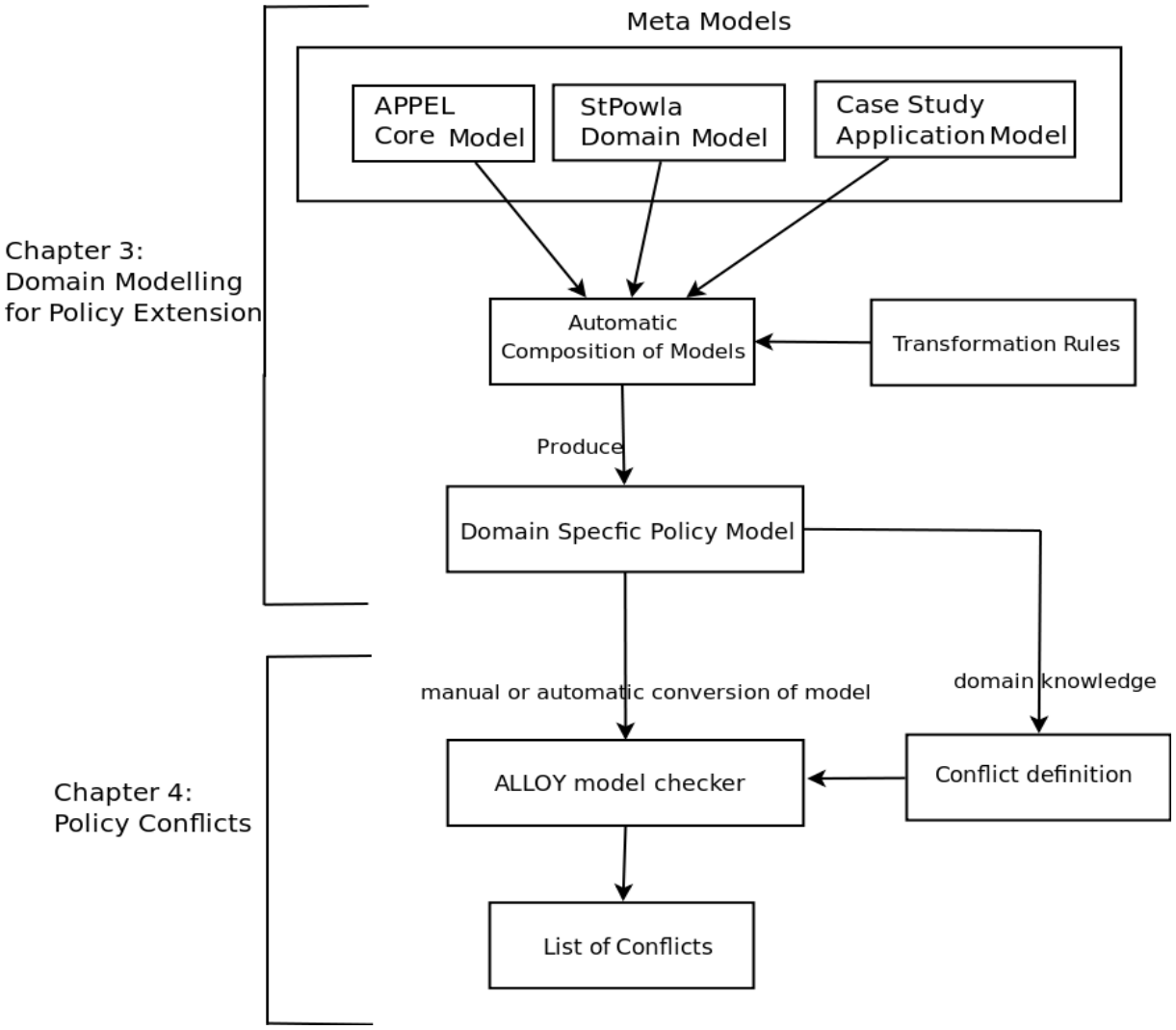


Figure 1.1: Organization of the Thesis

Chapter 4, describes the policy conflict problem in policy languages and the policy based environment. It also discusses conflict analysis, confirmation of conflicts by model checking and types of policies and conflicts in APPEL. Conflicts in APPEL and domain specializations are analysed and implemented in the ALLOY model checker.

Chapter 5, presents the evaluation and discussion of the proposed methods.

Chapter 6, presents conclusions drawn from the research and suggests areas for future work.

Chapter 2

Background and Literature Review

This chapter is comprised of three parts. In the first part we discusses the work relating to policy languages, policy conflicts and model driven techniques. Specifically we review work containing policy languages and their use in different domains, definitions of policy conflicts and existing methods of conflict detection. This highlights the intimate link between policy languages and the domain they are intended for. Overall it indicates requirements which makes policy languages suitable for domain specialization, conflict detection and brings out the need for a structured approaches to extension of PDLs. As model driven techniques have been used successfully in other areas where domain expertise is essential, we adopt them here. Specifically, we review existing work in model composition and transformation.

The second part provides a deep background on APPEL and STPOWLA, the specific PDLs considered in this work. We also provide a background for UML and graph transformations. We make use of model transformation and also verification tools, and use this section to discuss them.

In the final part we draw conclusions from the literature review and background.

2.1 Literature Review

There are several areas of existing work which have a bearing on this thesis, and they will be reviewed here. Firstly, the background work discusses various policy languages, their structure and use in different domains. The second part discusses policy conflicts, their types, conflict detection and resolution methods. The third part discusses Model Driven Development (MDD), model composition and model transformation techniques used in this work.

2.1.1 Existing Policy Languages

A policy is defined in a particular policy description language which specifies the syntax and semantics of a policy. This language specifies aspect such as the structure of the policy and the format of its rules. A policy system is a framework through which policies are stored, retrieved and executed. This policy system interacts with a human policy creator and an underlying controlling system.

Different policy languages have been developed for different domains, hence a number of policy systems and languages are in use. For example policies are used in the domain of access control, distributed systems, security, telecommunication. Every language has its own syntax, structure, semantics and implementation mechanisms. "There is no standard metric available that can be used to analyze and compare these languages" [50]. Most work deals with declarative policies. Examples of policy languages are eXtensible Access Control Markup Language (XACML) [37], Ponder [21], Web Service Policy Language (WSPL) [10], Rein [45] and APPEL [69]. These languages have much in common, but each address a specific domain and has been devised completely from scratch. These policy languages are briefly discuss in this section.

Before discussing individual policy languages, we need to introduce some ba-

sic terminology that is commonly used in almost every policy language.

Policy: A Policy is a main element of a policy system that is used to define and manage the behaviour of a system.

Policy Rule: A Policy Rule is a container that contains an optional Trigger, an optional Condition and an Action.

Policy Group A Policy Group is a container that contain one or more Policy Rule.

Trigger: A Trigger represents an occurrence of a specific event in the system or in its environment.

Condition: A condition is represented as a Boolean expression and defines the necessary state of the system and its environment where the policy is applicable.

Action: An action represents the necessary actions that should be performed if the policy is applicable.

Subject: A subject is a set of entities that is the focus of the policy. The subject can make policy decision and information requests, and it can direct policies to be enforced at a set of targets.

Target: A target is a set of entities that a set of policies will be applied to. The objective of applying a policy is to either maintain the current state of the target or to transition the target to a new state.

XACML

The eXtensible Access Control Markup Language (XACML) [37] is a declarative policy language for access control implemented in XML [15]. Access con-

trol policies define how services in a computer environment may be used. The XACML policy structure consist of subject, target, action and condition. The subject is used to identify group or role. The target identifies the set of requests. XACML has three policy elements: `Rule, Policy and PolicySet`. A `Policy` is a set of `Rule` elements and a `PolicySet` contains various `Policy` elements. As XACML is focused on access control, it is suitable for its purpose, and is not constructed in a way that tends itself to being extended for other domains, making it less suitable for our purpose than APPEL.

Ponder

Ponder [21] is a declarative, object-oriented policy specification language for access control and system configuration. It is also used for specifying management and security policies for distributed systems activities such as registration of users or logging and auditing events for dealing with access to critical resources or security violations. The language has three types of policies: basic policies, composite policies and meta-policies. Basic policies include the following policy types: *Authorization* policies that define permitted actions; event-triggered *Obligation* policies that define actions to be performed by manager agents; *Refrain* policies that define actions that subjects must refrain from performing; and *Delegation* policies that define what authorizations can be delegated and to whom.

A basic policy in Ponder consist of one or more policy elements. These elements might be common to all basic policy types: the subject, the target, the when-constraint, import statements, constant definitions and external specifications. Other policy elements are specific to a particular policy type. Policy elements can be specified in any order.

Extending the Ponder language to cater for new kinds of policies is simplified using an underlying object-oriented implementation. Ponder can be extended by

adding new base sub-classes to the existing ones, or by adding new attributes to existing base classes. However such extensions are at a very technical level and thus conceptually quite remote from the domain. APPEL is less complex as compared to Ponder as APPEL offers fewer types of policies (two verses three in Ponder). Also Ponder's principal application areas have been security policy creation and management, however the framework is designed to support system management in general, while APPEL was designed to be generic and did not evolve from the domain of security or access control.

WSPL

The Web Services Policy Language (WSPL) is used for authorization, quality-of-service, reliable messaging, privacy, quality-of protection, and application-specific service options [10]. The syntax of WSPL is a strict subset of the XACML Standard. A WSPL policy consists of one or more rules. Rules are listed in order of preference, with the most preferred choice listed first. A WSPL rule is a sequence of predicates. Each predicate places a constraint on the value of an attribute. Each policy also states the target aspect of the web service that is covered by that policy. WSPL has been implemented, and is under consideration as a standard policy language for use with web services.

2.1.2 Policy Conflicts

Policy conflict is a very common problem occurring in all areas where policies are used. Due to the use of policies in various application areas the exact definition of what constitutes a conflict is depended on the application domain. In general *"two policies are said to conflict with each other if there is inconsistency between them"* [32]. While this definition provides the general idea of a conflict, in other definitions define policy conflict more precise way, e.g.: *"A policy conflict occurs*

when two or more policies contradict each other in terms of what the system is instructed to do or what state it should maintain" [41].

While these definitions capture the idea of policy conflict, it is only possible to make a clear statement whether two policies conflict when understanding what it means for two actions to conflict in the domain. Additionally there might be types of conflict that exist within the policies, independent of the application domain.

As policies are defined by end users, the problem of policy conflict is increased substantially. Attempts have been made to deal with policy conflict in different domains, QoS [18, 19], pervasive computing [76, 77, 26, 71], security, access and resource control [25, 54, 25, 35, 76], telecommunication [58, 68, 17], policy based management system [27, 54, 28, 53] and homecare [32]. We will consider the results of this body of work next, but apply a conceptual structure bridging domains.

Policy Analysis Methodologies

It has been realized that research on policy conflicts can be categorized in different ways: by methods, by approaches and by domain dependence. There is an extensive literature on policy analysis, policy conflicts, conflict detection methods and conflict resolution. We have put the relevant work in appropriate categories. These approaches and related work are discussed below:

By Methods

Policy conflict analysis methods can be categorized by the time when the detection method is applied. The research in [68] provides two main types of methods for conflicts detection and resolution.

1. **Offline Methods**

   Offline methods are also known as static methods to detect conflicts at

design and specification time. They typically use some formal model or pragmatic technique to resolve conflicts by redesign and are useful at the time of policy deployment. Static conflict detection methods can also be used for run time conflict detection, made possible by using a snap shot of a system created at a certain moment and detect the conflict at that moment [70]. Static policy conflict detection methods have the advantage that the solution of the conflict is known before it happens [71].

The work presented in [75] defined a correspondence between APPEL [69] policies and UML state machines and uses UMC [1] as a model checker to verify that policies expressed in UML are free of conflicts. The study uses the model checker UMC and the associated action- and state-based branching-time temporal logic UCTL [74]. The approach is demonstrated by a case study. For their work, they modelled APPEL policies in UML explicitly, whereas in our work we utilize the available knowledge (APPEL, domain, case studies) in models to provide model based conflict analysis.

A logic based approach to deal with policy conflicts in APPEL is presented in [58, 59]. The research presents the formal semantics for APPEL. These semantics are based on $\Delta DSTL(x)$ (distributed state temporal logic). The policies and conflicts are defined by liveness formulae in a distributed temporal logic, and detected by (semi-)automatic theorem proving: a conflict is found if it is possible to derive, from the logical presentation of the policies, a formula stating that a conflict will arise. Based on these formal semantics they proposed a technique to reason about conflicts. The research represented in [59] is an extension of [58], which adds the formal semantics for distributed policies in APPEL, and techniques to deal with conflicts and their resolution. This work conducts conflicts in APPEL, but does not discuss how domain specific conflicts are detected. Also they used a theorem

proving approach for conflict detection, which is semi-automatic. We used model checking, which is automatic.

[46] presents a UML approach, based on existing knowledge of a managed system. They argue that policy conflicts remain undetectable using conventional (language based) approaches because the implicit knowledge of a system is unavailable to policy analysis. The study represents the implicit knowledge of managed systems explicitly in UML. Conflict definitions are derived from existing models using OCL invariants, constraints, pre and post conditions and applied on the abstract levels of models where the size of the model is small or moderate. Invariants and conflict definitions are defined manually. This approach represents the use of UML and OCL for conflict detection, however it is limited as it is applied on abstract and small models. Our conflict analysis method is based on our extension mechanism, so all knowledge of a policy language together with the domain knowledge is already captured in the models and we use that knowledge for conflict analysis. We work with complete models and concrete polices to analyse and confirm potential and actual conflicts.

An access control model based on a role based approach is presented in [76]. The approach has been proposed for a pervasive computing application, where the application needs contextual information. Their model uses environmental contexts (time and location) to determine whether a user is permitted to access some resource or not. These models have many features because of their application requirements, and these features can be in conflict with each other. An automated approach using ALLOY is used to detect these conflicts. They claim that the results obtained form the analysis would enable the users of the model to make an informed decision.

2. **Online Methods**

   Online or dynamic methods detect conflicts at run-time. The run-time detection methods rely on information provided by the system at run-time. These methods use some default strategy to deal with conflicts. Our methodology is an offline conflict detection method, but there are some online approaches that have bearing on our work uses APPEL for conflict detection.

   The research in [32] focused on conflict issues when using policy-based management in home care systems. They make use of the APPEL policy language for the home care system. Three types of conflicts in policy-based home care systems are identified: conflicts that result from apparently separate triggers, conflicts among policies of multiple stake holders, and conflicts resulting from apparently unrelated actions. They systematically analyse the types of policy conflicts, but do not demonstrate how they detect conflicts. However they emphasise a resolution strategy to resolve the conflicts. While we also analyse the conflicts in our approach, we additionally encode this analysis in the ALLOY model checker to confirm a potential and actual conflict.

By approaches The author in [22] identified that policy conflict analysis can be categorized into three approaches:

1. **Language Based Policy Conflict Analysis**

   This type of analysis is carried out by testing the policy language constructs to find conflicts. This approach is used for network filtering policies where language constructs and their relationships are semantically well defined [22].

2. **Information Model Based Policy Conflict Analysis**

   This method is based on the underlying information model, which is a struc-

tured representation of the entities, concepts and their relationships of a problem domain or managed system. The policies are applied on entities, these policies are separate from the information model [23].

A conflict prevention approach is presented in [23]. This methodology is based on a two phase algorithm. This algorithm obtains application specific data, i.e, constraints and relationships from an information model, that are subsequently used for analysis of policies conflicts in the second phase of algorithm. Conflicting policies are then modified and restored. The analysis in [24] is an extension of the above work. The method is the same but the approach is different. It again demonstrates the use of a two phase algorithm to determine conflicts. The authors argue that the algorithm is generic and independent of application specific knowledge in the sense that application specific knowledge of a problem domain is represented in the Information model. Their main contribution is the two phase algorithm which queries the information model. The first phase of the algorithm analyse the relationship between policies, and the second phase uses application specific conflict patterns in the form of matrix, to identify the conflicts. The approach takes into consideration only detection of a potential conflict, and does not suggest any resolution strategies. Their approach is limited in the sense that it can only be used where domain and application specific information is represented in an Information model. From this it becomes clear that capturing the domain and application information is crucial for conflict analysis - in our work we make such information the fundamental basis.

3. **Ontology Based Policy Conflict Analysis**

   This is a comparatively new method, where policy models are constructed using ontologies and the inherent reasoning opportunity of ontologies, sup-

ports policy analysis.

[17] present an approach that is based on the ontology driven method. This method examines policy conflicts of the APPEL policy language for the domain of call control. The authors argue that this method automatically identifies potential conflicts among policies. Their method examines domain knowledge to interpret policy action with their effects. Policy conflict detection method are conducted offline, but the method also supports conflict detection and resolution online. The method is supported by the RECAP tool (rigorously evaluated conflict among policies), which filters conflicting pair of actions and automatically generates resolutions. The detection of conflicts is completely automated and resolution is partly automated by RECAP. The details of resolution involve human judgement and are added in a manual step. Our work is related with this study in that it also makes use of the APPEL policy language, but the targeted domain and the conflict analysis methodology are different. Their conflict detection and resolution strategy is ontology based, whereas our methodology is based on modelling.

By Domain Dependence [29] classified policy conflicts into two groups: Modality and Semantic conflicts.

1. Modality Conflicts: Modality conflicts [57] are inconsistencies in the policy specifications, due to the existence of both a negative and a positive obligation or authorization policy that apply to the same set of managed objects. Modalities are independent of identities of managed objects, so modality conflicts can be detected without the knowledge of the managed system by syntactic analysis of the policies prior to deployment [29].

2. Semantic Conflicts: Semantic conflicts are existent in the system due to inconsistencies between policies. These can be detected by analysing sub-

jects, targets and actions of the policies together with the knowledge of external criteria such as limited resources, topology of the system, or overall policies of the system. These conflicts in policies cannot be determined directly from the syntactic analysis of the policy specifications without the description of the conflicts [29].

[29] presents a policy-conflict handling model, that is independent of both technical details of the policy enforcement architecture and syntax of policy-specification language. It uses a meta-policy based conflict handling mechanism, which defines a constraint about permitted policies. The model primarily focuses on handling the application-specific (semantic) conflicts. The model is applied to the POLICE Policy-Based Network Management (PBNM) framework. They argue that the POLICE framework does not suffer from modality conflicts, because it does not support negative policies. Application-specific (semantic) conflicts are handled according to the proposed model. They rely on a common schema in the model and argue that almost all policy specification languages include subject, target and action concepts and the fact that all languages can be modelled in similar way allows their model to be designed independently of other details of the language. What constitutes a conflict is dependent upon a particular application domain. The analysis of a conflict is then carried out according to the conflict definition. This is the drawback of their approach, as they did not discuss the conflict definition issue.

The comparison of policy conflicts approaches that are discussed above are summarized, and presented in Table 2.1.

Research	Approach	Methodology	Automation	Method
[75]	Model based (UML state machines)	Model checking (UMC model checker)	Automatic	Offline
[58, 59]	Logic based approach	Theorem proving	Semi-automatic	Offline
[46]	UML based approach	OCL invariants	Automatic	Offline
[76]	Model based	Model checking (ALLOY model checker)	Automatic	Offline
[23, 24]	Information Model based	Two phase algorithm	Automatic	Offline
[17]	Ontology Driven	RECAP tool	Automatic	Online
[57]	Syntactic Analysis	Modalities	Semi-automatic	Offline
[29]	Model based	Model Based	Automatic	Offline
Our approach	Model based	Model checking (ALLOY model checker)	Automatic	Offline

Table 2.1: Comparison of Policy Conflict Approaches

2.1.3 Model Driven Architecture (MDA)

Model-driven architecture (MDA) [2] is a software design approach for the development of software systems. It provides a set of guidelines for the structuring of models and their specifications. MDA defines standards and terminology for application design and implementation. It is a kind of domain engineering, and supports Model Driven Engineering (MDE) of software systems.

Metamodel

Metamodel denotes the definition of models for models. A metamodel is simply a model that defines the structure and constraints for a family of models (instances). A model is a way to specify abstract syntax and semantics of a problem domain. UML itself is defined in terms of a metamodel, called MOF (Meta Object Facility) [63]. This concept explained in Section 2.2.2.

Model Composition

Models are continuously gaining importance in the software development life cycle [2]. These models can be used as concrete artifacts [61] in operations like construction and transformation. Construction techniques allow to produce a new model from existing ones. Composition is a construction technique which permits to build a model from a set of smaller ones. Model composition is a process of merging two or more models to obtain a single model. In contrast, transformation is a process of transforming one or more (input) models into another (output) model. "In model composition, the new features that are to be incorporated into a model are explicitly described by one or more source models; in a transformation, the new features are implicitly defined in the transformation actions that are carried out on the input model" [12].

Metamodel composition is necessary for several reasons. A metamodel provides a modelling paradigm that is set of axioms, notions, idioms, abstractions, and techniques that specify how domains are to be modelled. Metamodels represent a large investment of efforts and time in the understanding of a particular engineering domain. Hence is is useful to reuse previously defined domain knowledge when constructing a new metamodel. This can be done by composing specific metamodels from abstract metamodels (i.e. metamodels that are not significant itself individually, but capture some general modelling constructs that are useful

when combined with others). This approach is only possible as part of the composition process, where domain-specific concepts and constraints can be added to the resultant metamodel. Such a compositional approach to metamodel specification and construction has the advantage of reusing the existing concepts and increasing the quality and functionality of the metamodel.

Parametrization

Parametrization is a composition technique that can be defined as an "act that replaces an existing element in a model by another compatible one" [61]. Model parametrization allows existing models to be reused in different contexts, thus significantly decreasing the effort on modelling. A metamodel can be enriched by means of another metamodel through Parametrization [61]. In this work, we have used parametrization for specialization of a policy language where the core language model specialized for domains (that are modelled separately).

Model Driven Development

In this section, model composition and transformation approaches from the literature are reviewed.

The EML (Epsilon Merging Language) [49] is a rule based language for merging models. EML is used for model management tasks such as model comparison, model transformation, model validation, etc. It uses three categories of rules: MatchRule, MergeRule and TransformRule [49]. The identified matched elements are merged into a sequence of model elements in the target model and a selection of the elements for which a match has not been found in the opposite model are transformed into elements of the target model. Our approach is different from the general merge mechanism provided by EML. In a EML merge operation, two or more input models are merged together and all other entities

of both models are added to the output (resulting model). Merge in our case is different, in that it is a parametrized merging, where two or more source (input) models are merged into one target (output) model by replacing elements in one input model by those in the other input models.

The work presented in [12] discusses some similarities between model composition and model transformation. The research analyzes a number of approaches to implement composition as transformations. The comparison of different approaches is based on generality, ease of use and ease of implementation. The work explored the possibility of composing a set of models based on cross-cutting concerns (aspects), with a primary base model (which represent the core functionality of an application). The model composition approach presented in [33] is the continuity of the work in [12] and offers a generic framework which is independent of any modeling language. The approach presents a generic metamodel, describing structural and behavioural features of a composition operator. This metamodel supports the composition directives concept presented in [67]. These composition directives are supported by the Kermeta language [62]. The approach is imperative because it describes the operation of composition (merge operation) in an algorithmic way. The imperative approach is not easily compatible with a declarative one, as the declarative approach specifies what should be transformed rather than how it should be done.

The metamodel merge method represented in [31] composes two modeling languages. The constructs of the two languages share a set of real world entities; those concepts are used as join points to stitch the two languages together into a unified whole. In MOF [7], terms this operation is a package merge because it operates at the package level and impacts all of the elements contained within the merged packages. Package merge is intended to allow concepts defined in one package to be extended with features defined in another [88]. “A package merge

is a directed relationship between two packages that indicates that the contents of the two packages are to be combined. This mechanism should be used when elements defined in different packages have the same name and are intended to represent the same concept. Most often it is used to provide different definitions of a given concept for different purposes, starting from a common base definition" [81]. Again, it is a general merging mechanism that operates on package level, and our parametrized merge approach is different from this merge, where the merging operation is applied on models (class diagrams) rather then the packages level.

Implementation inheritance and interface inheritance metamodel composition operators are proposed by the Meta GME (Generic Modeling Environment) [52]. Using the implementation inheritance operator, the children (sub class) inherit all the parent (super/base class) attributes, but only the containment associations where the parent acts as the container. The interface inheritance operator allows no attribute inheritance, but allows all the associations (with the exception of the containment relations) to be inherited. This technique is used when two metamodels capture conceptually different but related domains. This method uses generalization (sub-typing), whereas we have used the concept of parametrization, where original models are used for specializations.

The conceptual framework and methodology presented in [65] allows the creation of DSMLs (domain specific modelling languages) for prototyping and verification. The study introduces different model composition mechanisms such as parametrization, merge, union, inheritance, implementation inheritance and their transformation. The research claims that it provides syntactic and semantic composition of concepts allowing to define a specific DSML behaviour by starting with a more abstract view of the language and then by particularizing some of its concepts to fit a more precise semantics. The approach selects CO-OPN (Concurrent Object-Oriented Petri- Nets) [16] as a target language for implementation of

proposed compositions. According to the parametrization composition requirements for our work, we have defined transformation rules. For implementation and other requirements (related to our work), we have used VIATRA2 graph transformation tool.

[85] introduces the concepts of a generic policy model where specific policies languages are created as simple extensions of a generic policy model. They model the core concepts of the language, but did not discuss how their approach can be used for policy language extension or specialization. Our approach is applied to a specific policy language (albeit it will be transferable to other such languages) and focuses on providing a structured approach for tailoring this to specific domains and applications.

Composition of parametrized models, their needs and use in MDE in discussed in [61]. The study provides an approach of parametrized model application which allows parametrized merging of two models to obtain an extended model. The approach is formalized with an operator named *apply*. The approach is imperative rather than declarative, specifying how to merge two models. A tool (cocoa modeler)[1] is implemented for this approach, however the tool does not support aggregation and generalization relationships of a model, which is crucial for our work.

2.2 Background

2.2.1 Graph Transformation

This section aims to introduce the formalisms and languages of graph transformation used for modelling and extension of a policy language. However, it is important to mention that we cannot cover all aspects of graph transformation theory, but

[1] http://www.lifl.fr/mullera/cocoamodeler

we will selectively explain only those features which are relevant to the work in this thesis.

Graphs and Graph Schemas

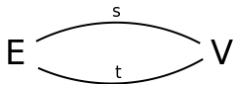
Generally graph transformation systems provide an intuitive description for the manipulation of graphs and graph-based structures as they occur in representations of programming language semantics, data bases, object-oriented systems, and various kinds of software and distributed systems. Due to their formal, operational semantics, these descriptions can be analyzed and executed using suitable graph transformation tools.

The literature on how to define graph transformations and their semantics is split into two different mathematical techniques: the *set theoretic approach* and the *algebraic approach*. The basic difference is the way how the local effect of a graph transformation rule is embedded into the original host graph. We adopt the use of the *algebraic approach* for this work. The algebraic approach was proposed by Ehrig, Pfender, and Schneider in the early seventies in order to generalize Chomsky grammars form string to graphs [30]. The gluing of graphs plays a central role in the *algebraic approach* to graph transformation, where a subgraph, rather than a node or an edge only can be replaced by a new subgraph. The approach was called *algebraic* beacuse graphs are considered as special kind of algebras and the gluing for graphs is defined by an *algebraic construction*, called *pushout*, in the category of graphs and total graph morphisms. Originally rewriting of graphs based on gluing has been formulated by the so called *DPO (Double PushOut)* that uses two gluing constructions (i.e. pushouts) to model a graph transformation step. The other approach using a single step for modelling the transformation step is know as *SPO (Single PushOut)*.

Graphs: A graph consists of nodes and edges, where an edge connects two nodes.

Graphs can be directed or undirected. In directed graphs each edge has a specific start (source) and end (target) node.

Definition (Graph:) [30] *A graph $G = (V, E, s, t)$ consist of a set V of all nodes, a set E of edges and two function $s, t : E \longrightarrow V$, the source and target functions.*



In the literature e.g. [30], a graph is G is often represented by a set of nodes V and a set $I \subseteq V$ X V of edges. This notation is the similar to the one presented above: for an element $(v, w) \in E$, v represents its source and w is target node, but the parallel edges are not expressible. To model undirected graphs, for each respective edge between two nodes v and w, we add both directed edges (v, w) and (w, v).

Type Graph and Typing Morphism

Schemas are used to restrict the set of allowed graphs in declarative ways, similar to class diagrams in object-oriented models or entity relationship diagrams in data models. This means to restrict the shape of an object is to prescribe a certain type for the object. In the following we will introduce the notion of typed graphs. A type graph TG is a graph whose nodes represent node types and whose edges represent edge types. A graph that is typed over a type graph TG, also called *instance graph* over TG, is a graph G equipped with a graph morphism $type_G$: G $\longrightarrow TG$ that assigns a type to every node and edge in G. The type and typed graph have the same relationship as metamodel and model have. An edge type in TG represents a structural relationship among nodes of TG-typed graphs. This is because, due to the typing morphism, edges of an edge type may only connect nodes of the node types that are incident to the edge type in TG.

A model can be naturally represented as a graph based structure. Model transformation problems can be relatively easily formulated as graph transformation problems [73]. A type graph represents metamodels, and graphs represent models. A model will be well-formed if its graph conforms to the type graph [55]. This is visualized in Figure 2.1.

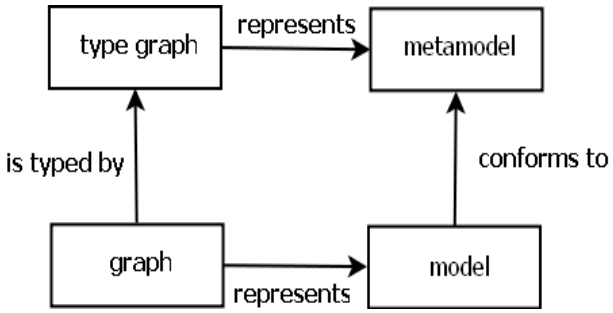


Figure 2.1: Relationship of Model and Graph

A node type can be compared to a class and an edge type can be compared to an association in a UML class diagram (except that in type graphs directed edges are used). Thus, a type graph can also be presented by a UML class diagram as shown in Figure 2.2. Similarly, instance graphs can be represented by corresponding UML object diagrams as shown in Figure 2.3. As per UML syntax, each node is labelled by an identifier followed by a reference to its type in the type graph. Whereas, edge labels do not contain any identifier but refer to the respective edge type only.

Attributes and Typed Attributed Graphs

In graphs, attributes are used to store additional information in a node, so nodes are enriched with attributes. As in object-oriented languages, an attribute has a unique name and data value. In the context of a type graph, there is a need to declare the attributes that belong to a certain node, each has a name followed by

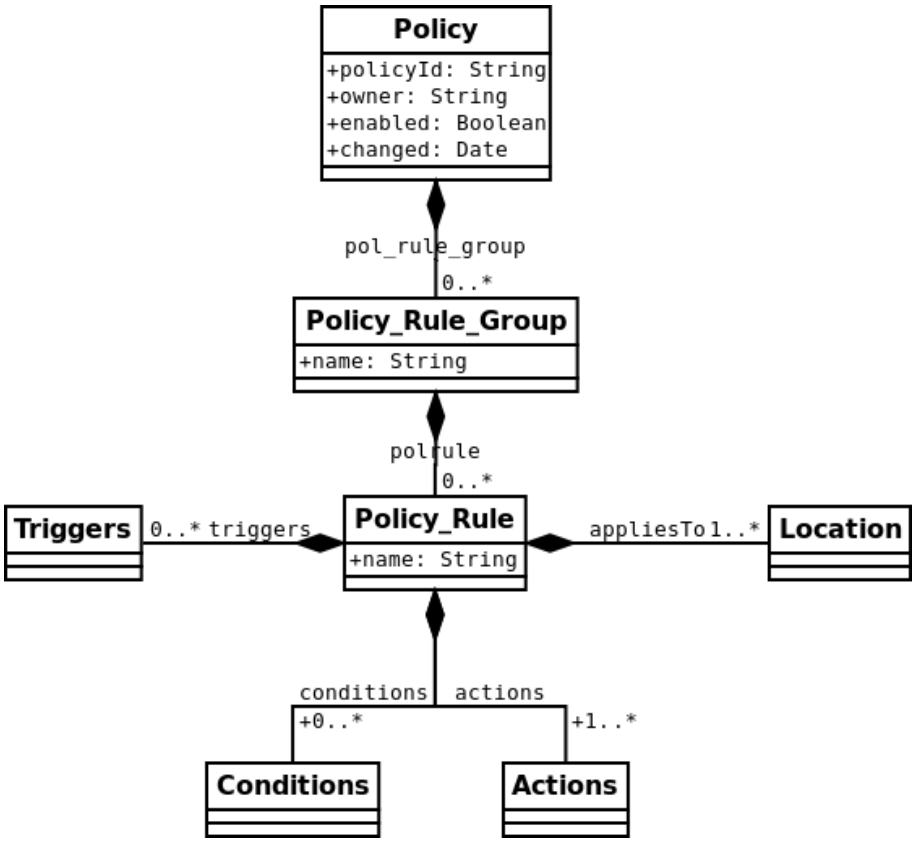


Figure 2.2: Type Graph as UML Class Diagram

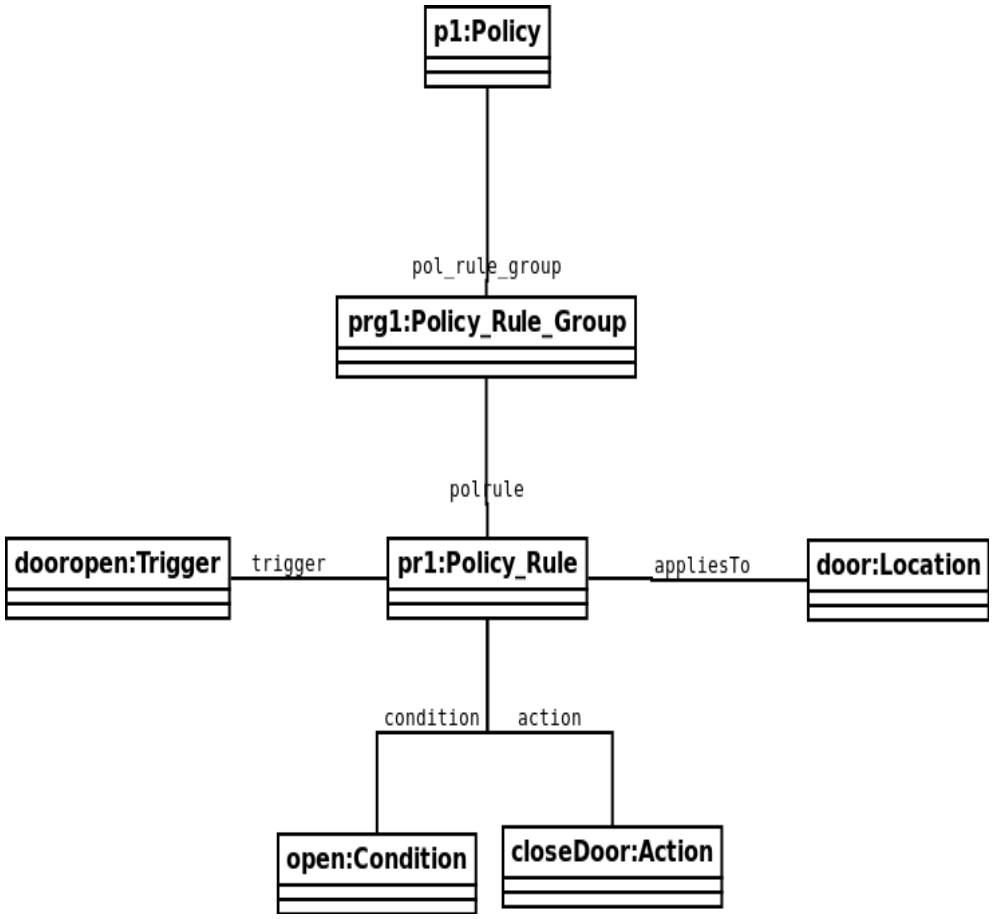


Figure 2.3: Instance Graph as UML object Diagram

its data type. So each node in the instance graph can store different values in the attribute of the same data type.

Graph Schemas

A type graph with constraints is called a graph schema, and resembles a class diagram with constraints or an ER-diagram with cardinalities.

Graph Transformations System

The graph schema provides the set of valid instance graphs and graph transformation rules are used to define the transformation of instance graphs.

Graph Transformation Rules

Graph transformation rules consist of a left-hand side and a right-hand side each. The left-hand side represents the pre-conditions of the rule, and the right-hand side represents its effects as post-conditions. The left-hand side and the right-hand side are instance graphs L and R.

A rule can be applied to a concrete instance graph G, also called a host graph, whenever there is an occurrence of the left-hand side L in G. In this context, occurrence means a sub-graph of G which has the same structure as L and whose elements conform to the typing and attribute values of L. Sometimes, such an occurrence is also called a match of L in G. If an occurrence of L has been found in G, the rule can be applied as follows: The first step is to remove those elements from the occurrence that do not appear in the right-hand side R of the rule. Then, a certain embedding mechanism is used to merge the remaining graph D with an isomorphic copy of R.

Example

A simple example of a graph rule is shown in Figure 2.4. In the LHS, the existence of three class instances *(p, dooropen, doorlock)* are required as pre-condition of the rule. This rule requires the deletion of an instance *dooropen* and addition of an instance called *room* on the RHS.

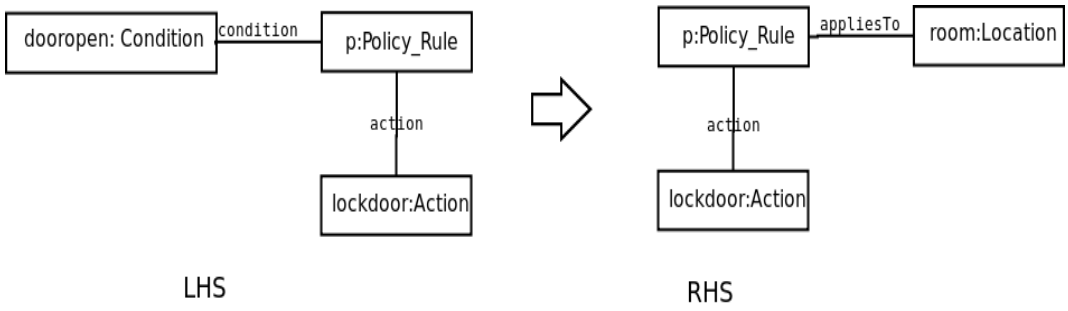


Figure 2.4: Graph Transformation Rule

Double Pushout (DPO) Approach

In agebric approaches direct derivations are modelled by gluing construction of graphs. These graphs are formally characterized as *pusouts* in suitable categories having graphs as objects, (and total or partial) graph homomorphism as arrows. A production p in the DPO approach is given by a pair of $p = (L \leftarrow K \rightarrow R)$ of graph homomorphisms form a common *interface graph* **K**, and direct derivation is consist of a two gluing diagram of graphs and total graph morphism shown in Figure 2.5 (1) and (2). A *context graph D* is obtain form given graph G by deleting all elements of G which have pre image in L, but none in K. In DPO approach the match m must satisfy an application condition, called the *gluing condition*. This condition is consist of two parts.

dangling condition: To ensure that D will have no dangling edges, the *dangling*

condition requires that if application of p specifies the deletion of a vertex of G, then it must specify also the deletion of all edges of G incident to that node.

identification condition: requires that every element of G that should be deleted by the application of p has only one pre-image in L. The gluing condition ensure that the application of p to G deletes exactly what is specified by production.

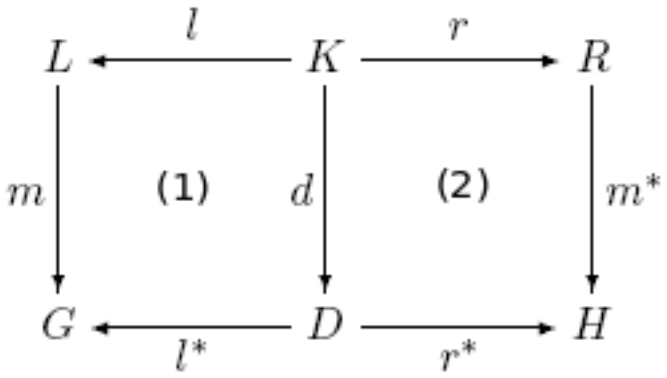


Figure 2.5: DPO graph transformation

2.2.2 Unified Modelling Language (UML)

UML is used for modelling in this thesis and we will discuss basic concepts of UML here. The Meta Object Facility (MOF) [63] is an object-oriented modelling language, its meta-language defines the foundational concepts required to build the Unified Modelling Language (UML) [64], so the model elements in UML are instantiates of model element defined in MOF.

UML is a graphical language that defines a notation and a metamodel. The metamodel of UML defines the concepts of the language through its abstract syn-

tax. UML is a standard modelling language controlled by the Object Management Group (OMG). UML uses various visual notations to create models for object-oriented modelling such as class diagrams, activity diagrams, use case diagrams etc. UML diagrams are used for two types of modelling: *static* and *dynamic*. The static diagrams are used for structural aspect of modelling, whereas the dynamic diagrams are used for behavioural modelling. Class diagrams and object diagrams are used for structural modelling, while activity and use case diagram are used for dynamic modelling. We have used UML for structural modelling in this work.

A *class diagram* describes the types of objects in a system and their relationships [34]. A class is composed of two features: the *properties* and *operations*. The *properties* represent the structure of the class and are shown as *attributes* and *associations* in the class diagram. An attribute is defined in the first part of the class with the name and its type, while associations are represented as a solid line between two classes, directed from the source to target or bidirectional. An *association* has a name and multiplicity. The *multiplicity* of a property shows how many objects may fill the property. *Operations* are the actions that a class may carry out. They are defined in the second part of the class in the class diagram. Figure 2.6 shows the basic structure and concepts of a class diagram.

Generalization defines inheritance between classes. In the Figure 2.6, the `Customer` class is called super class (supertype), while the `PersonalCustomer` and `CorporateCustomer` are called sub class (subtype).

Aggregation and composition are two specialized forms of association between two classes. Aggregation is a *"part-of"* relationship, where all child classes are owned by the parent class (the child class can not be owned by more than one parent class, and the child class has its own life cycle). Composition represents a *"has-a"* relationship, and is a strong type of aggregation, where a child class does not have its own life cycle; if the parent class is terminated, all its child classes

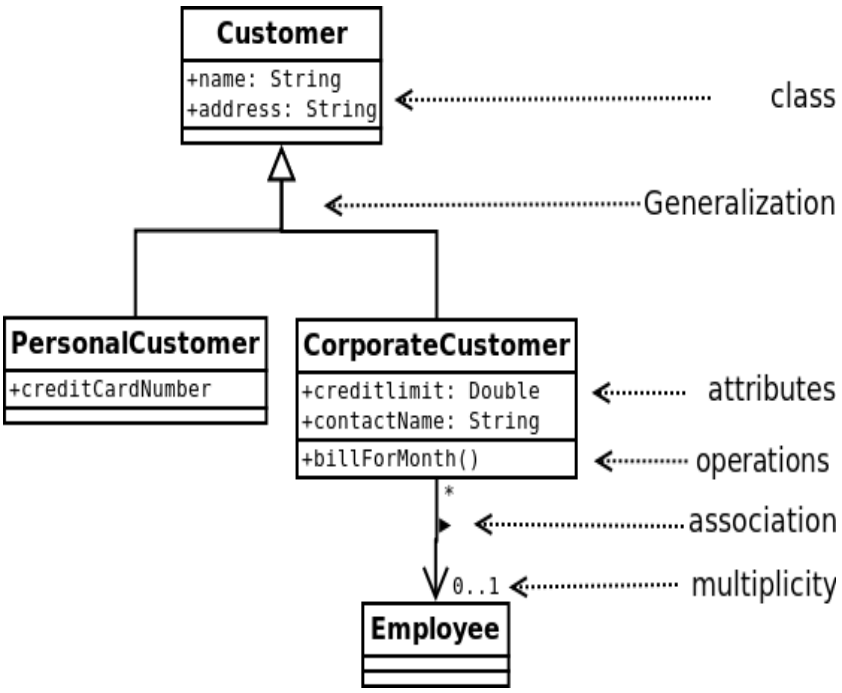
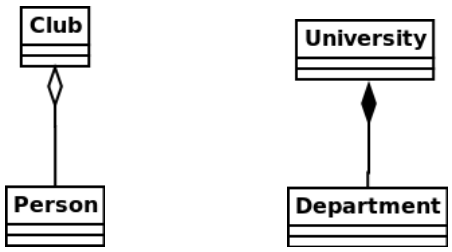


Figure 2.6: Basic Class Diagram

are deleted too. Figure 2.7(a) shows the aggregation relationship where `Club` own `Person` as a member of the club. Figure 2.7(b) shows the composition relation where a `University` has a `Department`, and if the university is terminated, all departments will be deleted as well.



(a) (b)

Figure 2.7: Aggregation (a) Vs Composition (b)

2.2.3 APPEL

APPEL [69] is a Policy Description Language (PDL) originally developed to provide a practical and comprehensive policy language for call control. It is a general language for expressing policies in variety of application domains. It is conceived with a clear separation between core language and specialization for concrete domains. APPEL is designed for end users; its style is close to natural language permitting ordinary user to formulate and understand policies.

The APPEL policy language, has been designed with extensibility and domain customization in mind. It has a well defined syntax and a formal semantics, however domain specialisations have been rather ad-hoc. APPEL has an XML syntax, and a more succinct BNF representation.

policy	::=	[location] pol_rule_group | pol_rule_group policy
pol_rule_group	::=	[location] polrule | pol_rule_group op pol_rule_group
op	::=	**g**(conditions) | **u** | **par** | **seq**
polrule	::=	[triggers] [conditions] actions
triggers	::=	[location] trigger | triggers **or** triggers
conditions	::=	[location] condition | **not** conditions |
		conditions **or** conditions | conditions **and** conditions
actions	::=	[location] action | actions actionop actions
actionop	::=	**and** | **or** | **andthen** | **orelse**
location	::=	constantLocation | VariableLocation

The core language defines the structure of the language that is the basic constructs we would expect in all policies such as `Triggers`, `Conditions` and `Actions` and their relationships. The details are left for later definition as required for a particular domain.

A policy in APPEL is a main element. A *policy* is composed of a number of *policy rule groups*. A *policy rule* is either a single *policy rule* or a composition of *policy rules*. Policy rules can optionally be combined in pairs with a number of operators (g:guarded, u:unguarded, par:parallel, seq:sequential). Each policy rule consists of *triggers*, *conditions* and *actions*, where triggers and conditions are optional. A trigger group is either a single trigger or a composition of two or more triggers. A condition group is either a single condition or a composition of one or more conditions. An action group is either a single action or a composition of two or more actions. A policy rule is applicable if its trigger occurred and its condition is satisfied (if no trigger or condition is specified they are considered to be trivially true). Triggers, conditions and actions occur in simple forms, but there are also complex versions composed by using specific operators (and, or, andthen and orelse) out of the simple forms. In addition policies are assigned a *location*, the exact semantics of this depends on the domain (in telecommunications it is

typically a user or domain name, for workflow a task forms a suitable location).

APPEL in its original incarnation deals with two types of policies: regular policies and resolution policies. These policies are discussed below:

Regular Policies

Regular policies are used to define new policies for the system that is being controlled. APPEL has been used for call control, but also for a number of other domains, including sensor networks, homecare [79] and service oriented architecture (SoA)[39], with each domain define its own regular policies.

Resolution Policies

The resolution policies have similar structure to regular policies. A main difference is that, the resolution policies are triggered by the conflicting actions. The triggers of the resolution policies are therefore the actions of the regular policies. The resolution policies are used to deal with policy conflicts.

2.2.4 StPowla

The Service Target Policy Workflow Approach STPOWLA [39] addresses the integration of business processes, policies and SoA at a high level of abstraction. It captures essential requirements at a business level in the form of workflows and the variability in terms of policies that are expressed in a language close to the business goals. There are two types of regular policies which have been defined for this domain of business workflows: refinement and reconfiguration policies. The syntax of the ad-hoc extension to APPEL used in STPOWLA policies is:

Policy <name>

appliesto <taskName>

when taskEntry

if <condition>

do req $(main, < args >, [])$

Where,

- *appliesto* is a keyword that identifies that the next item is a location to which the policy applies, and as this specific version deal with workflows *taskName* is the name of a Task. Tasknames are not part of STPOWLA, but rather of the application domain.

- *taskEntry* denotes the trigger event, in STPOWLA these are related to entering tasks.

- *req* is an action $req(\_\ ,\ \_\ ,\ \_)$ for refinement policies that takes three arguments: the type of Service, which express its basic functionality (The default is the name of task, denoted by *main*), a list of service parameters, in terms of task parameters and attributes (the parameters and attributes are specific to the application domain) and finally a service level requirement (the default is empty; specific values are application domain specific). For reconfiguration policies, the action *req* is replaced with an action such as *insert* or $delete$, specified for reconfiguration (this concept is discussed in the next section).

Reconfiguration policies

Reconfiguration policies are used to make structural short lived changes in a workflow/system. STPOWLA reconfiguration policies add some additional triggers and operations that can be used in policies; these are not application but rather workflow domain specific. In STPOWLA a policy expresses a reconfiguration rule based on number of available function such as *insert(x,y,z), delete(x), abort(),*

fail() and *block(s,p)*. These functions are defined by STPOWLA, the detailed discussion can be found in Section 4.1.

Example

Consider an example of a reconfiguration policy from [14]: a supplier whose business process is to receive an order from a registered customer, and then to process that order (which includes collecting, packing and shipping the items, plus invoicing the client). There are no extra constraints on each task, therefore the default task policies are effectively "empty". Now consider that under certain conditions (e.g. financial pressure), a financial guarantee is required from all customers whose order is above a certain amount. We may have the following policy:

GetDepositIfLargeOrder
appliesTo *receiveOrder*
when *task_completion*
if *receiveOrder.orderValue >250000*
do insert*(requestDeposit,receiveOrder, false)*

This policy (*GetDepositIfLargeOrder*) applies to the *receiveOrder* task. It says that, when the task completes successfully and the attribute *orderValue* (bound to that task) is above 250000, then there should be an action. The action in this case is the insertion of a task *requestDeposit* into the workflow instance after (not in parallel to) the *receiveOrder* task.

Refinement Policies

Refinement policies specify criteria for the selection of services to be chosen and invoked. The *conditions*, *actions*, *task* and task attributes for a specific `policy`

are not defined at domain level. The required concepts come from specific application domain. Each individual application area has its own concepts: for example banking applications differ from traffic management application. For example in a hotel booking workflow "Book double room" is a task; "order an application form" could be a task from the banking domain. So in the workflow domain, the exact definition of the policies are application dependent.

Example

Here is an example of a policy **P1** based on the *On Road Assistance Scenario* [48]. The policy **P1** expresses:

"if the car fault happens in the driver's home town, then the driver will select the garage, otherwise one is chosen automatically."

Policy *P*1
appliesTo *OrderGarage*
when *taskEntry*([ ])
if *location=myTown*
do *req* (*main*, [ ], [*Automation* = *interactive*])
seq
when *taskEntry*([ ])
do *req* (*main*, [ ], [*Automation* = *Automatic*])

The above example is dealing with an instance of a policy that contains some core concepts (e.g. *when, if, do*), some workflow specific (aka domain) concepts (*taskEntry* and *req*) but also some application specific terms (*OrderGarage*, the *automation* attribute and also the fact that location inside conditions applies to physical locations, that is *'places'*).

So these refinement policies clear the need of application specific concepts so

we refer them as Application Policies.

2.2.5 Tools

VIATRA

VIATRA (Visual Automated model Transformation) is an Eclipse-based general purpose model transformation engineering framework that supports the entire life-cycle for the specification, design, execution, validation and maintenance of transformations within and between various modelling languages and domains.

Graph patterns are used to define model transformation in VIATRA. Patterns can be defined as a collection of model elements arranged in a certain structure. Patterns can be matched on certain model instances, and upon successful pattern matching, elementary model manipulation is specified by graph transformation rules. These rules describe pre- and post conditions to the transformations.

The static syntax of the VIATRA modelling language is defined in the form of UML class diagrams, it follows the basic concept of MOF metamodelling [63]. VIATRA belongs to the MDA technological space and uses an XMI input/output format that conforms to the MOF model. The syntax is formalized by typed, attributed and directed graphs. Metamodels are interpreted as type graphs, and models are valid instances of their type graphs [83]. Models, modelling languages and transformations are all stored uniformly in the so-called *model space*, which provides a very flexible and general way for capturing languages and models on different meta-levels and from various domains, tools or technological spaces following the Visual Precise Metamodeling (VPM) approach[83].

Existing application domains of VIATRA include dependable embedded systems, robust e-business applications and business workflows, middleware, and service-oriented architectures [4].

VIATRA has a transformation-based verification and validation environment for improving the quality of systems designed using the UML approach by automatically checking consistency, completeness, and dependability requirements [20]. It is the only graph transformation tool that supports higher-order transformations [82]. The VPM, the metamodelling framework used by the VIATRA tool, also supports the reusability of transformations by means of rule inheritance[83]. A planner algorithm [84] could be used for verifying syntactic correctness and completeness of VIATRA transformation rules [20]. The SAL intermediate language [13] is used for verification of semantic correctness of transformation. This can be done by projecting model transformation rules [20].

ALLOY

ALLOY [44] is a declarative modelling language, for expressing complex structural constraints and behaviour, specifications of object models through textual syntax. ALLOY is a formal method that include a logic, language and an analysis tool. The logic is a relational logic providing the building block of the language. All structures are represented as relations, and structural properties are expressed with operators. State and execution are both described using constraints. The language adds a small amount of syntax to the logic and structuring description. The analysis performed is a form of constraint solving. The tool is a first order model checking tool that allows to describe a system model and will check it for consistency. Checking involves finding a counterexample if the given predicate is inconsistent. The tool also allows simulation involving finding instances of state or execution that satisfy a given property.

An ALLOY model consists of a number of signature declarations, facts and predicates.

- **Signatures.** A signature introduce a basic data type, denotes a set of atoms

and a collection of relations (signature fields), similar to the concept of class and set of objects in Object oriented programming. Atoms are indivisible (they cannot be divided into smaller parts), immutable (their properties remain the same over time) and uninterpreted (they do not have any inherent properties).

- **Facts.** Constraints that are assumed always hold are recorded as facts.
- **Predicates.** A predicate is a constraint, with zero or more arguments. Instances of a model can be checked to satisfy a given predicate.
- **Assertions.** Assertion is a constraint that is intended to follows from the fact of the model.

ALLOY is supported by a fully automated constraint solver, called ALLOY Analyzer [3], which allows analysis of system properties by searching for instances of the model. It is possible to check that certain properties of the system (assertions) are satisfied. This is achieved by automated translation of the model into a Boolean expression, which is analyzed by SAT solvers embedded within the ALLOY Analyzer. A user specified scope on the model elements bounds the domain. If an instance that violates the assertion is found within the scope, the assertion is not valid. However, if no instance is found, the assertion might be invalid in a larger scope.

ALLOY has received considerable attention in the research community. It has already been used to detect policy interactions. For example, it has been successfully applied to modelling and analysis, detecting feature interaction [51], Access Control policy validation [43], spatio-temporal access control model [66], model checking of health care domain[11] and Governance policies for privacy access control and their interactions [42].

In this thesis (Chapter 4), ALLOY has been used for model a policies and detecting conflicts among the policies. Following are the main reasons why we choose to use ALLOY as a model checker for policy conflict analysis.

We have used a model driven technique to provide the method for extension of the APPEL policy language. The metamodel of APPEL is represented as a UML class diagram. As the syntax of ALLOY is suitable for object oriented model, UML representation of models would benefit. There are some similarities between ALLOY and UML, i.e, UML Classes can be defined as Signatures in ALLOY. Association can be defined as relations in ALLOY. UML constraints can be defined by facts and predicates. Fact are the values that hold always true, predicates and assertions are used to verify certain properties of model. As interactions are exists between the policies, it is necessary to detect these interaction before proceed to make use of these policies. ALLOY has already proven useful for detection of feature and policy interactions.

Chapter 3

Domain Modelling for Policy Language Specialization

Policy languages have been used for a variety of applications in software systems, and usually each application has received its own language. Developing these languages requires a large investment in time, both of designers as well as domain experts. In the light of this, it is desirable to reuse these as much as possible.

APPEL is a policy language that has been designed with domain specialization in mind, however these domain specializations are usually conducted in an ad-hoc fashion, rather than following a structured process. Currently there is no any formal structured process for domain specialization of the APPEL policy language.

The objective of this chapter is to define a formal structured way of policy language specialization that can be used to extend the APPEL policy language for domain specialization. Specialization usually represents a large amount of effort involving domain experts. Using UML this knowledge is often captured in well-defined models. The aid of these models is to provide as much reuse as possible. MDD (Model Driven Development) describes a method where the definition of the models and meta-models is central. MDD underlies support techniques and

provide tools to work with these models. The chapter is structure as follows. Section 3.1 defines the parametrization approach. The domain specialization of the APPEL policy language is presented in section 3.2. Section 3.3 presents the implementation of parametrization.

3.1 Parametrization

Parametrization is a model composition technique that allows existing model to be reused in different context. Advantages of this are:

- Complexity can be managed at the level of smaller models. In general one only needs to focus on modelling either domain or application specific concepts and the small intersection between the two models.
- The number of concepts of individual existing models can be reused. The core only needs to be modelled once and domains can usually be reused.
- For policy languages a structured specialization process is defined.

The idea of parametrized composition approach is inspired by [65], and we explored this approach in [47], which explicitly describes how to compose three specific models with parametrization. Our solution is build on this idea and adds a graph transformation based methodology to work with generic models.

The standard process of parametrization defined in [65] is expressed by:

$mm' = mm[fp \xrightarrow{\varphi} ep, F_{fp}]$
where

- mm, mm', fp and ep are models,

- $ep \supset \varphi(fp)$, redefines the elements in ep, by replacing the correlated elements of fp by those from ep.
- $\varphi : fp \longrightarrow ep$ is a total function that creates a map between elements of fp and ep, and
- F_{fp} is a set of formulas representing constraints over fp that must be respected.

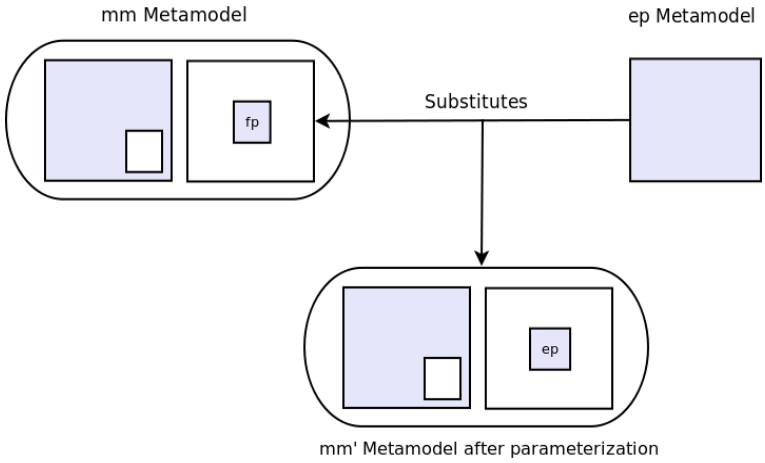


Figure 3.1: Graphical representation of Parametrization Process [65]

A simplified diagram of the metamodel parametrization is presented in Figure 3.1, which shows mm' metamodel is extended by defining its formal parameters and substituted it with an effective parameter.

3.2 Extension of a Policy Language

This section describes how parametrization approach is be used for policy language extension.

The APPEL core language leaves the details of Triggers, Conditions and Actions undefined. There is then a need of extending domain concepts inside the policy

language by specializing triggers, conditions and actions with the domain concepts. This is achieved by defining the domain concepts as a metamodel which is then used as a parameter to the APPEL metamodel, to produce a target model using our composition approach.

The parametrization approach is used here for the extension of policy language the APPEL. There are three models involved in the extension process. The APPEL policy model, STPOWLA: the domain model and the application model. The parametrization process is applied twice, first to specialize the APPEL with the STPOWLA, and then a resultant model with the application model. All three models are briefly discussed in following section.

3.2.1 Modelling of APPEL

The APPEL policy language has been designed with extensibility and domain customization in mind. It has been defined syntactically and been given formal semantics (discussed in Section 2.2.3), however extension has been rather ad-hoc. This section formalizes APPEL by providing a metamodel (as a UML class diagram) for the core language as a basis for the structured extension.
The core defines the structure of the language that is the basic constructs we would expect in all policies such as Triggers, Conditions, Actions and their relationships. The details of these constructs are left for later definition, and can be specialized for specific domains when needed. The core concepts of APPEL are used to design a metamodel for APPEL. We already discussed the APPEL policy language in Section 2.2.3. The UML model of APPEL is represented in Figure 3.2.

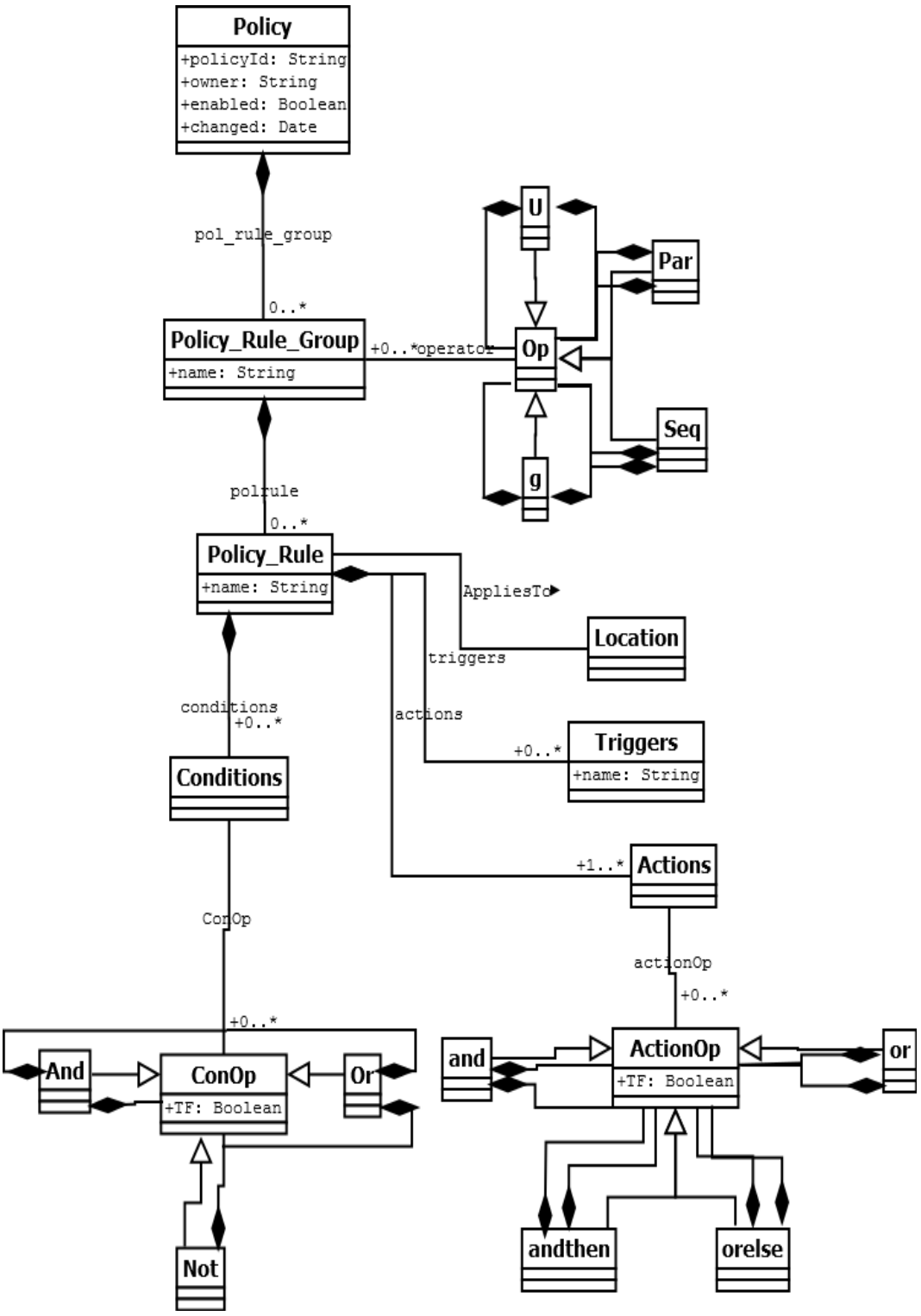


Figure 3.2: APPEL Meta-Model

3.2.2 Domain Modelling: STPOWLA

A Domain is an area of knowledge that include a set of concepts and terminologies understood by practitioners meant to use the language. Domain models are abstract representations of the knowledge and activities that govern a particular application domain.

Considering a policy language, this would need the specific actions and triggers to be used in the application domain. In the case of APPEL and its origin, these would be "call control" concepts such as "forward-call" or "outgoing-call". These concepts would come from a call-control domain model. As APPEL has been extended by modelling STPOWLA to deal with workflows, new concepts have to be made available, e.g, "start-task" as a trigger. In this spirit the domain specialization to workflow needs a domain model for workflow to be composed with APPEL; domains such as security or access control easily can be envisioned as other domains.

STPOWLA [39] has already been discussed in Chapter 2. The UML model of STPOWLA is shown in Figure 3.3.

3.2.3 Application Modelling

What is an action at domain level? It could be simply the execution of a task, certainly true at the workflow domain level, but not necessarily of much use to the practitioner. Typically the required concepts would come from a specific application domain and hence would be a specialization beyond "workflow" to express e.g "the booking of a hotel" or "payment for a car repair".

Clearly this argument is not restricted to actions, but applies to other elements as well. This is referred to as a second level of specialization for the application, but it is again just a model composition. Based on this argument, parametrization

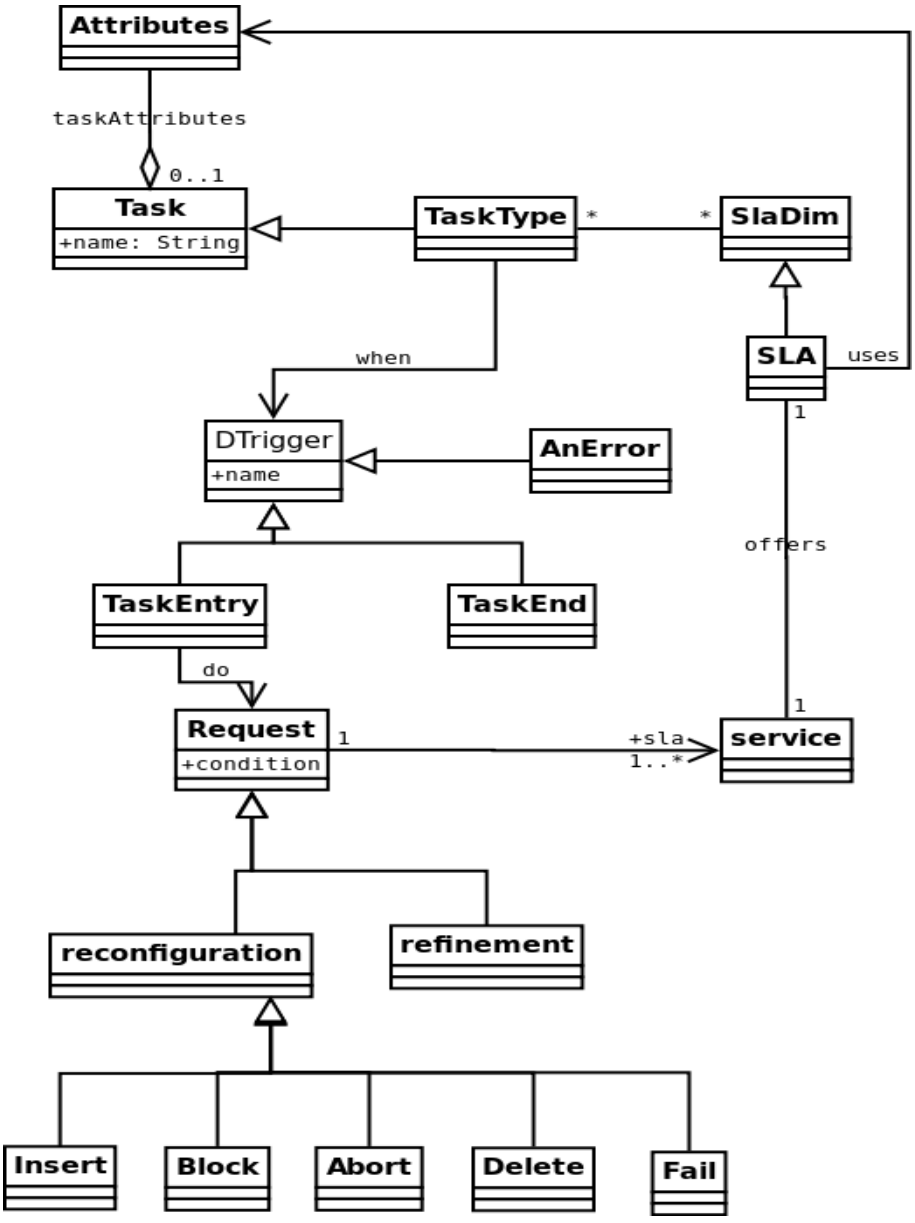


Figure 3.3: Domain Meta-Model: STPOWLA

process is applied twice to specialize a policy language.

As an example we consider the *On Road Assistant Scenario* the *automotive* case study of SENSORIA [87], to illustrate our approach. We define a UML model for the case study, which is given in Figure 3.4. The scenario is described below:

"The diagnostic system of a car engine reports a severe failure in the car engine, the car is no longer driveable, and sends a message with the diagnostic data and the vehicles GPS data to the car manufacturer or service center. Based on availability and the drivers preferences, the service discovery system identifies and selects the appropriate services in the area: garage, tow truck and rental car. When the driver makes an appointment with the garage; the data is automatically transferred to the garage, A towing service is also identified by the discovery system, the driver makes an appointment with the towing service, and the vehicle is towed to the garage. It is assume that the owner of a the car has to deposit a security payment before able to order services" [36].

3.2.4 Model Extension

This section describes how model parametrization can be used to extend a policy language. The method of parametrization defined in Section 3.1 is now applied to the APPEL and Domain model. The process of parametrization in terms of the APPEL policy language can be defined as:

$$tm = am[fp \xrightarrow{\phi} dm, F_{dm}]$$

where

- am is the metamodel of the APPEL policy language, shown in Figure 3.2
- fp the metamodel subset of am are the formal parameters,

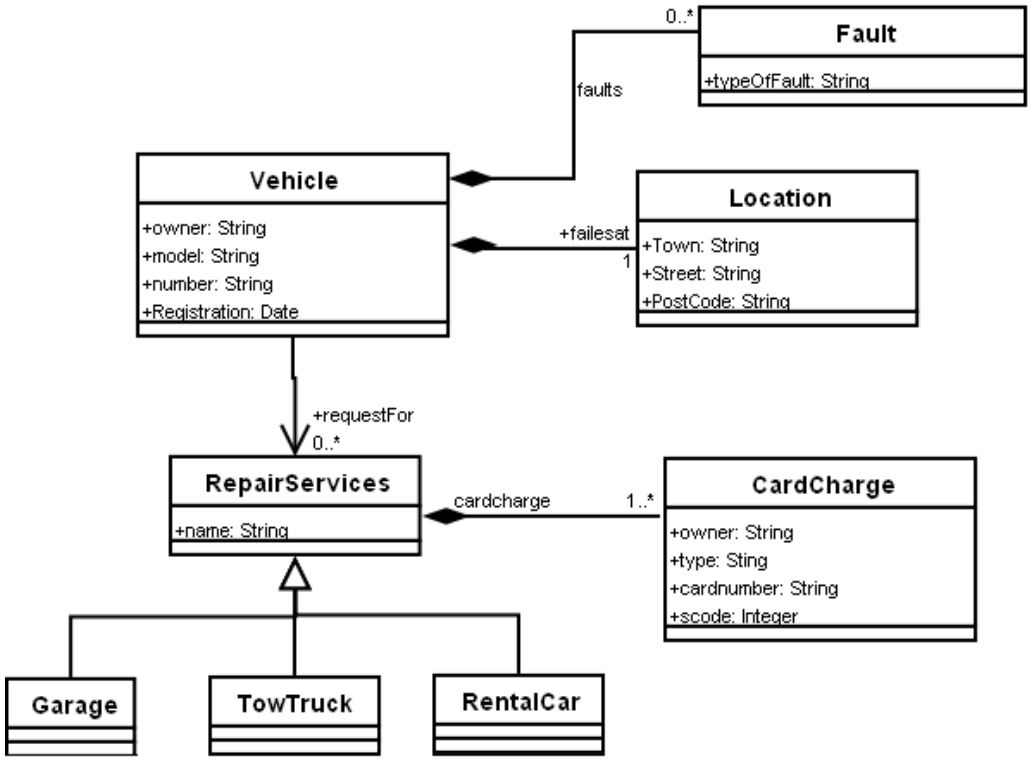


Figure 3.4: Application Model

- dm is the domain metamodel for workflow as shown in Figure 3.3,
- tm is the target metamodel obtained after instantiation,
- ϕ is a total function that creates a map between elements of fp and dm
- F_{fp} is a set of formulas representing constraints over fp that must be respected.

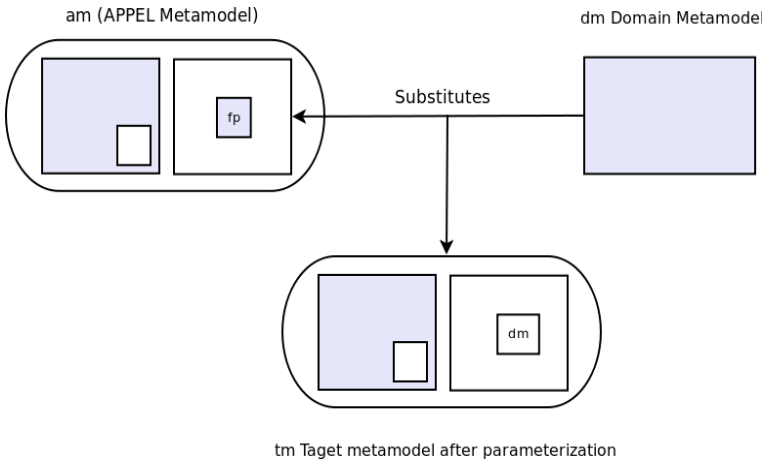


Figure 3.5: Graphical representation of Parametrization Process

The simplified representation of above process is shown in Figure 3.5. In APPEL metamodel (Figure 3.2), we consider four classes (`Location`, `Trigger`, `Condition` and `Action`) as fp (formal parameters). We use the term *actual parameters* for the elements in dm that are replaced by $formal\ parameters$. The operator ϕ can be defined as replacing the element of fp by dm. *Task*, *DTrigger*, *Dcondition* and *Request* are actual parameters that replaces the formal parameters. *Location* will replaced by *Task*, *Trigger* with *DTrigger*, *Condition* with *DCondition* and *Action* with *Request*. After parametrization the target model tm would graphically be seen as one shown in Figure 3.6.

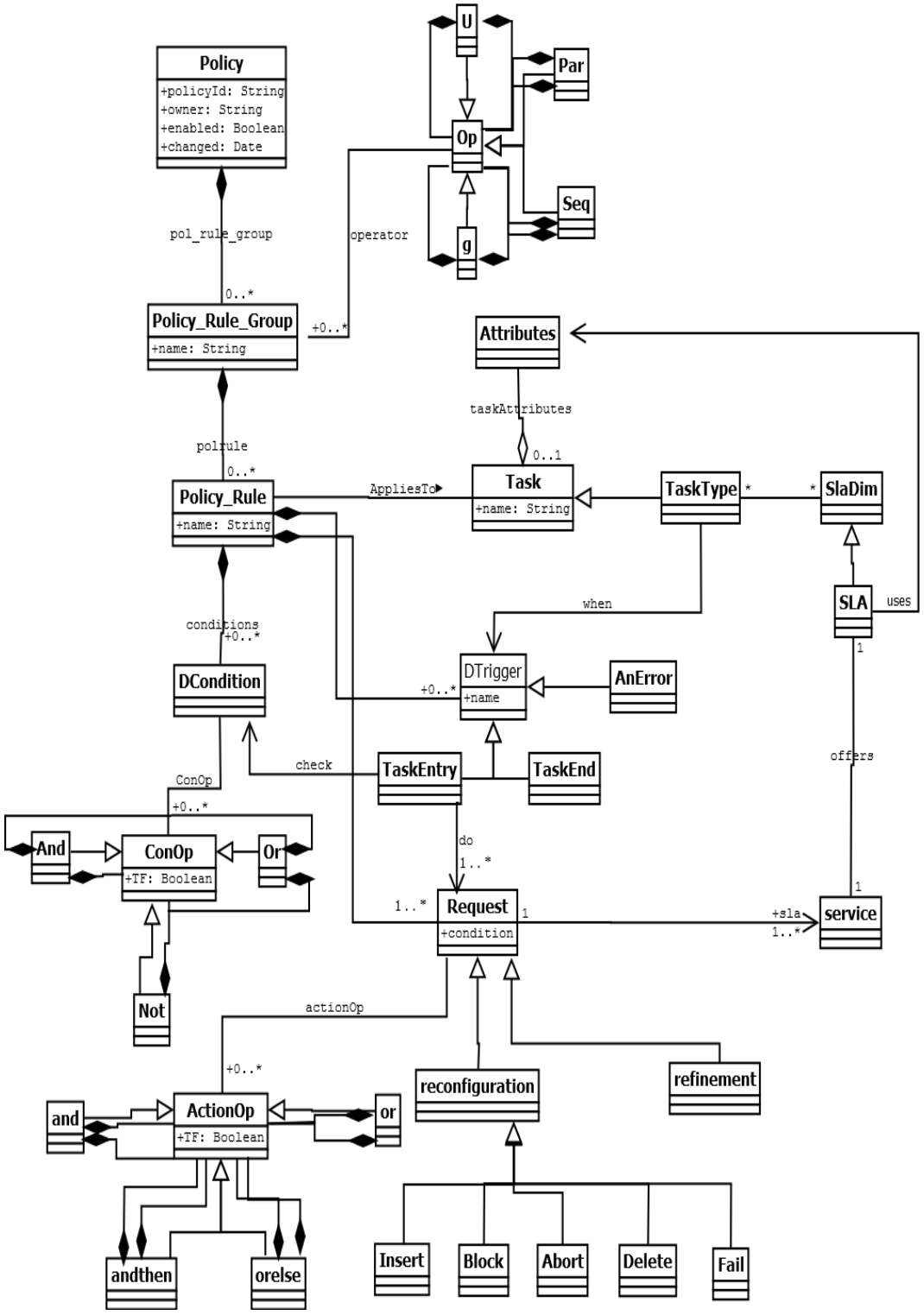


Figure 3.6: Target Model (*tm*) after Parametrization

As we discussed above some domain concepts need to be refined in order to make it applicable in the specific application. The domain model has some gaps e.g, `TaskType` can't be defined at this level, and `Attributes` needs to come from the specific application. These gaps need to be filled by application-specific concepts. This is refered to as a second level of specialization.

The Application model represents the concepts of a particular application in terms of its attributes, entities and their relationships. As an application, we have considered the *On Road Assistant* scenario from the SENSORIA case study (presented in section 3.2.3). In order to specialize the domain-specific policy model (*tm*) with the Application, we will repeat the parametrization process.

Consider *ap* as an Application Model and *tm* is the target model obtained previously. *Attributes* and *TaskType* in *tm* are considered as fp ($formal parameters$), while *Vehicle* and *RepairServices* in *ap* are $actual parameters$. The process of parametrization at this stage is:

$$fm = tm[fp \xrightarrow{\phi} ap]$$

$$\phi : fp \longrightarrow ap$$

where, ϕ is a function of replacing the elements of fp by ap. *RepairServices* and *Attributes* are actual parameter in *ap* and *TassType* and *Vehicle* are formal parameters.

After second level of specialization (parametrization) the Final Model would graphically be seen as one shown in Figure 3.7.

3.3 Towards Automated Instantiation

In Section 2.3, we have discussed the relationship between model and graph and how model transformation problems can be formulated as graph transformation

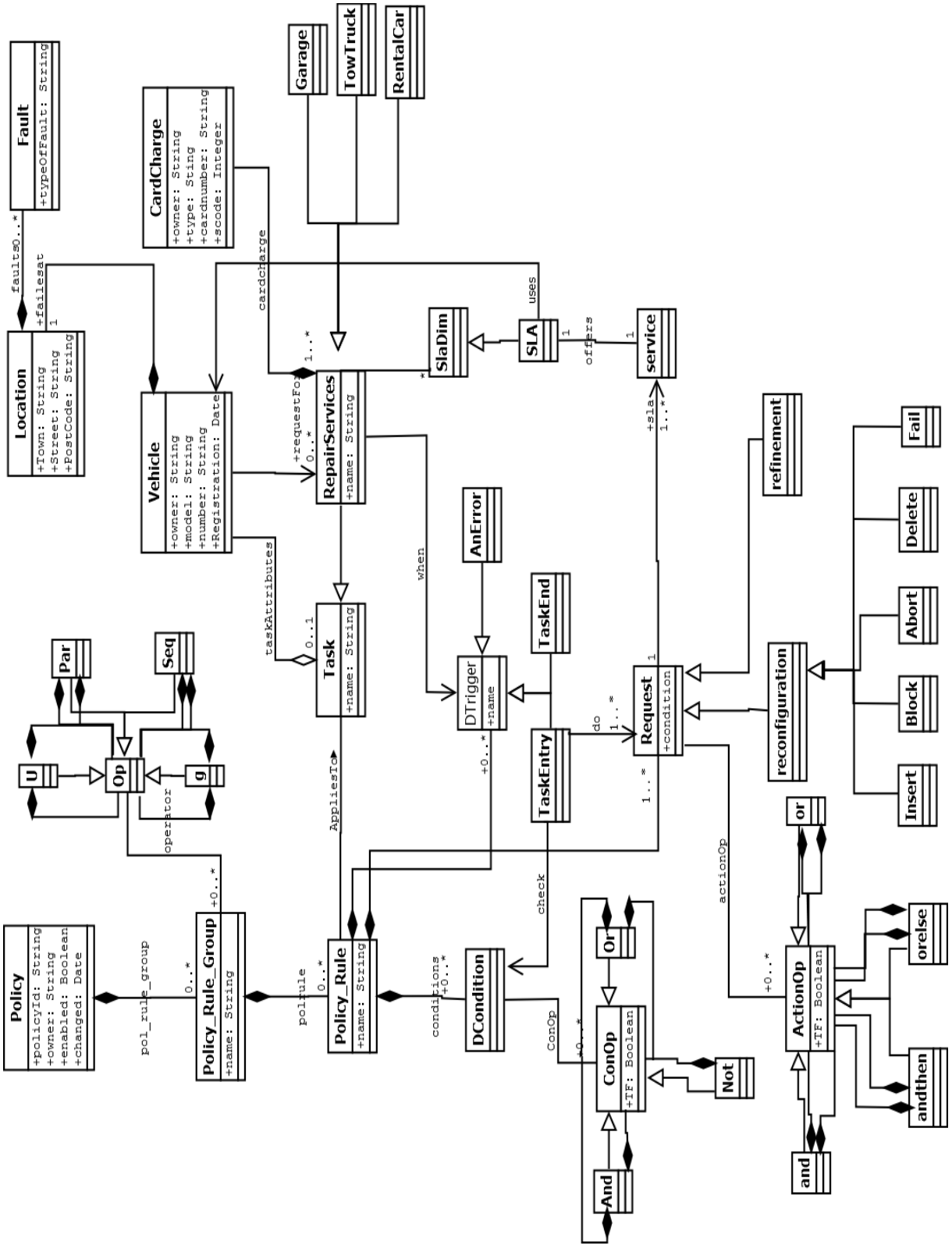


Figure 3.7: Final Model after Parametrization

problems. The above parametrization process can be implemented using a graph transformation tool, subject to the appropriate transformation rules being identified and applied in the right order. These needs to capture the function φ.

In order to automate the parametrization process, we have some requirements on tools. Specifically, we have identified two main requirements: support for many to one transformations and support for models expressed in UML syntax. The former requirements arises as we have two input models (here *mm* and *dm*) which are transformed into one (here *mm*'). One of the main requirements for this work is a UML representation of these models (this would also be required for the policy conflict analysis method). The required tool should support many to one transformations and also supported by UML syntax.

We investigated graph transformation technologies to find a suitable a tool for the parametrization. Research in [56] compared many graph transformation tools, their advantages, disadvantages and their use. With respect to the the requirements of proposed transformation, the VIATRA2 seems to be most suitable tool, among others (AGG [72], Fujaba [6], GReAT[8]). With respect to the technological space, it supports MDA, so UML, business process models and XSD models are supported. It uses an XMI input/output format that conforms to the MOF model. It also supports *many-to-one* and *many-to-many* transformations. "Compared to AGG and Fujaba, the VIATRA tool is more tuned to the activity of model transformation since it was specifically built for this purpose" [56].

The static syntax of the VIATRA modelling language is defined in the form of UML class diagrams. It follows the basic concepts of MOF meta modelling [63]. The syntax is formalized by typed, attributed and directed graphs. Meta models are interpreted as type graphs, and models are valid instances of these type graphs [83]. Models, modelling languages and transformations are all stored uniformly in the so-called *model space*, which provides a very flexible and general

way for capturing languages and models on different meta-levels and from various domains, tools or technological spaces following the Visual Precise Metamodeling (VPM) approach [83].

Graph transformation rules are defined by pre and post conditions in terms of LHS and RHS patterns respectively. The most elementary behavioural concept in VIATRA2 is the machine. Each machine has a single main rule that defines the rule which is first called when the transformation specified in the machine has been started [5]. There are no pre and post condition for the main rule, as the purpose of the rule is to start the execution of other graph transformation rules.

As VIATRA2 seems to fulfil our requirements, so we used VIATRA2 graph transformation tool for the implementation of parametrization. VIATRA2 uses the VPM metamodeling language for model and meta model representation. The meta model and model are defined either by textual syntax, or using the Visual Editor of Viatra as a (VTML) file. We defined our meta models using VTML files. Figure 3.8 and 3.9 present the APPEL and STPOWLA metamodel receptively in the (VPM) VIATRA2 Model Space. In the following section, we provide two general rules for the implementation mechanism of parametrization, so it can be applicable to similar problems as well. The terms $formal$ and *actual parametesr* are one defined in Section 3.2.4.

Consider M_i as a metamodel element (that has a relation with $formal\ parameter$) and m_i as a model elements, where $i \in \{1,2,3\}$. Let FP_i be $formal\ parameter$ in the metamodel and fp_i formal parameter in the model. Let r_i be a relation between m_i and fp_i in the model and AC_i and ac_i be $actual\ parameter$ in the other metamodel and model respectively. We are defining two general rules. First for the assocaition and aggregation relations and second for the generalization relation. Considering $i = 1$, the rule to replace a $formal\ parameter$ with the respective $actual\ parameter$ in VIATRA2 can be defined as follows:

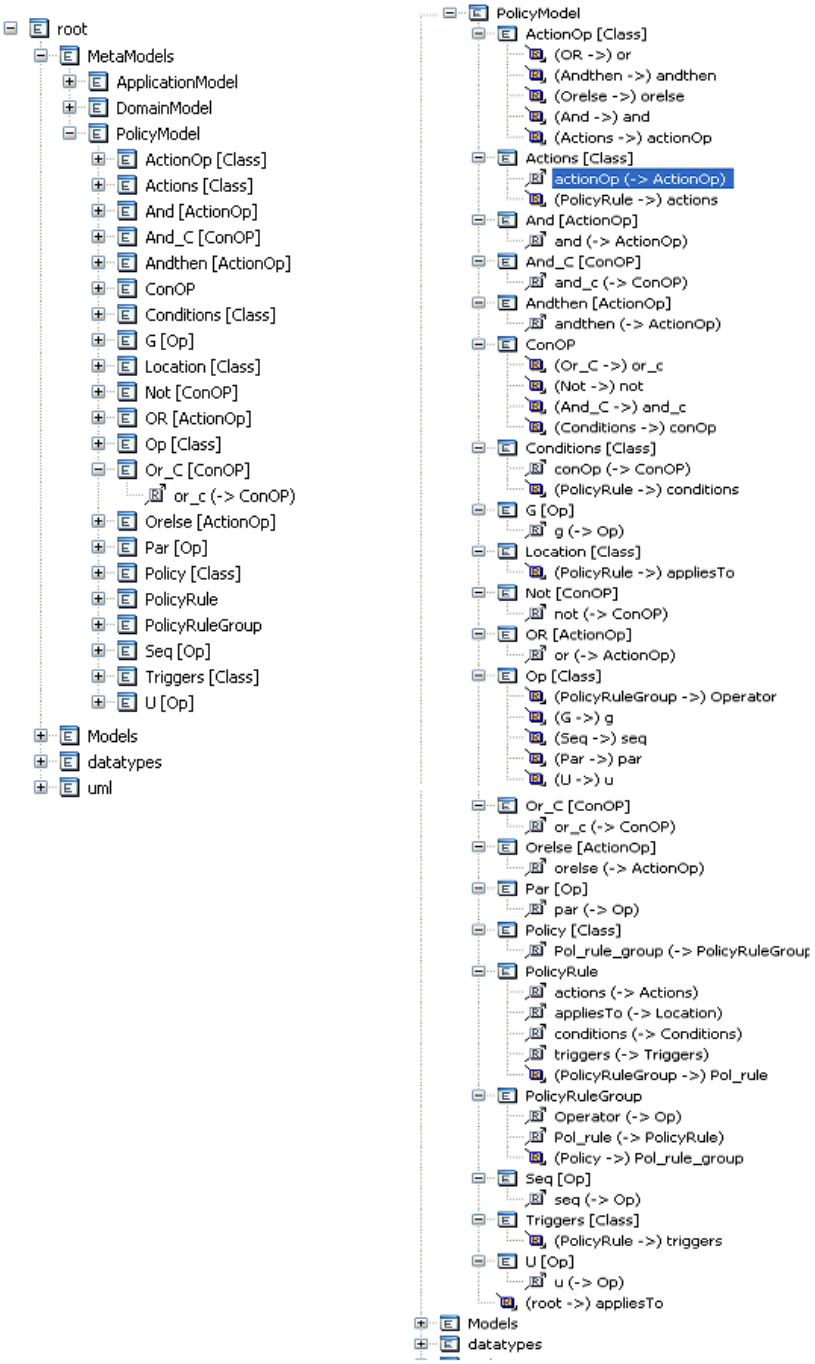


Figure 3.8: APPEL Model in VIATRA2 Model Space

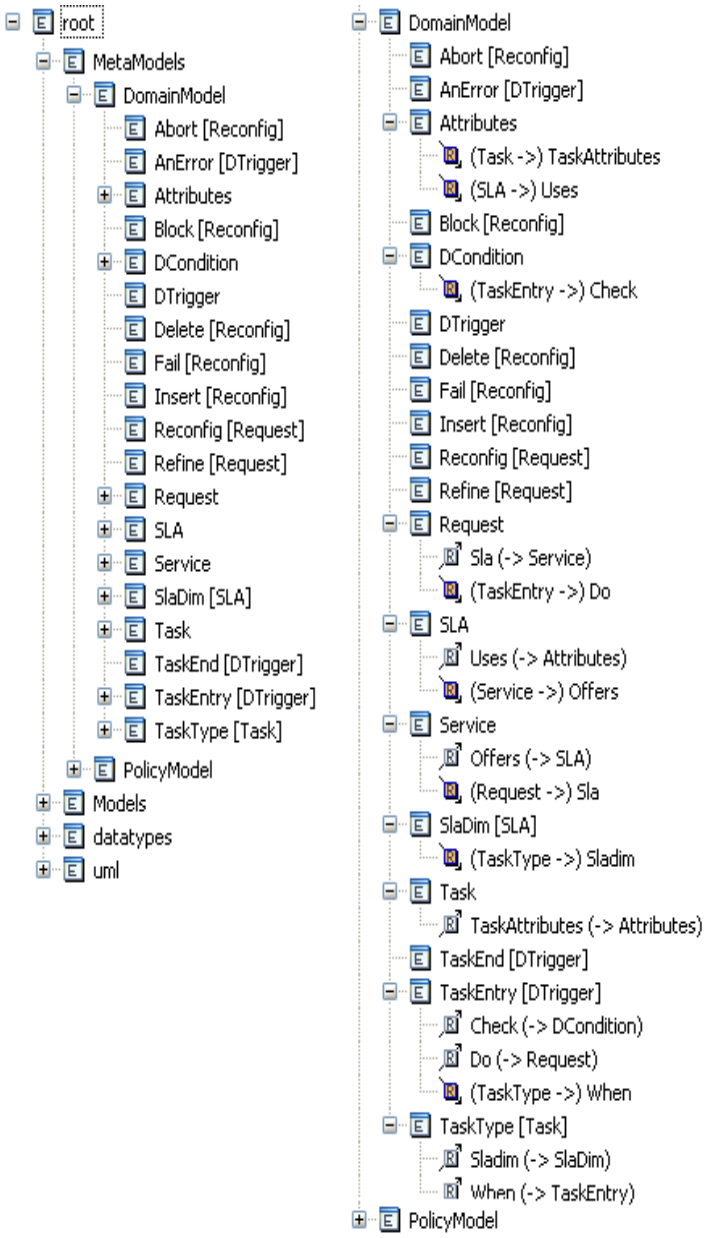


Figure 3.9: Domain Model in VIATRA2 Model Space

```
pattern Formal1(m1,fp1,r1) =
        {
        M1(m1);
        FP1(fp1);
        relation(r1,m1,fp1);
        }
pattern Actual1(ac1) =
        {
        AC1(ac1);
        }
gtrule rule1(inout m1, in fp1, inout ac1, inout r1)=
{
      precondition pattern lhs(m1,fp1,ac1)=
      {
      find Formal1(m1,fp1,r1);
      find Actual1(ac1);
      }
      postcondition pattern rhs(m1,ac1,r1) =
      {
        M1(m1);
        AC1(ac1);
        relation(r1,m1,ac1);
      }
      action
      {
        delete (fp1);
```

```
    }
}
```

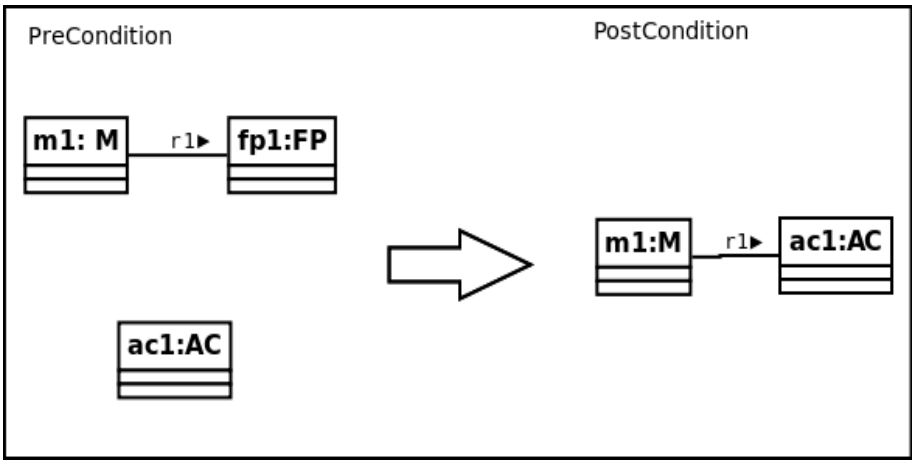


Figure 3.10: First General Rule

The above rule is encoded into VIATRA2 textual syntax, is shown graphically in Figure 3.10. The *rule*1, uses two patterns *Formal*1 and *Actual*1, one from each model. The rule will match the pattern described in LHS and create a relation $r1$ between the model element $m1$ and the actual parameter $ac1$ as described in post condition. Finally the action part of the rule will delete $formalparameter\ fp1$ from the model.

We need a second general rule for generalization relationship between the model entity and $formal\ parameter$. The rule is given below:

```
pattern Formal1(m1,fp1) =
        {
        M1(m1);
        FP1(fp1);
        supertypeOf(m1,fp1);
        }
```

```
pattern Actual1(ac1) =
        {
        AC1(ac1);
        }
gtrule rule1(inout m1, in fp1, inout ac1)=
{
        precondition pattern lhs(m1,fp1)=
        {
        find Formal1(m1,fp1,r1);
        find Actual1(ac1);
        }
        postcondition pattern rhs(m1,ac1) =
        {
          M1(m1);
          AC1(ac1);
          supertypeOf(m1,ac1);
        }
        action
        {
          delete (fp1);
        }
```

All four $formal parameters$ replace their respective *actual parameters* in the parametrization process following the same structure as first general rule. Hence the rule can be used to implement the parametrization process for the APPEL model and domain model. We defined four GT rules one for each formal parameter to replace with actual parameters. These rules can be applied in any order. The first rule (graphically shown in 3.11) for replacing *Location* with *Task* is

given below:

```
pattern PolicyRelation(P1,L1,ApTo) =
        {
        PolicyRule(P1);
        Location(L1);
        relation(ApTo,P1,L1);
        }
pattern TaskAttribute(Ts1) =
        {
        Task(Ts1);
        }
gtrule rule1(inout P1, in L1, inout Ts1, inout ApTo)=
{
      precondition pattern lhs(P1,L1,Ts1)=
      {
      find PolicyRelation(P1,L1,ApTo);
      find TaskAttribute(Ts1);
      }
      postcondition pattern rhs(P1,Ts1,ApTo) =
      {
        PolicyRule(P1);
        Task(Ts1);
        relation(ApTo,P1,Ts1);
      }
      action
      {
        delete (L1);
```

```
        println("Replace Location with Task");
    }
}
```

The second rule replaces *Actions* with *Request* shown in 3.12. The third rule replaces *Conditions* with *DCondition* shown in Figure 3.13. The fourth rule replaces *Triggers* with *DTrigger* shown in Figure 3.14. The complete transformation is given in Appendix A.

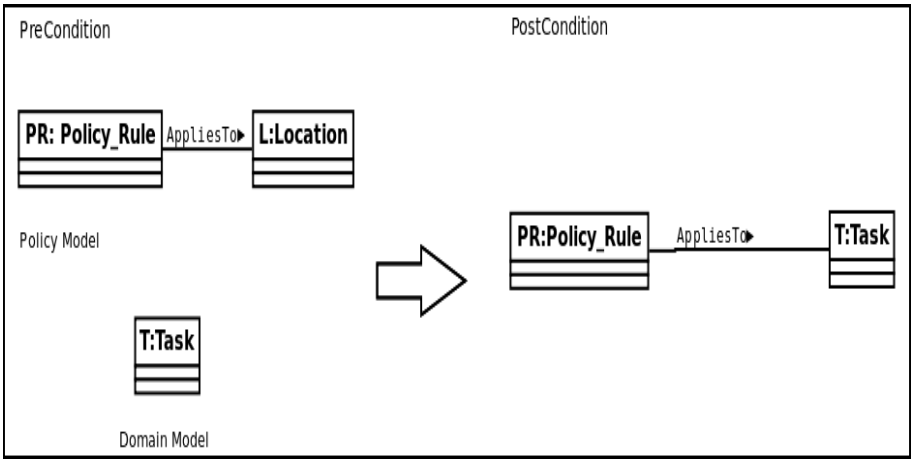


Figure 3.11: Rule 1: Replace Location with Task

The Figure 3.15(a) shows model that is before transformation. By applying transformation rules we obtained a Target Model shown in the Figure 3.15(b).

For the implementation of the second level of specialization (Target model with Application), we will use the target model that we have obtained in the domain specialisation phase. The Application metamodel in (VMP) is shown in Figure 3.16. As there are two formal parameters that need to be replaced by actual parameters, we defined two graph transformation rules. Here we need to use the second general rule as well. The rules are graphically shown in Figure 3.17(a) and Figure 3.17(b). Figure 3.18(a) shows a model before transformation and Figure

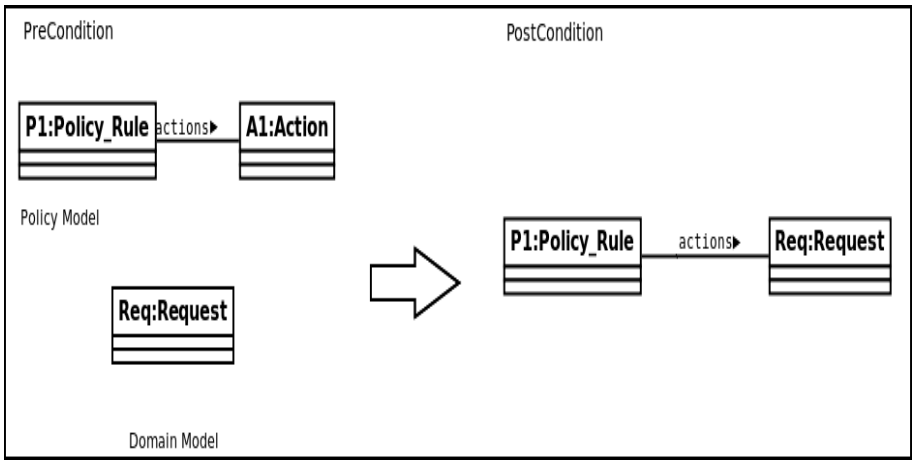


Figure 3.12: Rule 2: Replace Action with Request

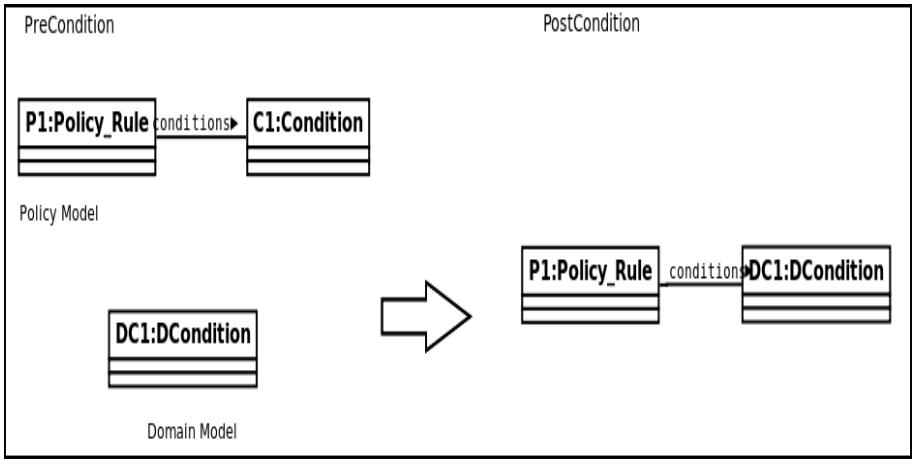


Figure 3.13: Rule 3: Replace Condition with Dcondition

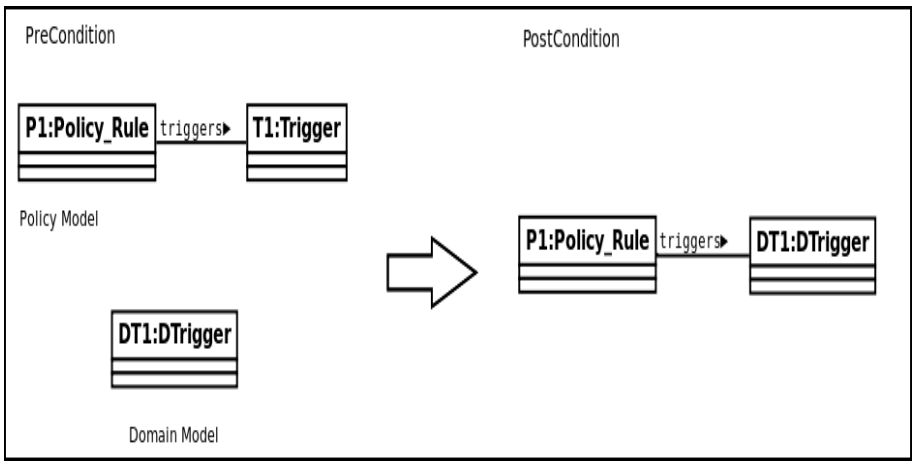


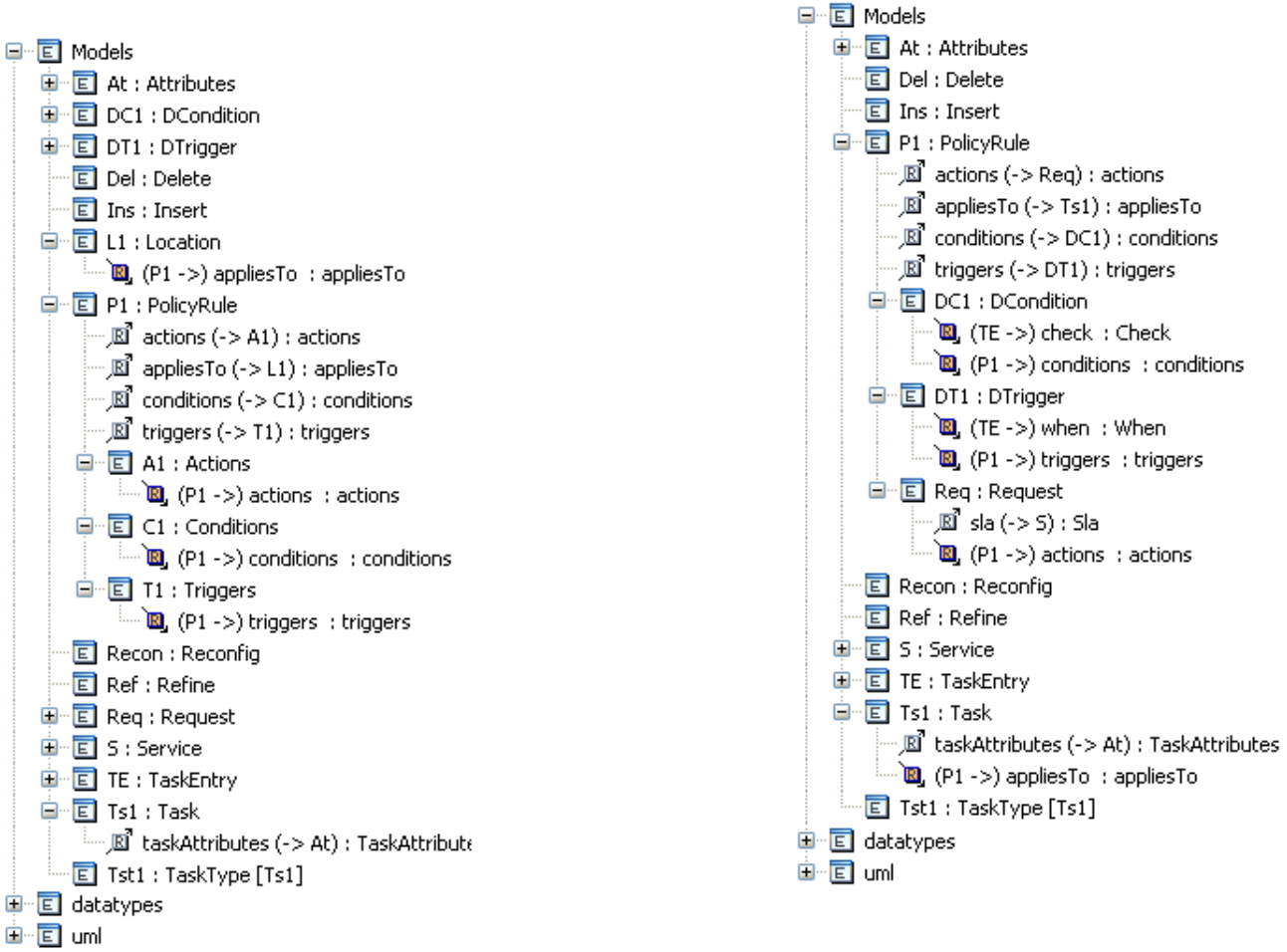
Figure 3.14: Rule 4: Replace Trigger with DTrigger

3.18(b) is after transformation.

Instantiation of metamodels shown in Figure 3.6 and 3.7 are drawn manually. Automation of these instantiation is an option, but we didn't use in these examples.

3.4 Discussion

In this chapter, we have shown how parametrization approach can be used for domain specialization of a policy language. We define this solution for domain specialization of the APPEL policy language for STPOWLA domain, however the approach and its implementation is general and can be applicable to other domains as well.



(a) (b)

Figure 3.15: Before(a) and After transformation(b)

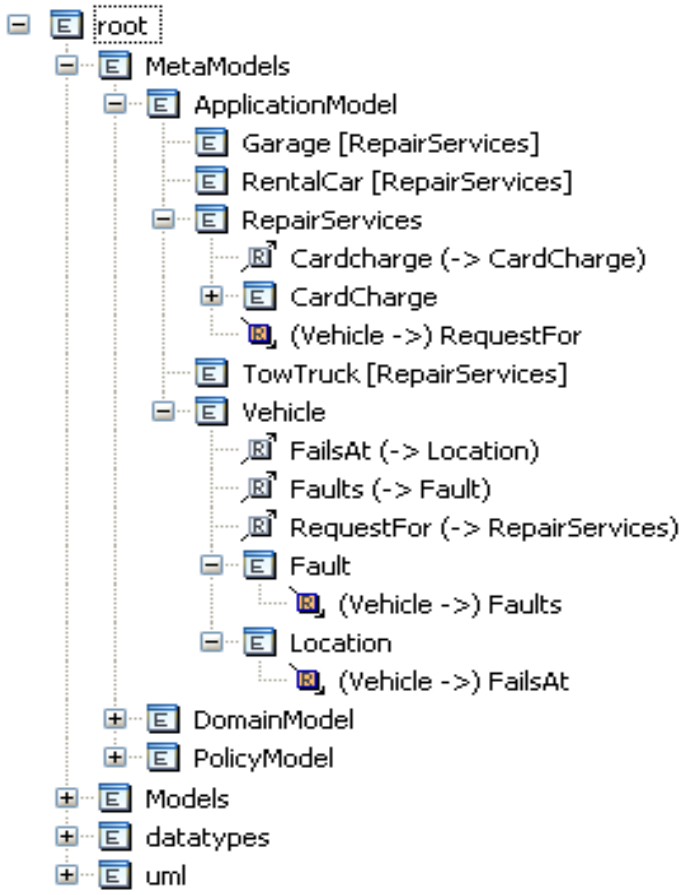
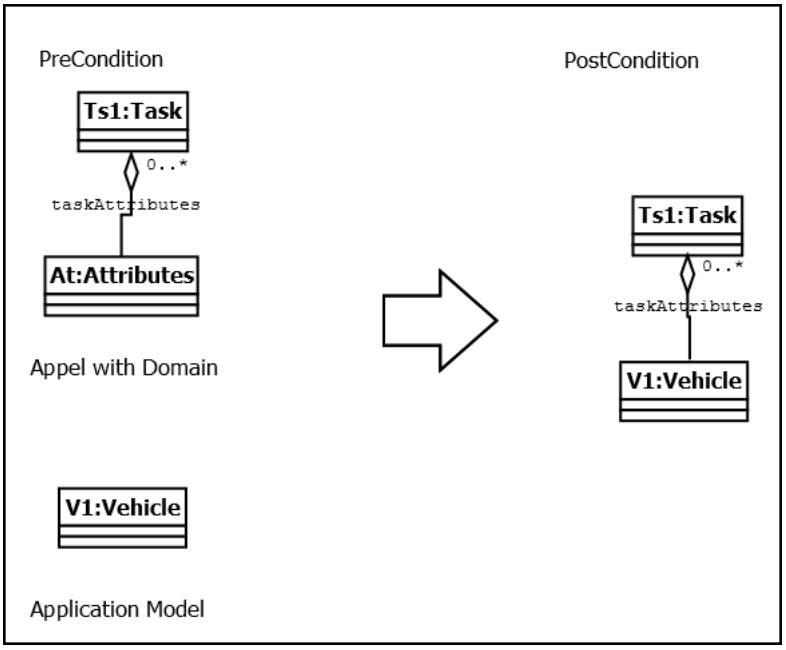
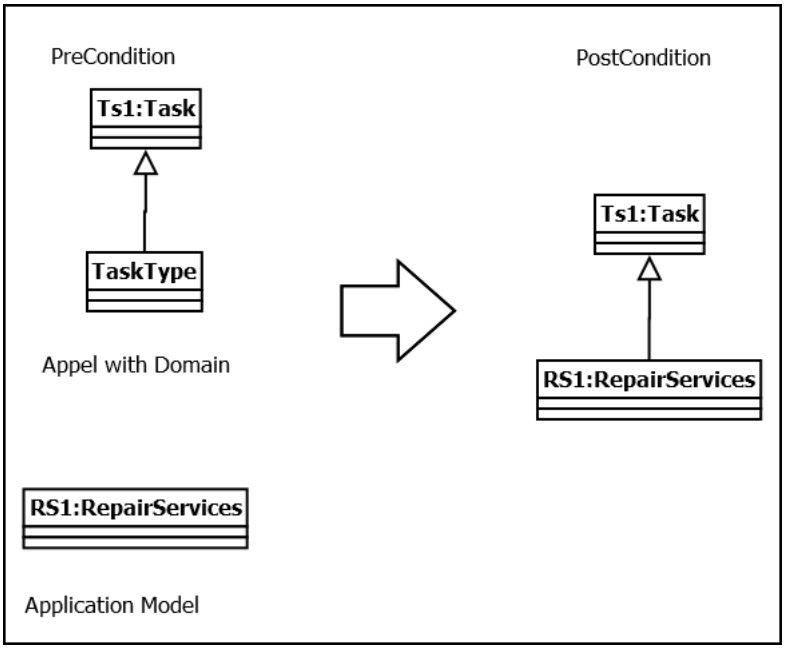


Figure 3.16: Application Model in VIATRA2 Model Space



(a)



(b)

Figure 3.17: Application Rules

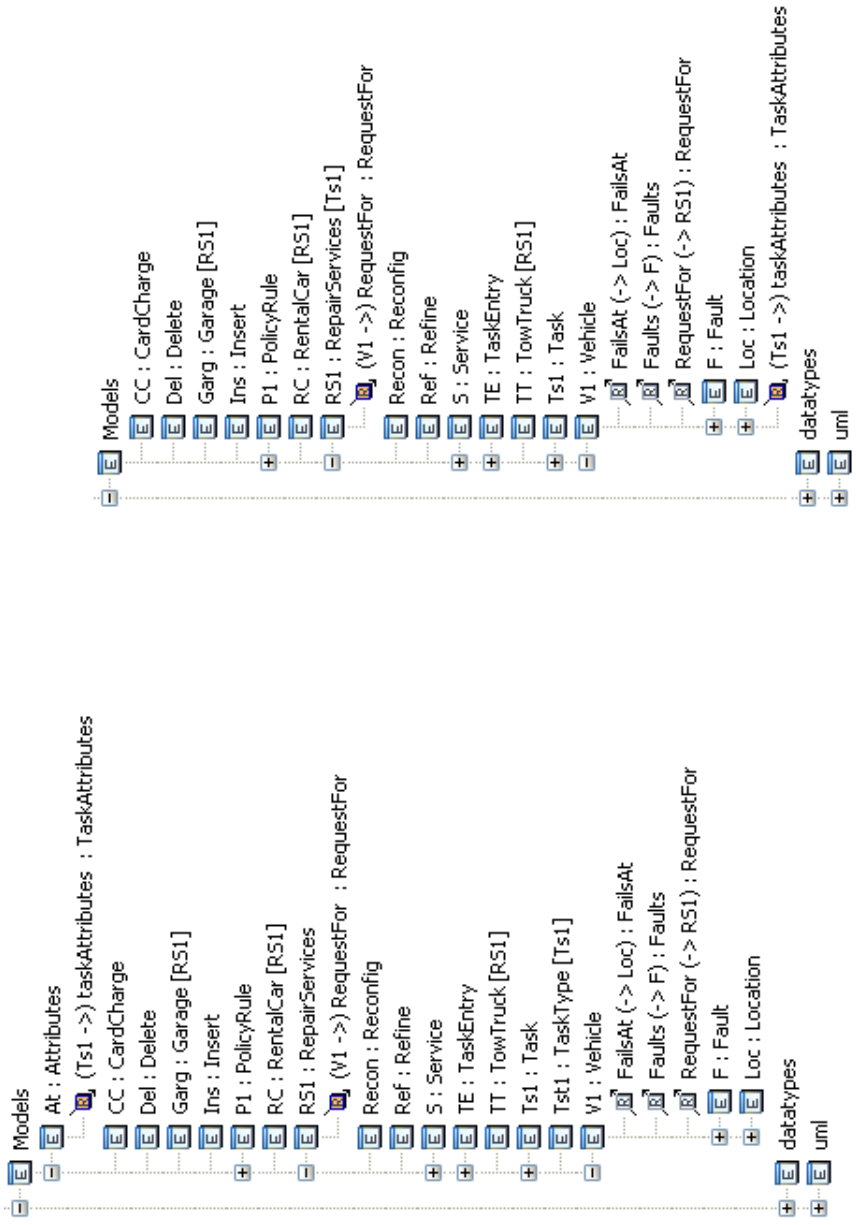


Figure 3.18: Before(a) and After transformation(b)

Chapter 4

Policy Conflict Analysis

Policy conflict can occur when a new or a modified policy is deployed in a policy server which leads to unspecified behaviour. To make policy based systems conflict free it is necessary to detect and resolve conflicts before they occur, otherwise the intended behaviour of a policy cannot be guaranteed.

In general these conflicts leave the system in an unspecified state which is undesirable and, it is therefore necessary to provide a method to detect and resolve these conflicts. As we are analysing conflicts during a design of the policies, resolution is by redesign. The redesigned policies need to be checked again, to ensure that there are no further or new conflicts. The model checking results help to identify the source of the problem and hence allow to reduce the redesign efforts.

The objective of this chapter is to define a conflict analysis method. The analysis is based on our work (structural modelling methodology) as conflicts depend on domain knowledge, so using this knowledge while modelling the domain is sensible. The chapter is structured as follows: Type of policies and conflicts in APPEL are discussed in Section 4.1. The conflict analysis methodology is discussed in Section 4.2. The modelling of a policy language in ALLOY is discussed

in Section 4.3. Conflict analysis and confirmation of conflicts are represented in Section 4.4. Resolution of conflicting policies is discussed in Section 4.5.

4.1 Types of Policies and Conflicts in APPEL

In this thesis, we developed a model driven technique to provide a method for extension of a policy language to achieve domain specialization. We exemplified this using the APPEL policy language. The meta model of APPEL is represented in the form of class diagram in Figure 3.2, Section 3.2.1.

APPEL originally deals with two types of policies, Regular and Resolution policies. These policies have already been discussed in Chapter 2, Section 2.2.3. Regular policies are used to define new policies for the system that is being controlled. The details of regular policies are dependent upon the domain in which the policies are meant to be used, while resolution policies deal with dynamic policy conflict resolutions.

Domain Model

As an example, we consider STPOWLA [39] as a domain in our work. In Section 2.2.4, we discussed STPOWLA in detail, however some of the details are more relevant here, and therefore are summarized below.

Recall that STPOWLA addresses the integration of business processes, policies and SoA at a high level of abstraction. It captures essential requirements at a business level in the form of a workflow and the variability in terms of policies that are expressed in a language close to the business goals (see Figure 3.3 in Section 3.2.2). There are two types of *regular* policies that have been defined for STPOWLA: refinement and reconfiguration policies. Although they have been discussed in Chapter 2, Section 2.2.3 in detail, for better understanding of the

Functions	Description
Insert(y,x,z)	Insert task y into the current workflow instance after task x if z is true, or in parallel with x if z is false.
delete(x)	Delete task x from the current workflow instance
abort()	Abort the current task and progress to the next task, generating the task abort event
fail()	Declare the current task to have failed, i.e. discard further task processing and generate the task failure event
block(s,p)	Wait until predicate p is true before start the task s.

Table 4.1: Reconfiguration Functions

concepts, they are summarised below:

Reconfiguration (Domain) policies

Recall that reconfigurations are used to make structural short live changes in a workflow/system. STPOWLA defines a number of functions to make structural changes in a workflow. These functions are shown in Table 4.1. As changes affect the workflow, they are independent of the specialization area and hence we can see these policies as domain policies.

Refinement (Application) Policies

These policies specify criteria for the selection of services to be chosen and invoked. The domain of STPOWLA assumes that workflow tasks are executed by services in a service- oriented architecture. The judgement of what constitutes conflicting actions is dependent on the particular application.

4.2 Conflict Analysis Methodology

Conflict analysis is the process of identifying policy conflicts in management systems, languages and systems where policies are used. The process can be fully automated or semi automated. We have discussed conflict analysis methods in Section 2.1.2. Two major groups of methods that are commonly used in many approaches for conflict detection are offline/static and online/dynamic. The former are used to detect conflicts at design time, using some formal technique or pragmatic approaches, The detected conflicts are the typically resolved by redesign. These approaches are used before or during policy deployment. The online/dynamic method detects and resolves conflicts at runtime. The proposed method in our work investigates policy conflicts at design time, so this proposed conflict analysis method is static. The various approaches, methods and tools existent to detect conflicts have been already discussed in Chapter 2. We adopt a model checking approach here. It is important to understand that the contribution of this work is not in using model checking, but rather in addressing the issue of using domain information on what constitutes a conflict that is being made an integral part of the domain modelling – not an after thought when dealing with conflicts. In this way the conflict analysis step should be entirely independent of using a domain expert.

4.3 Modelling of the Policy Language in ALLOY

To understand the procedure of conflict analysis, Figure 4.1 shows the overall strategy of conflict analysis. The process is started with the translation of the Domain Specific Policy model into ALLOY, followed by the translation of concrete policies into ALLOY. Finally conflict definition is encoded into ALLOY to confirm the conflicts.

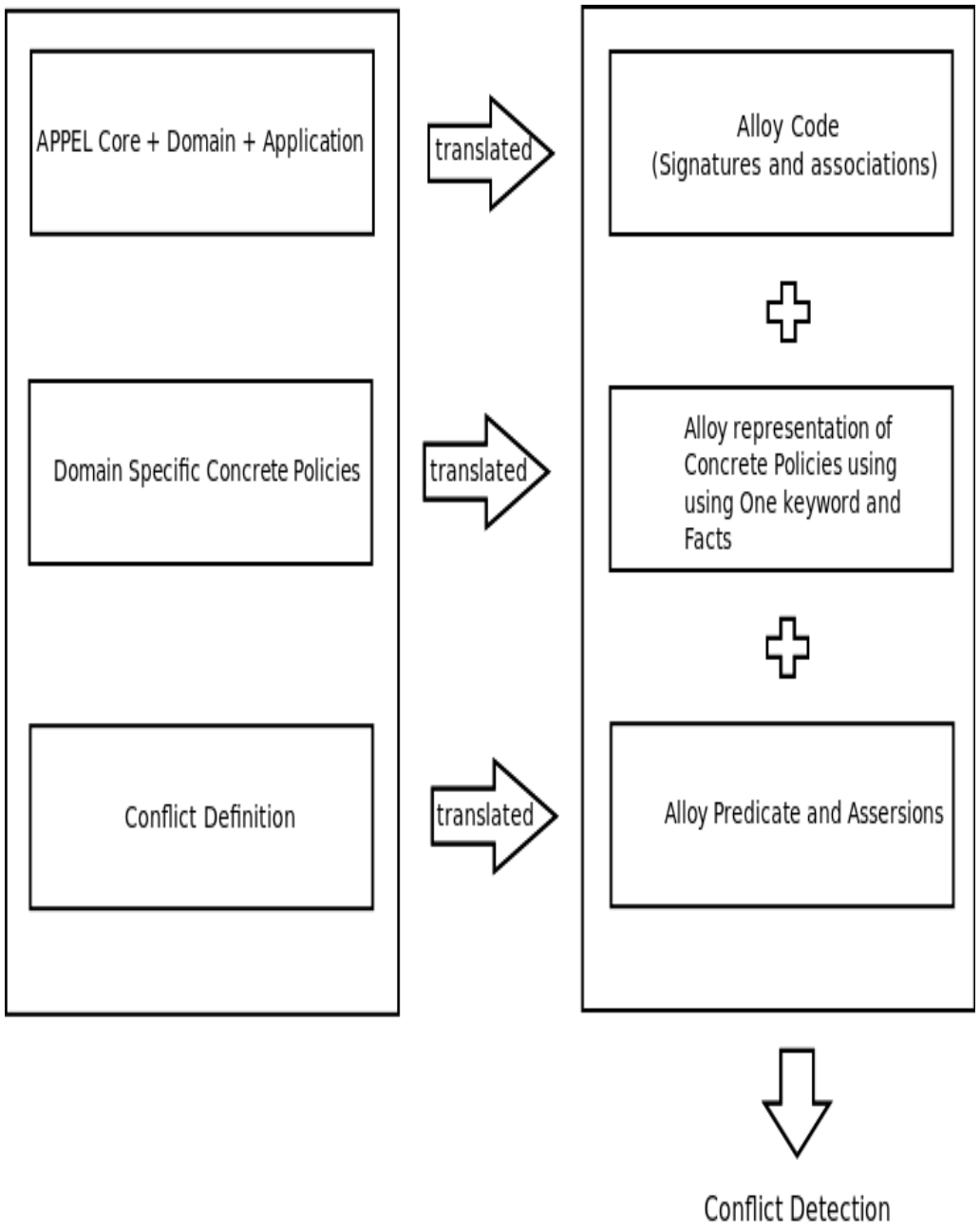


Figure 4.1: Overview of the Conflict Analysis Procedure

As discussed above, the APPEL, STPOWLA and Application meta models are defined by UML class diagrams. The combined Domain and Application Specific Policy Model can be saved in the textual XMI format, allowing for possible automatic conversion into other model languages. In order to make use of the ALLOY model checker for confirmation of conflicts we need to transform or translate the models into ALLOY. We have identified two options for translating/transforming the UML models into ALLOY code.

The first option is the automatic conversion of the UML models to ALLOY code. This transformation can be performed through the UML2Alloy [9] tool. The tool expects a UML class diagram as input, represented in XMI format and produces the equivalent ALLOY model (code). Unfortunately the current tool does not allow to convert all UML class diagrams into ALLOY code, as it has some restrictions. Firstly the tool does not support the transformation of some of the UML associations. Specifically, it does not accept the *aggregation* and *composition* associations that are an integral part of the UML class diagrams we have used. Secondly the tool does not accept and support the transformation of a sub class that is generalized by more than one class (multiple inheritance). Additionally one could not redefine *attributes* and *operations* of that *subclass*. Thirdly the only primitive type supported by UML2Alloy is *integer*, all other data types such as *Boolean or String* are not supported, while we have used the *String* data type in our models. There are some other restrictions, but we have only discussed the issues relevant to our work.

In [38] UML2Alloy is used for automatic conversion of UML models into ALLOY, with an alternative strategy to overcome the restriction discussed above. It is suggested that aggregation and composition associations can be defined using OCL constraints in UML diagram. We cannot use this strategy as our model contains many such associations. As our model uses UML features that are not

currently supported, UML2Alloy is not a suitable approach and therefore we had to rely on a manual approach. Note that the existence of the tool provides a hope that in the future a generic tool can be used to perform the conversion of UML models to ALLOY code.

The manual approach for converting UML to ALLOY is detailed in [60]. Classes are translated into *Signature* declarations in ALLOY, *Attributes* of a class are translated into relations within the corresponding signature. Similarly, associations between classes are also translated into relations. We followed the process of [60] and successfully translated the model into ALLOY. For instance, the following ALLOY code fragment specifies the classes, attributes and associations appearing in the class diagram of the APPEL meta model (see Figure 3.2 in section 3.2.1). The representation of the APPEL core model in ALLOY is shown in Figure 4.2.

```
abstract sig PolicyRuleGroup {
policyrule: some PolicyRule }
abstract sig  PolicyRule {
policy:some  Policy }
sig Policy {
appliesTo: set  Location,
trigger: lone  Trigger,
condition: lone Condition,
action: one  Action
}
```

ALLOY only supports the *integer* data type, while in our models we need to use enumeration of the string data type. Declaring an *abstract signature* with scalar extensions introduces an enumeration using the *one* keyword. An enumeration

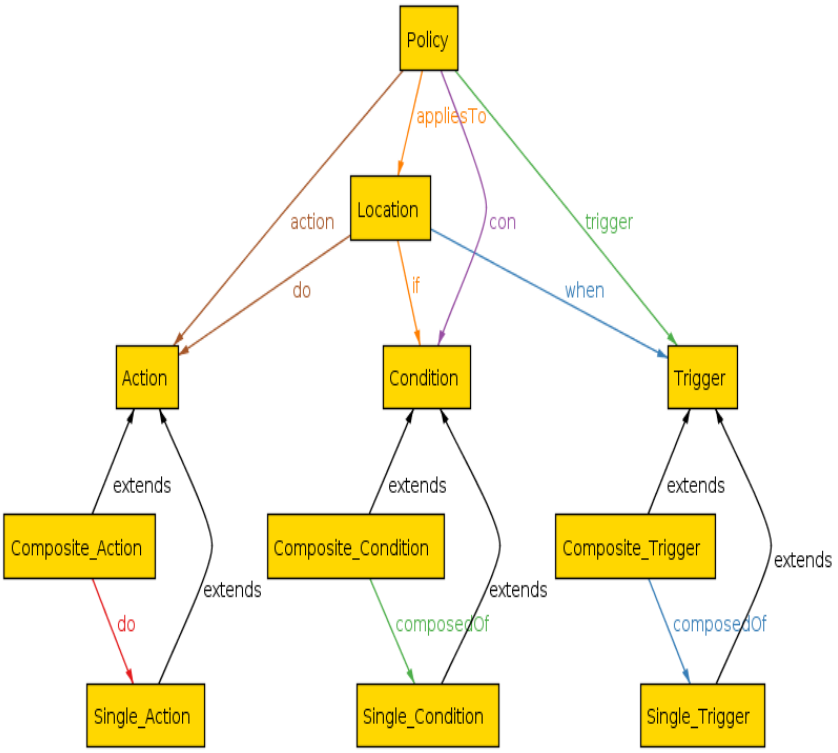


Figure 4.2: Meta Model of APPEL Core in ALLOY

of `ActorRole` is given below, `ActorRole` is an abstract signature, while `clerk`, `branchManager`, `managerRepresentative` are the string values for `ActorRole`.

```
abstract sig ActorRole {}
one sig clerk, manager, branchManager,
        managerRepresentative extends ActorRole {}
```

In STPOWLA a simple policy is written as:

```
Policy P1
appliesTo task1
do insert(X)
```

We discuss the policies and conflict analysis in the next section. Here we only provide an example to show how a concrete policy can be written in ALLOY. The policy P1 in ALLOY is given below. The *signature* is used to define classes. Within the signature definition, *relations* are used to define associations. For example, `do` is an association of `Task` to `Action` and then to `Workflow`. Using the keyword *one*, we have defined P1 as an instance of a signature `Policy`. Figure 4.3 shows the policy P1 instance graphically. In the figure, `Insert` is an `Action` form Task to `Workflow X`. It is not shown connected with both Task and X, but is shown on the association label as **do[insert]**.

```
abstract sig Policy {
appliesTo:disj set Task }
abstract sig WorkflowTask {}
abstract sig Action {}
abstract sig request extends Action {}
abstract sig reconfigure extends request {}
```

```
abstract sig Task {
do:disj set Action->WorkflowTask
}
abstract sig Insert  extends reconfigure{}
one sig p1 extends Policy {}
one sig task1 extends Task {}
one sig insert extends Insert {}
one sig X extends WorkflowTask {}
```

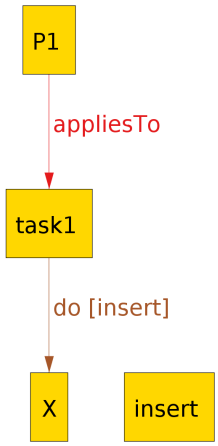


Figure 4.3: Concrete Policy P1

4.4 Conflict Analysis and Confirmation using ALLOY

The ALLOY Analyzer is a first order model checking tool that is used here for automated confirmation of conflicts. Policy conflicts are discussed in the following sections (4.4.1 and 4.4.3), and will be encoded in ALLOY. In the next

section (4.4.4 and 4.4.2) we formally define these conflicts in the form of predicates and assertions, so that they can be confirmed automatically. An assertion is a mechanism for checking and finding counterexamples in an ALLOY model. The properties are stated in the negative so that the Analyser tool can search the state space for counter-examples. For confitmation of a conflict we use ALLOY's small scope concept, that is "if there is a flaw in a system, it can be found by checking small scopes of the system, i.e. considering a small number of instances" [44]. STPOWLA has defined two types of policies. We provide methods for conflict analysis for both policy types.

Policy conflicts can occur when several policies are applicable simultaneously and request for contradicting actions to take place. What exactly a contradicting action is based on domain understanding. For example the conflict definition in the access control domain is different from the conflict definition in the telephony domain. In access control, allowing and denying permission simultaneously on the same resource will be conflict. In the same way, in telephony one policy states to forward a call to mobile if its lunch time, and the caller is a family member, while another policy states not forward call if it is lunch time. From the above scenario we can see that the conflict definition is based on two factors: available knowledge and understanding of the domain. As the definition of conflicting actions is based on domain understanding, the analysis methods rely on what a conflict is in a certain domain. The conflict definition for the STPOWLA domain can be derived by analysis of policy actions. We will discuss the conflicting actions for both types of policies.

4.4.1 Conflicts in Reconfiguration Policy Actions

Policies can express reconfiguration rules based on the available actions discussed in Section 4.1. To analyze reconfiguration actions, four workflow tasks **x, y, a** and

b are considered in a workflow instance to examine conflicts. **z** is a Boolean variable. To understand reconfiguration conflicts, a manual pairwise analysis is taken in consideration. The result is shown in Table 4.2. We analyse these action pairs manually as the definition of conflicts depends on domain understanding. Then we encode them in ALLOY to confirm these conflicts. In Table 4.2, the first action pair **Insert(x,y,z)** and **Delete(y)** are conflicting. If the the action in policy1 is **Delete(y)** and the action in policy2 is **Insert(x,y,z)**, these will conflict because the action **Insert(x,y,z)**, inserts task **x** in the workflow after the task **y**, while **Delete(y)** deletes the task y. This means that the existence of the task **y** is necessary for the action **Insert(x,y,z)**. So if policy1 deletes the task **y**, policy2 could not be applicable.

Conflicts between the policies can be actual or potential conflicts. Actual conflicts exist between the policies, and they will always occur – they do not depend on specific instantiation data, hence they will need to be resolved. For example in Table 4.2 the third action pair is **Delete(y)** and **Delete(y)**, this means that two policies are deleting the task **y**. This is an actual conflict. There are some other factors that need to be considered to determine if an actual conflict occurs. For example, if we change the order of actions in the action pairs discussed above, the policies will not conflict. This would be classified as a potential conflict, as it can be seen from the policies that there is potentially a conflict, but it depends on specific instance situations, e.g., the order of the policies. If policy2 is executed first and inserts task **x** after **y**, then policy1 can delete task **y**. So both policies can be applicable without conflict. This option can also be used when considering resolution for the conflicting policies.

Another factor that needs to be considered is to determine whether the conflict occurs due to the parameters of actions. Some of these actions have parameters, which can themselves cause interaction/conflicts. For example if policy1 says

Action Pair	Insert(x,a,z)	Delete(x)	Delete(y)
Insert(x,y,z)	Y	Y	Y
Insert(a,b,z)	Y(order)	N	N
Delete(x)	Y	Y	N
Delete(y)	N	N	Y
Delete(a)	Y	N	N

Table 4.2: Reconfiguration Functions Pairwise Analysis

Delete(x), i.e. delete task **x**, and policy2 says **Insert(x,y,z)**, i.e, insert a task **x** after **y**, it would lead to a conflict. But again the order of application matters. If policy2 is executed first, they will not conflict.

We will use these results as a domain input to the ALLOY Analyser for the confirmation of these conflicts.

4.4.2 Confirmation of Conflicts in Reconfiguration Policies

The analysis of reconfiguration policy actions was discussed in the previous Section and is shown in Table 4.2. As an example we take a pair of **Insert(x,y,z)** and **Delete(x)** form the table to confirm the conflict. Having policy *P1* trying to **Insert(x,y,z)** and the other policy *P2* trying to **Delete(x)** will lead to a conflict.

The following two example policies are first given in STPOWLA syntax and then we show their ALLOY representation.

```
Policy P1
appliesTo task1
do insert(x,y)

Policy P2
appliesTo task2
```

```
do delete(x)
```

From the following ALLOY representation of policies P1 and P2, we can generate the graphical output shown in Figure 4.4.

```
abstract sig Policy {
appliesTo:disj set Task }
abstract sig WorkflowTask {}
abstract sig Action {}
abstract sig request extends Action {}
abstract sig reconfigure extends request {}
abstract sig Task {
do:disj set Action -> WorkflowTask
}
abstract sig Insert  extends reconfigure{}
abstract sig Delete  extends reconfigure{}
one sig P1, P2 extends Policy {}
one sig task1, task2 extends Task {}
one sig insert extends Insert {}
one sig delete extends Delete {}
one sig X,Y extends WorkflowTask {}
fact {P1.appliesTo=task1 and P2.appliesTo=task2 }
```

To examine the conflict in Policy P1 and P2, we define a predicate and an assertion in ALLOY given as:

```
pred InsertDeleteConflict
{
```

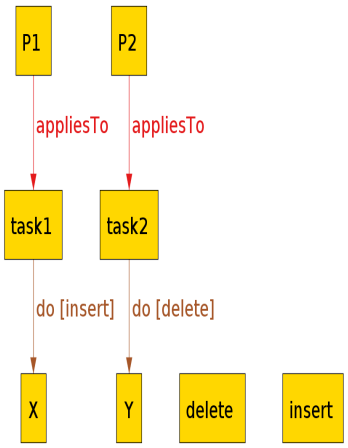


Figure 4.4: Concrete Policy P1 and P2

```
(task1.do=insert-> X and task2.do=delete->X )
or (task2.do=insert-> X and  task1.do=delete->X )
 }
assert conflict {
InsertDeleteConflict [] }
check conflict
```

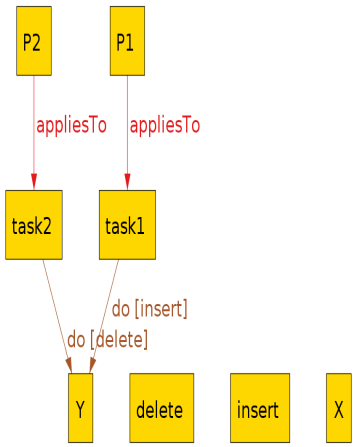


Figure 4.5: P1 and P2 Policy Conflict

Figure 4.5 shows this conflict graphically, where policy **P1** inserts task **X**, while policy **P2** tries to delete the task **X**.

4.4.3 Conflicts in Refinement Policy Actions

For the analysis of refinement policies, a detailed knowledge of the particular application domain is required. The constraints of that particular application are essential to analyse the conflict.

As an example of refinement policies, we consider a case study of a loan approval process discussed in [39]. The workflow of the process in shown in Figure 4.6, consists of five tasks. The first step is to complete and submit a loan application. Next, the submitted application is vetted. An offer will be made next, if the customer is found to be credible. Then the created offer needs to be checked, and finally the approved offer will be sent to the customer, or the application will be rejected, if the offer is not approved [39].

The analysis of refinement conflicts is based on the attributes and policies from the loan approval case study, as conflict definition is based on domain knowledge. [39] defines a number of attributes for the case study, and these are shown in Table 4.3. Some concrete policies are given below, all three policies are applicable to a task *CheckforApproval*.

P1: In a big branch the request should be vetted and approved by two distinct members of staff.
P2: In a small branch the branch manager has to approve all applications.
P3: If the branch manager of a small branch is out of office, his representative signs all applications.

As all three policies are applied to a single task, it is necessary to make them con-

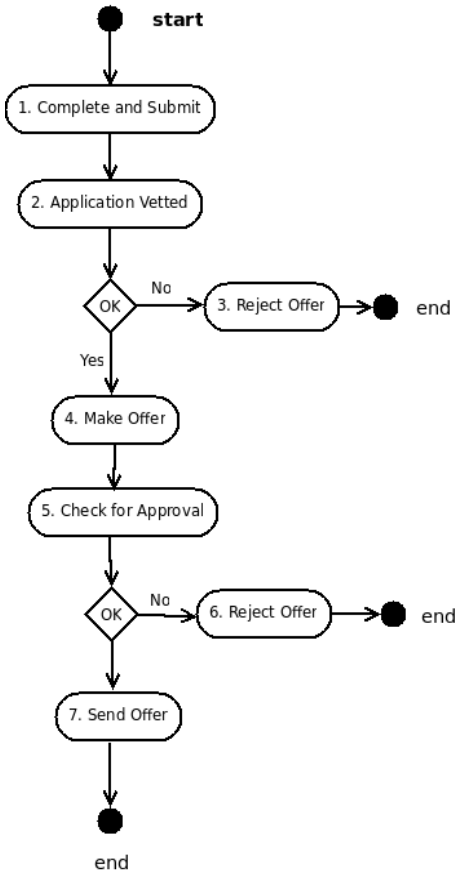


Figure 4.6: Loan Approval Workflow

STPOWLA ATTRIBUTES
SLA dimensions
ActorId = String
Automation = enum{automatic, interactive}
ActorRole = enum{clerk, manager, branchManager, managerRepresentative}
General Attributes
actorId: ActorId
automation: Automation
Domain Dependent Attributes
actorRole: ActorRole
Workflow Attributes
branchSize: enum{small, large}
branchId: String
loanRequest: Integer
Task CompleteAndSubmit
applicantAccountBranch: String

Table 4.3: STPOWLA Attributes of SOA

flict free. Let us consider an informal understanding of conflict in these policies:

1. **Checking inconsistency of Policy P1:** Policy *P1:* It says *In a big branch the request should be vetted and approved by two distinct members of staff.* This policy is checked against inconsistency as two *actor roles* are involved to carry out the task. This policy is only applicable if vetted by two distinct member of staff. For this we will check the attributes *branchsize* and *actorRole*. If the *branchsize* is *large*, the application cannot be vetted by the same *actorRole*, by whom the application was approved.

2. **Conflict Analysis of Policies P2 and P3:** Policy *P2* says *In a small branch*

the branch manager has to approve all applications and Policy *P3* say *If the branch manager of a small branch is out of office, his representative signs all applications.* Policies *P2* and *P3* seem to conflict if both are applicable at the same time. The attribute *branchsize* in both policies needs to be *small*. This means both policies satisfy the condition, while the actions for both policies are different and would lead to a conflict.

4.4.4 Confirmation of Conflict in Refinement Policies

We modelled application specific concepts in ALLOY. The ALLOY code for application specific attributes (defined in Table 4.3) is given below. Figure 4.7 shows the meta model of the application specific concepts, together with the policy.

```
abstract sig ActorRole {}
one sig Clerk, Manager, ManagerRepresentative
            extends ActorRole {}
one sig BranchManager extends ActorRole {
state: one State }
abstract sig State {}
one sig Inoffice, outOfOffice extends State {}
abstract sig Automation {}
one sig Interactive, Automatic extends Automation {}
abstract sig BranchSize {}
one sig Small, Large extends BranchSize {}
```

We analysed refinement policies in the previous section. The analysis is now coded in ALLOY to confirm the conflict.

Conflict for Policy P1: We analysed policy *P1* in previous section, the concrete policy, predicate and assert are given below and graphically shown in Figure

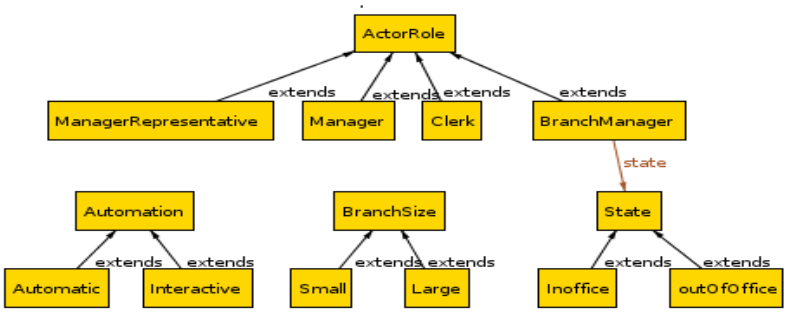


Figure 4.7: Case Study Attributes (ALLOY Model)

4.8:

```
abstract sig Policy {
appliesTo:Task,
do:some Action,
branchsize: Condition->BranchSize }
abstract sig Condition { }
abstract sig Action {
automation: one Interactive, }
abstract sig Refine extends Action { }
abstract sig Task {
actor_role: some ActorRole }
one sig P1 extends Policy { }
one sig CheckforApproval, VetProposal extends Task { }
one sig Req extends Refine {}

pred AppVet[ ] {
P1.branchsize=Condition->Large =>
      VetProposal.actor_role != CheckforApproval.actor_role
}
```

```
assert appvet { AppVet[ ] }
check appvet
```

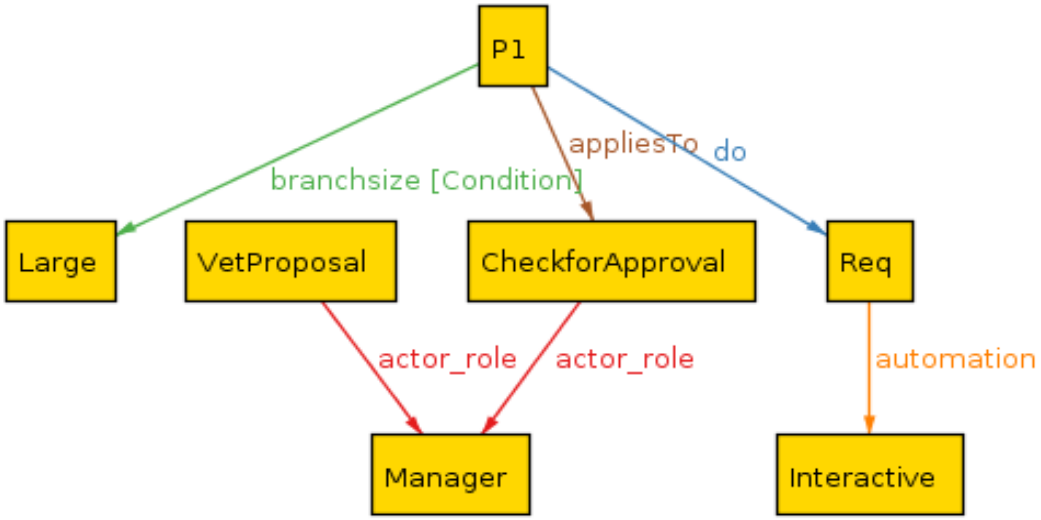


Figure 4.8: Refinement Policy Conflict

Conflict in Policy P2 and P3: We analysed policy P2 and P3 in the previous section. The ALLOY code for these two policies is given below. The *ActorRole* of the policy P2 is *BranchManager* and for the policy P3, it could be *BranchManager* or *ManagerRepresentative*, while *BranchSize* is *Small* in both policies. These condition are defined using **fact**.

```
abstract sig Policy {
appliesTo:Task,
do:disj set Action,
branchsize:one BranchSize
}
abstract sig Action {
automation: one Interactive }
```

```
abstract sig Refine extends Action {}
abstract sig Task {
actor_role: some ActorRole }

one sig P2,P3 extends Policy {}
fact P2P3 { P2.branchsize=Small and P3.branchsize=Small
and (CheckforApproval.actor_role= ManagerRepresentative
   or CheckforApproval.actor_role=BranchManager) }

one sig CheckforApproval extends Task {}
one sig Req extends Refine {}
```

Generally, a predicate is defined to check the conflict between these two policies. Using assert the predicated is checked. A counterexample is shown if the predicate is inconsistent. The ALLOY code for this conflict is given below:

```
pred BranchSizeConflict[ ] {
BranchManager.state=outOfOffice =>
not (CheckforApproval.actor_role=BranchManager)
}
assert BSConflict { BranchSizeConflict[ ] }
check BSConflict
```

The assertion is inconsistent, and a counterexample is shown in Figure 4.9. The two policies P2 and P3 are applied on the task *CheckforApproval*. As both policies satisfy the condition: *Branchsize* is *Small*, their actions are conflicting. The inconsistency can be easily seen in the Figure. The policy P2 is applicable when the Branchsize is *Small*, whereas in case of policy P3, when the status of a *BranchManager* is *outOfOffice*, the *actor_role* should not be *BranchManager*.

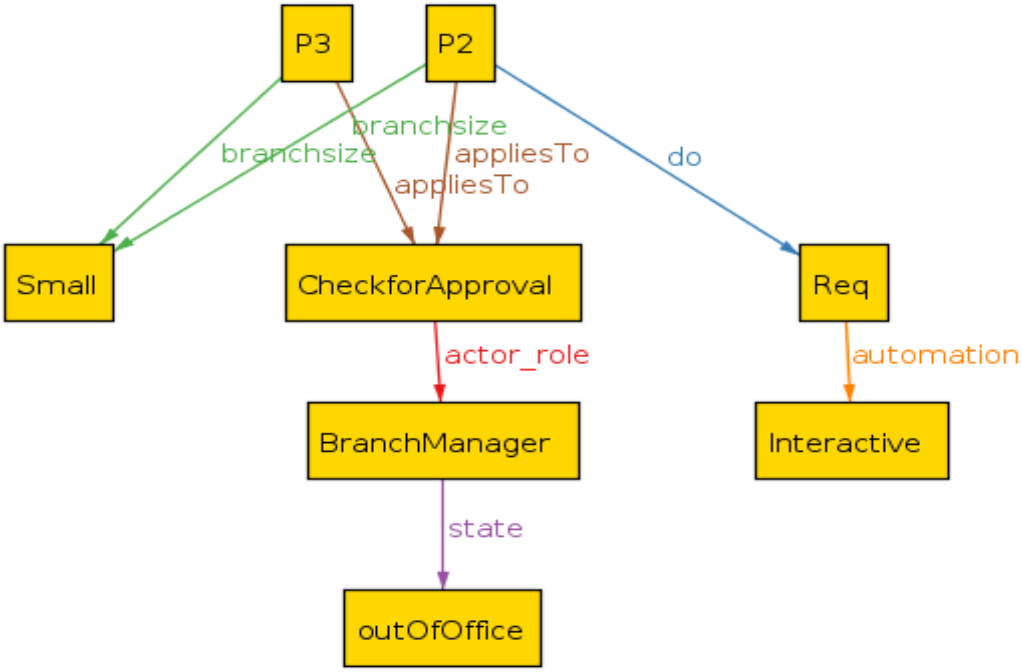


Figure 4.9: Policy2 and Policy3 Conflict

Form this counterexample, we realize that policy P2 is inconsistent with policy P3, so it needs to be modified. We will discuss this in the Section 4.5.

The conflicting policies P2 and P3 are discussed above are defined using the **one** keyword. Till now we have used **one** keyword to instantiate signatures, example the concrete policies P2, P3 and the task *CheckforApproval* are instantiated using the **one** keyword. We will now show another way to instantiate the signatures and then confirm the conflict of the two policies P2 and P3. The signatures can also be instantiate inside the predicate definition. We define the conflicts between P2 and P3 by instantiating the concrete policies inside the predicate. In the predicate we instantiate two policies P2 and P3, two task t1 and t2, and two actor roles ac1 and ac2. We check for conflict between these two policies. The predicate is given below, the counter example graphically shown in Figures 4.10 and Figure 4.11. In Figure 4.10, the *actor role* is *Manager Representative*, but as

both policies are applied the *actor role* are overlapping. In Figure 4.11, the *actor role* is *Branch Manager*, while the status of the *Branch Manager* is *Out of Office*, that is not stated by the policy.

```
pred TConflict[] {
all P2,P3:Policy, t1,t2: CheckforApproval, ac1,ac2:ActorRole
|(ac1=BranchManager and ac2=ManagerRepresentative )
and (P2.appliesTo=t1 and P3.appliesTo= t2) =>
((t1.actor_role!=ac1 and t2.actor_role!=ac2) and (t2.state!=outOfOffice))
 and (t1.branchsize !=Small and t2.branchsize !=Small) }

assert conflict {
TConflict[]
}
check conflict for 2
```

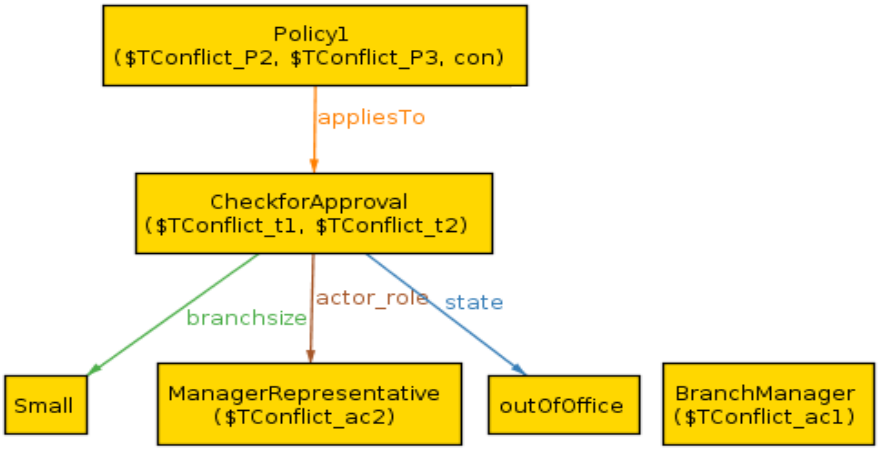


Figure 4.10: P2 and P3 Conflict (1)

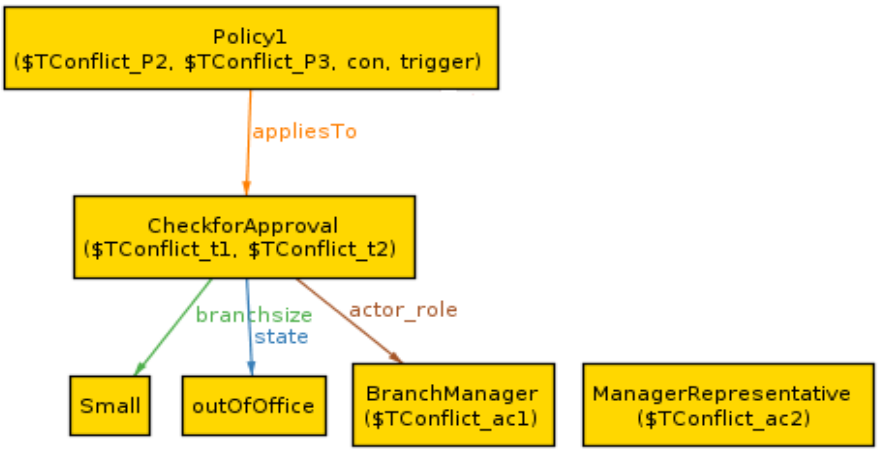


Figure 4.11: P2 and P3 Conflict (2)

4.5 Conflict Resolution

Some resolutions strategies for APPEL policies are already discussed in [69], and we summarized them here. Policy authors are able to specify modalities in the form of preferences such as **should, prefer, must, should not** and their negations. The preference of a policy states how strongly the policy author feels about it. Preferences are optional and can be omitted, this means that the policy author is neutral about this. From strongly positive to strongly negative, the ordering of preference is **must, should, prefer, empty, prefer not, should not, must not**. Principally preference used to automate conflict resolution.

In the previous section we discussed the two types of policies that are defined for STPOWLA, and we show and confirm the conflicts between them. Resolution for these policies can be by defining preferences for the policies or redesigning these policies.

An Example of refinement policy conflict was discussed in Section 4.4.4, where policies P2 and P3 are conflicting over *BranchSize*. A Resolution of this conflict can be redesigning of the policy P2. The policy P2 says that "In a small

branch the branch manager has to approve all applications". From the counter-example, shown in Figure 4.11 and Figure 4.9, we realize that policy P2 is inconsistent and it needs an additional condition. Policy P2 is redesigned by adding an additional condition: if the status of a *BranchManager* is *inOffice*. So the new policy P2 after redesign is: "In a small branch, if a branch manager is in the office, he or she has to approve all applications".

An example of reconfiguration policies was discussed in Section 4.4.2, where policy P1 inserts a task, while policy P2 deletes a task. Resolution for this conflict can be by defining preference *prefer* for the policy P1, so P1 always executes before P2. As policy modalities (must, should) can be used with policies, in case of two conflicting policies that both have the same modality (must in the worst case), resolution is required. We will consider policies with modalities in future work.

4.6 Discussion

ALLOY Analyzer is a well-known and efficient tool. It offers useful features such as counterexample generation and visualization which help the modeler to model and debug the flaws in the model. However there is an issue of scalability with the tool, as the analysis takes some time when the model becomes large. In our work, we do not require to model large number of instances, as we found couter-examples even in smaller scope, but to test scalability we test our model by creating instances of policy. The time taken to test the 12 policy instances was 33ms and for 200 instances it was 721ms. So time gradually increases as the number of instances were increased. Hence depending upon the scope and size of the model, scalability is an issue of the tool.

Chapter 5

Evaluation and Discussion

The aim of this chapter is to evaluate the approach and methodology for both model extension and conflict analysis. In this thesis, we have exemplified our methodology in the STPOWLA domain and its applications as running case study for model extension and conflict analysis. To evaluate the approach in terms of generality and practicality, we apply the methodology to the Home Care domain.

The Home Care case study that we use for comparison is based on the one from the MATCH project [1] (Mobilising Advance Technologies for Care at Home) as published in [86]. In this approach, the behaviour of a care system is specified by policy rules. They specialize a policy language for this purpose. As the purpose of our approach is to define a structural way of policy language extension, we adopt the Home Care case study to validate our approach. The proposed Home Care System has a complete operational architecture, that contains different layers (Policy Server, OSGi, Java, Operating system, etc). Here we focus only on the points related to the policies and specifically the policy language and its specialization.

[1] http://www.match-project.org.uk

5.1 Home Care Domain

This section presents the model of the policy language for the Home Care domain. Home Care systems are used for automated support of care at home. A policy system in Home Care automates support of how a home network should deliver care. The policies for Home Care are expressed in theAPPEL policy language. We model the Home Care domain with UML class diagram, to show how our structural modelling approach specializes the APPEL policy language. To model the Home Care domain, the relevant details of the domain are extracted from available publications [86, 79, 80, 32] and technical reports.

Figure 5.1 shows the model of the Home Care domain. The model consists of `Triggers`, `Conditions`, and `Actions` defined for the Home Care domain. The other important entities of the Home Care model are `sensors, actuators` and a number of `stakeholders. Stakeholders` can be `the user, the family, community nurses, social workers` and `housing wardens. Sensors` include `movement detectors, pressure mats, smoke alarms` and `door switches`. Movement detector sensors capture the movement of a resident in a certain area of the house, such as a bed room, the kitchen, or the lounge etc. A pressure mat sensor is used to determine the occupancy of certain objects such as a bed, sofa, or a chair. Smoke alarm sensors are used to detect an event of fire. Door switch sensors can be used to determine the status of a door such as door closed, open, left open or broken open etc. `Actuators` include `appliance switches, mains supply shutoff, alert messaging` and `lighting control`. Actuators act upon certain condition as specified in policies, e.g, an appliance switch actuators for a TV can have switch on, off or sleep actions. An alert messaging actuator is used for reminder services such as the reminder for taking medicine.

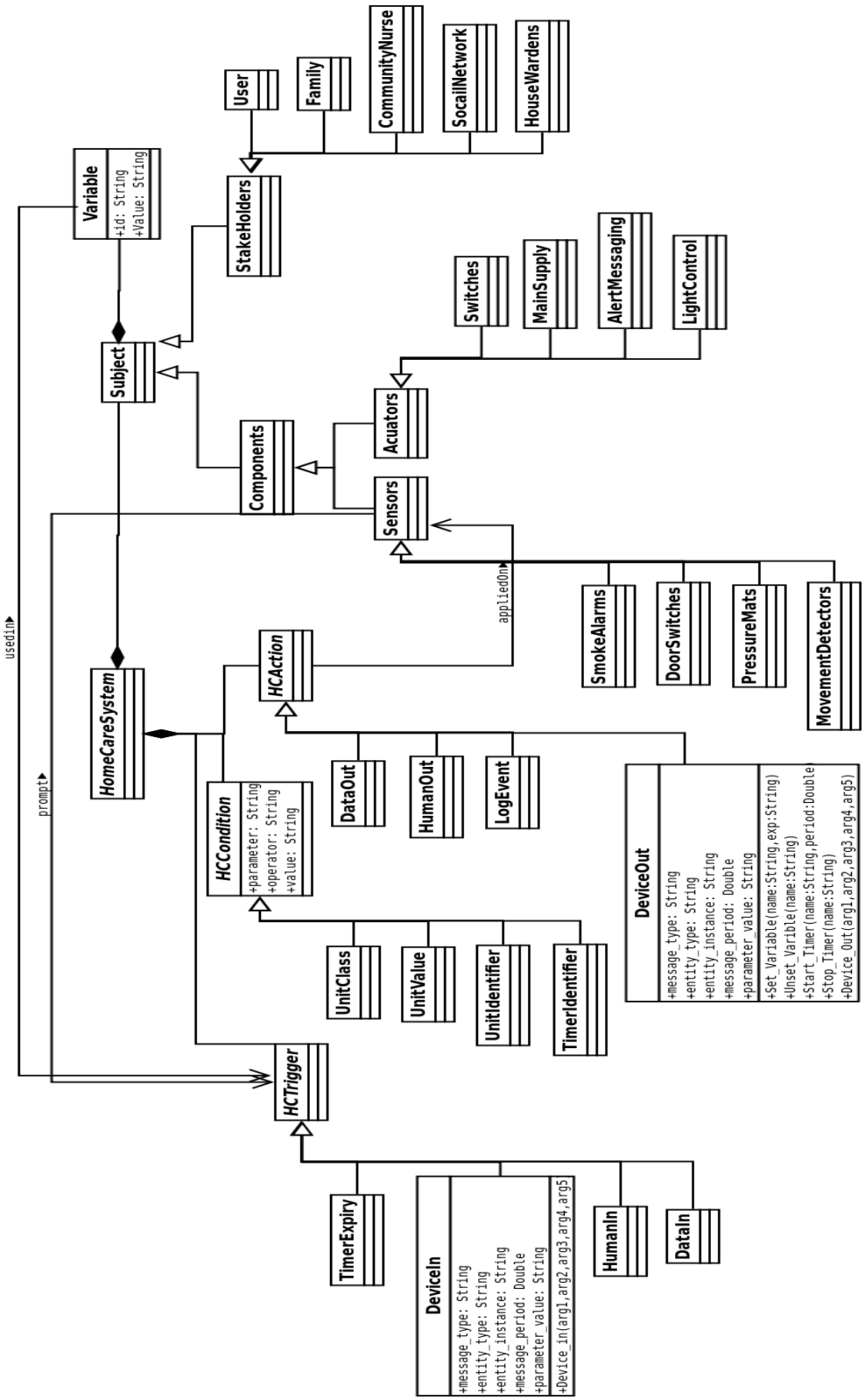


Figure 5.1: Home Care Model

5.1.1 Policy Language Extension for Home Care

We will now apply the structured approach of policy model extension to the Home Care domain. Our approach was discussed in detail in Chapter 3, here we will simply apply the approach. The MATCH project used APPEL as a core language, so we can reuse the APPEL model from section 3.2.4 shown in Figure **??**. Recall that the APPEL core language leaves a gap in that the details of Triggers, Conditions and Actions are undefined. These undefined bits *(formal parameters)* need to be specialized with some specialized Triggers, Conditions and Actions of some domain. For this case study these elements are based on the Home Care domain model.

The Home Care model defined specialized trigger, condition and actions namely (*HCTriggers, HCConditions* and *HCActions*), and they form a straight forward replacement of the *Triggers, Conditions* and *Actions* of the APPEL core language. The other *formal parameter Location* is the main element of the APPEL policy language that defines where a policy actually applies. In home care, policies are applied to *Subject*, so the *Location* is replaced with the *Subject* in the Home Care model.

The transformation process of these two models is straightforward as each *formal parameter* in the APPEL model is replaced by an *actual parameter* in the Home Care model. We define the relevant transformation rules graphically in Figure 5.2, 5.3, 5.4, 5.5.

When applying the structural approach to extend APPEL to the Home Care domain, we realize that the specialization for Home Care needs more than one level of specialization. For example, for an installation of the Home Care model to a new home, some concrete vocabulary needs to be defined that is installation specific. To follow our earlier split in domain and application models, we can say that this is the application level in the Home Care domain. For example the

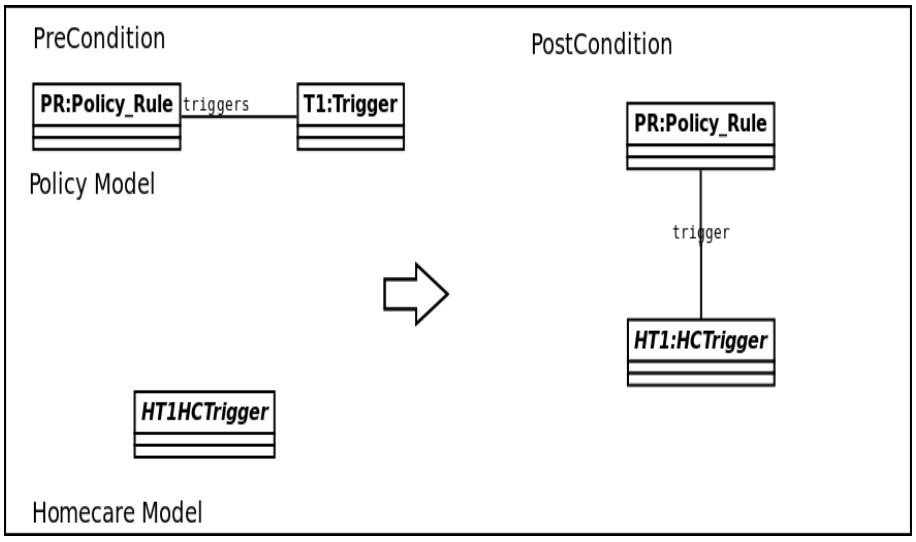


Figure 5.2: Transformation rule for Triggers

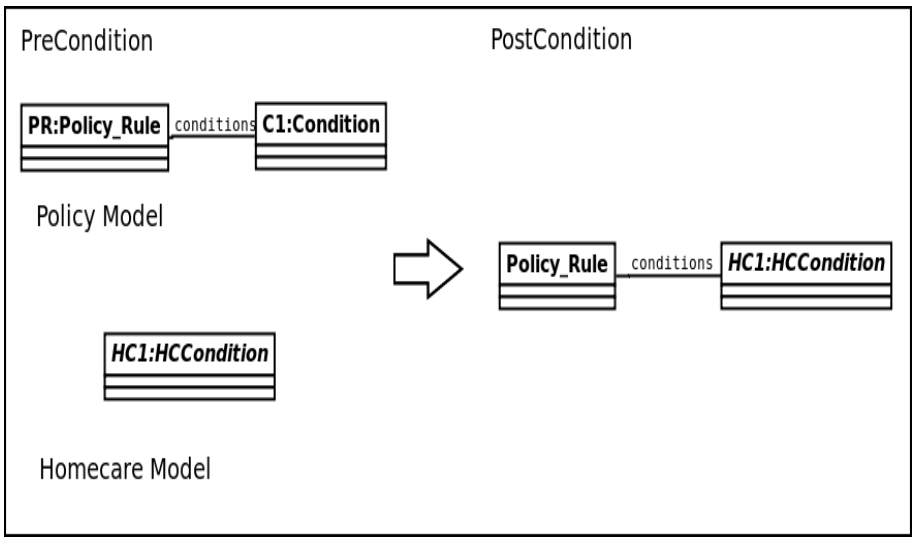


Figure 5.3: Transformation rule for Conditions

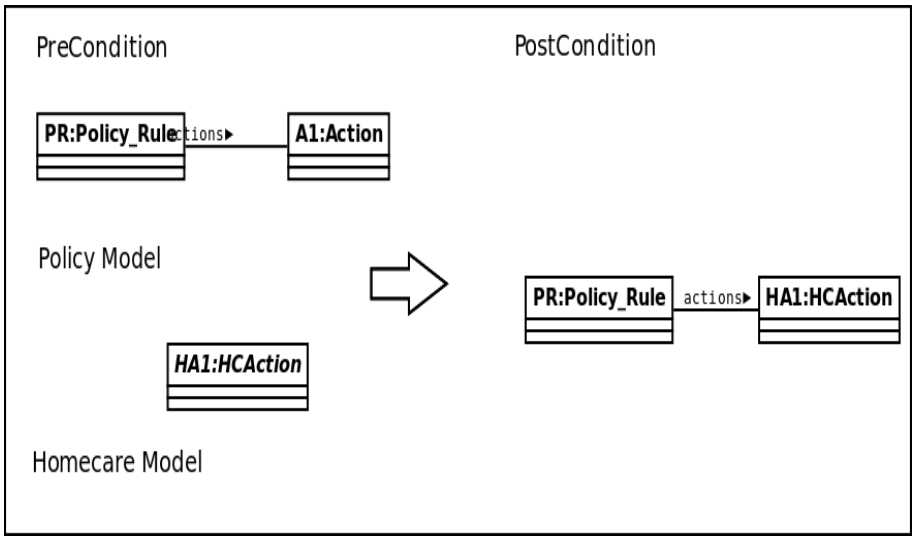


Figure 5.4: Transformation rule for Actions

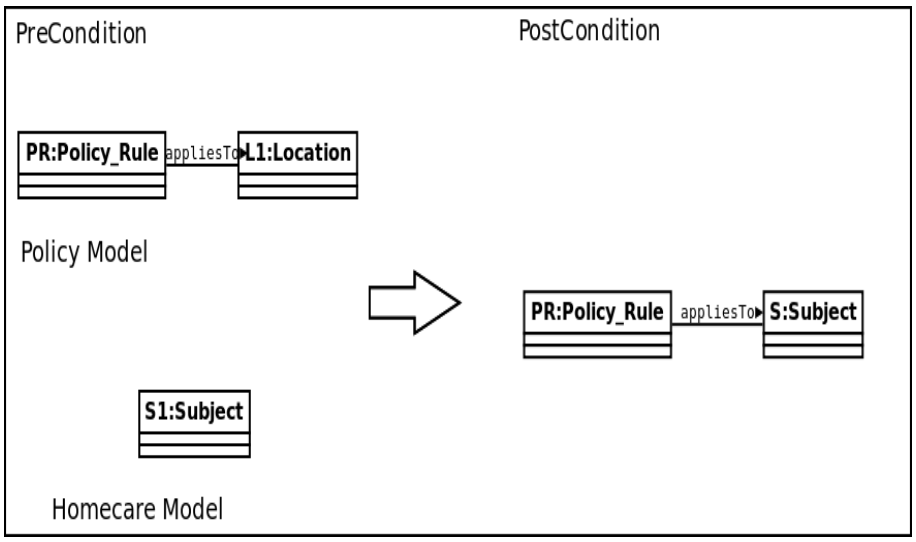


Figure 5.5: Transformation rule for Location

following policy (from [80]) ***Policy1*** says if the user gets out of bed and tries to leave the house during the night, a synthesised reminder should be spoken.

> ***Policy1***: **When** the front door is opend, **If** the hour is
> 11PM-7PM and the user's bed is unoccupied, **Do** remind
> the user to go to bed as it is night time.

We can see in ***Policy1*** that there is specific installation data e.g front door (an instance of a door switch) or the bed (probably with pressure mat). Considering the above example policy, we note that specific elements of the domain model need extension. In Figure 5.6, we only show that elements that are required for this specific policy.

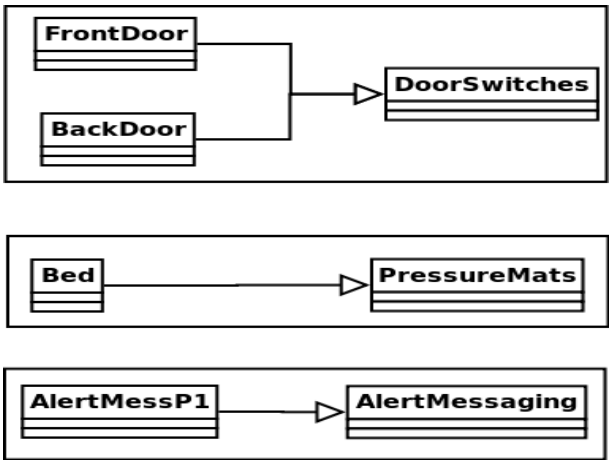


Figure 5.6: Home Care Application Level Specialization

5.1.2 Policy Conflicts in Home Care

The problem of policy conflicts, analysis methods and resolutions are discussed in Chapter 4 in detail. Since definition of a conflict is dependent upon a particular domain, we will analyse the Home Care domain here. Using the APPEL policy language for Home Care, [32] identified three type of conflict in home care system:

- conflicts resulting from apparently separate triggers
- conflicts among policies of multiple stake holders
- conflicts resulting from apparently unrelated actions

In Chapter 4, we showed the model based conflict analysis method. We will use the method here to analyse and confirm the conflicts discussed above. As the Home Care domain is an APPEL extension, it is therefore simpler to provide the analysis method for these conflicts.

In our conflict analysis approach, we have used the ALLOY model checker. We already have a meta model of the APPEL in ALLOY which was shown in Figure 4.2 in Section 4.3, that will be reused here. Hence, we only need to model the Home Care domain concepts in ALLOY. We follow the same approach to model Home Care concepts in ALLOY, that we have discussed and used in section 4.3. The complete Home Care code in ALLOY is given in Appendix A, however in Figure 5.7 we show the Home Care concepts modelled in ALLOY.

Conflicts resulting from apparently separate triggers: For the analysis of policy conflict in Home Care, we choose a situation [32] of conflict that says "a 'door open' sensor can detect the situation of door being open. Suppose a policy states that when the front door is left open, a reminder should be given to the resident to close the door. Combining the door sensor and the sensor in the door lock, a new

Figure 5.7: HomeCare Model in Alloy

situation can be detected: the door has been broken open. Suppose another policy states that, when the door is broken open, the resident should be advised to stay in the room and call for help." The policy examples are given below as ***Policy1*** and ***Policy2:***

Policy1:
When Front Door left open
Do reminder close the door

Policy2:
When Front Door lock broke up
Do advice stay in room to call for help

As we discussed in the previous section (section 5.1.1), by looking at concrete policies new vocabulary is available and we term this *application level*. These *application level* concepts/vocabulary need to be added in Home Care model. This new vocabulary includes `Trigger:DoorBroken, DoorOpen`, `DoorSwitches: FrontDoor`, and `AlerMessaging:DoorCloseReminder, AdviceStayInRoom`. This *application level* ALLOY model is shown in Figure 5.8.

To confirm the conflict in above policies, we analyse the Triggers, Condition and Action of the policies. By looking at the Trigger of ***Policy2*** it seems that ***Policy1*** and ***Policy2*** are applicable at the same time because the door sensors trigger both policies if the door is broken open. As ***Policy2*** is applicable when the door is broken open that is also true in other case as in the event of door left open. In this situation the action of both policies are conflicting with each other. We encode this situation of conflict in ALLOY by using predicate and assertion. ALLOY confirm the between Policy1 and Policy2 as shown in Figure 5.9. The predicate

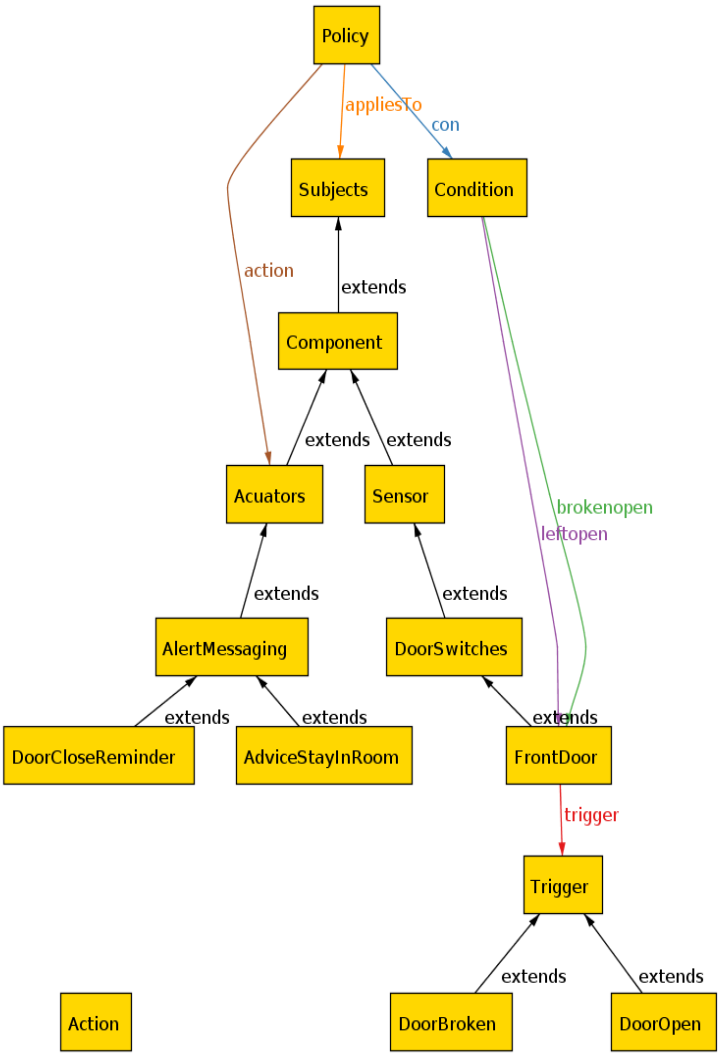


Figure 5.8: Home Care Application Concepts Model in Alloy

and assertion for the conflict are given below:

pred *DoorConflict[]*

{

all *p1,p2:Policy, f:FrontDoor, c:Condition | (p1.appliesTo=f* ***and*** *c.leftopen=f)*

=> p2.appliesTo=f ***and no*** *c.brokenopen*

}

assert *checkDoor* {*DoorConflict[]*}

check *checkDoor*

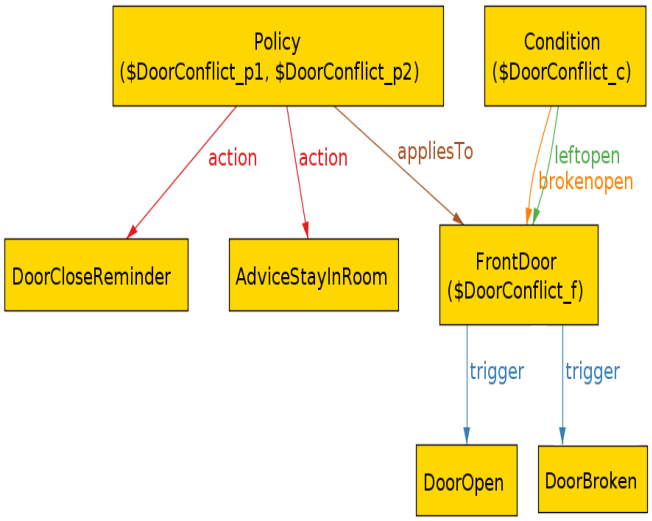


Figure 5.9: Policy1 and Policy2 Conflict on Door Situation

Conflicts among policies of multiple stake holders: In Home Care systems policies can be defined by different stakeholders such as house warden, a tenant, a social worker, or a nurse. These policies can conflict with each other. For example the TV volume in a home can be set by different stakeholders such as a tenant or house warden. Suppose a house warden policy sets TV volume lower between 23:00 to 7:00, whereas a resident or tenant policy sets the volume louder.

As both policies are applied on TV and specifically have same action of setting the volume, they might conflict with each other. These conflicts can be confirmed by the same method we discussed above (by analysing Trigger, Condition and Action). These conflicts can be resolved by setting priorities for stake holders such as a house warden's policy has higher priority than the tenant's policy.

Conflicts resulting from apparently unrelated actions: The policy actions in the Home Care system take time to complete. So there is some possibility that new action might conflict with ongoing actions. For example a medical reminder service reminds a patient to take medicine at certain times. While a medical reminder action is running, an urgent situation such as a fire may be detected in the house following a fire alarm policy. As a action the system will remind the user to leave the house. These actions might conflict with each other. Analysis, confirmation and resolution of these conflicting actions need application specific information and the policy system needs to record all the running actions. As we can see that these conflicting actions need to be confirmed at run time, any static confirmation would not work for these type of conflicts and hence we do not cover here. The author discussed these actions over the time in [32], handling of these conflicts in the policy system is not easy. It is simpler to handle these conflicts in the actuators rather than the policy server. For example if the alarm system is be an actuator and uses a priority based approach to handle actions that conflict over time. A new alarm with a higher priority would stop an existing alarm with lower priority.

5.2 Discussion

The evaluation in this chapter shows how the structural modelling approach can be used to extend the policy language. The Home Care domain is used for the

evaluation of our methodology, as the base policy language for the Home Care domain is APPEL. We have applied both approaches: the extension of a policy language and the conflict analysis for the Home Care domain.

The structural modelling of policy extension brings out several advantages. Firstly, when applying the structural modelling approach to extend the Home Care domain for the APPEL policy language, we have come up with interesting facts about the level of specializations the domain. Initially, it seemed that in the Home Care domain, we have only two levels of specialization. The first is the modelling of APPEL core concepts (presented in Section 3.2.1, hence available for reuse here) and the second level is domain modelling (Home Care here). Actually the structured approach shows that there are three levels. The third level is for a particular home where the Home Care Policy System is installed and used. The concrete policies showed that there are certain concepts and vocabulary that are not available in the Home Care domain model, and these concepts were discovered when examining certain policies (as an instance of installation of the system to the new homes). We term this level the *application level*.

Secondly, after following the structural approach of policy language extension, with the complete model (*Domain Specific Policy Model*), it is simpler to provide model based conflict analysis. It is simpler because all the domain and application specific knowledge is captured in models. As conflicts are analysed manually, we only encoded the relevent conflicts in ALLOY using predicates and assertions.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This thesis presented policy language specialization approach for different domains, designed and engineered using model driven techniques. The modelling approach not only allowed to define specialized policy languages, but also to capture domain knowledge that is then taken into the conflict analysis phase.

We found that policy languages are a typical example of systems where a general core exists, that can be reused while at the same time a strong need to specialize this core for many domains occurs. The domain concepts usually be defined by domain experts and can be captured in domain models. These models can then be used for a number of purposes. In the area of policy languages, the domain models could contain information on conflicting actions and their resolutions, while many basic elements present domain specific actions or triggers that refine the core model.

Model driven techniques are used to provide a structured way of policy language domain specialization. Metamodels are defined, so concepts of a language can be addressed at more abstract levels and will then automatically hold for all

specializations. These models are defined for different levels such as core, domain and application. The APPEL core language defines some elements, while specific elements are normally defined per-domain as they are predefined refinement points that are requiring specialisation. In our models, we formally term these elements formal parameters, that need to be substituted by specialized model entities termed effective parameters. We compose core, domain and application models using parametrize composition method, and defined the transformation rules for each formal and effective parameter. The automatic transformation of these rules are implemented in VIATRA graph transformation tool.

We provide the conflict analysis method that is based on our "structural modelling for domain specialization" approach. Policy conflicts are dependent on domain knowledge, and this knowledge is already captured in models. As domain models contain information on conflicting actions so we utilize this knowledge for conflict analysis. This approach gives us an advantage to use models for analysis while modelling the domain.

The APPEL policy language deals with two types of policies: regular and resolution. Regular policies are used for defining new policy for a system, while resolution policies are used for conflict resolutions. We specialize APPEL for STPOWLA, and the STPOWLA domain deals with two types of regular policies: refinement and reconfiguration. Hence, we have analysed both types of policies to confirm potential and actual conflict. Conflicts in refinement policies are analysed by examining concrete policies, and conflicts in reconfiguration policies are examined by pair wise analysis of reconfigurations actions. The identified conflicting actions are then encoded in the ALLOY model checker for that confirm the existance of conflicts. We also suggested the resolution for conflicting policies. The conflict actions can be resolved by defining preferences for the policies or redesigning of these policies.

The APPEL policy language has already been used for Home Care domain, so we use the Home Care domain to evaluate our work. Using our structural modelling approach, we successfully specialize the APPEL policy with the Home Care domain. The structural modelling for the Home Care domain was useful because, we discovered that Home Care domain need more than one level of specializations, which might be true for some other domains too.

Furthermore, using our conflict analysis approach, we successfully analysed and confirm the actual and potential conflicts in the Home Care domain. Use of the conflict analysis method second time seemed simpler, because the APPEL core was already modelled in ALLOY. We translate the Home Care model into ALLOY and encode predicates and assertions for analysed concrete policies to confirm conflicts.

6.2 Future Work

As future work we are looking at using the generated models as basis for creation of specialised editors. As policies are created by users, it would be useful to provide the user with tools. Also we will use these domain models to provide ontologies required for policy enforcement environments specialised to the respective domains.

We provide conflict analysis methodology, and confirm the existence of conflicts using ALLOY model checker. As a future work we are looking at automatic detection of policy conflicts.

Appendix A

Viatra2 Graph Transformations Rules and Alloy Code

A.1 Viatra2 Definitions of Graph Transformations Rules

The following section illustrates and discusses the graph transformation rules for the policy language extension discussed in Section 3.2.4. The rules are coded in VIATRA2 graph transformation tool.

A.1.1 First Level Specialization

Specialization of a policy language is done in two levels. First level specialize the APPEL policy language with STPOWLA.

```
namespace NewModel;
import MetaModels.PolicyModel;
import MetaModels.DomainModel;
import Models;
```

```
machine MetaTest
{
//************** Rule 1: Policy to Task**********
pattern PolicyRelation(P1,L1,ApTo) =
{
PolicyRule(P1) ;
Location(L1);
relation(ApTo,P1,L1);
}
pattern TaskAttribute(Ts1) =
{
Task(Ts1);
}
gtrule rule1(inout P1, in L1, inout Ts1, inout ApTo)=
{
precondition pattern lhs(P1,L1,Ts1)=

find PolicyRelation(P1,L1,ApTo);
find TaskAttribute(Ts1);
}
postcondition pattern rhs(P1,Ts1,ApTo) =
{
PolicyRule(P1) ;
Task(Ts1);
relation(ApTo,P1,Ts1);
}
action
```

```
{
delete (L1);
println("Replace Location with Task");
}
}
/// Rule 2 Action to Request ************/
pattern ActionRe(P1,A1,Act) =
{
PolicyRule(P1);
Actions(A1) ;
relation(Act,P1,A1);
}
pattern ServiceRequest(R)=
{
Request(R);
}
gtrule rule2(inout P1, in A1, inout Req, inout Act)=
{
precondition pattern lhs(P1,A1,Act,Req)=
{
find ActionRe(P1,A1,Act);
find ServiceRequest(Req);
}
postcondition pattern rhs(P1,Req,Act) =
{
PolicyRule(P1) ;
Request(Req);
```

```
relation(Act,P1,Req);
}
action
{
delete (A1);
println("Replace Request with Actions");
}
}
//*********** Rule 3: Condition to DCondition ****************
pattern RCondition(P1,C1,Con) =
{
PolicyRule(P1);
Conditions(C1) ;
relation(Con,P1,C1);
}
pattern DCondition(DC1) =
{
DCondition(DC1);
}
gtrule rule3(inout P1, in C1, inout DC1, inout Con)=
{
precondition pattern lhs(P1,C1,Con)=
{
find RCondition(P1,C1,Con);
find DCondition(DC1);
}
postcondition pattern rhs(P1,DC1,Con) =
```

```
{
PolicyRule(P1) ;
DCondition(DC1);
relation(Con,P1,DC1);
}
action
{
delete (C1);
println("Replaces Condition with DCondition");
}
}
//**************** Rule 4: Trigger to Dtrigger ********//
pattern RTrigger(P1,T1,Trig) =
{
PolicyRule(P1);
Triggers(T1) ;
relation(Trig,P1,T1);
}
pattern DTrigger(DT1) =
{
DTrigger(DT1);
}
gtrule rule4(inout P1, in T1, inout DT1, inout Trig)=
{
precondition pattern lhs(P1,T1,Trig)=
{
find RTrigger(P1,T1,Trig);
```

```
find  DTrigger(DT1);
}
postcondition pattern rhs(P1,DT1,Trig) =
{
PolicyRule(P1) ;
DTrigger(DT1);
relation(Trig,P1,DT1);
}
action
{
delete (T1);
println("Replaces Trigger with Trigger");
}
}
rule main()=
let P=Models.P1,
L=Models.L1,
ApTo=Models.P1.AppliesTo,
R=Models.Req,
A=Models.A1,
Act=Models.P1.actions,
C=Models.C1,
DT=Models.DT1,
DC=Models.DC1,
Trig=Models.P1.triggers,
T=Models.T1,
Con=Models.P1.conditions,
```

```
Ts=Models.Ts1
in seq
{
choose with apply rule1(P,L,Ts,ApTo) do
choose with apply rule2 (P,A,R,Act);
choose with apply rule3(P,C,DC,Con);
choose with apply rule4(P,T,DT,Trig);
println("All Rules Executed");
}
}
```

A.1.2 Second Level Specialization

Second level specialize the previously specialized model with the Application model.

```
namespace NewModel;
import MetaModels.DomainModel;
import MetaModels.ApplicationModel;
import Models;
machine MetaTest
{
//************** Rule 1: TaskType To RepairServices **********
pattern TaskTpe(Ts1,Tst1) =
{
TaskType(Tst1);
Task(Ts1);
supertypeOf(Ts1,Tst1);
}
```

```
pattern RepSer(RS1) =
{
RepairServices(RS1);
}
gtrule rule1(inout Ts1, in Tst1, inout RS1)=
{
precondition pattern lhs(Ts1,Tst1)=
{
find TaskTpe(Ts1,Tst1);
find RepSer(RS1);
}
postcondition pattern rhs(Ts1,RS1) =
{
Task(Ts1) ;
RepairServices(RS1);
supertypeOf(Ts1,RS1);
}
action
{
delete (Tst1);
println("Replace TaskType with RepairServices");
}
}
//********* Attribute to Vehicle**********
pattern AttTsk(Ts1,At,TAt) =
{
Attributes(At);
```

```
Task(Ts1);
relation(TAt,Ts1,At);
}
pattern Veh(V1) =
{
Vehicle(V1);
}
gtrule rule2(inout Ts1, in At, inout V1, inout TAt)=
{
precondition pattern lhs(Ts1,At,TAt)=
{
find AttTsk(Ts1,At,TAt);
find Veh(V1);
}
postcondition pattern rhs(Ts1,V1,TAt) =
{
Task(Ts1) ;
Vehicle(V1);
relation(TAt,Ts1,V1);
}
action
{
delete (At);
println("Replace Attribute with Vehicle");
}
}
rule main()=
```

```
let Ts1=Models.Ts1,
Tst1=Models.Tst1,
V1=Models.V1,
At=Models.At,
TAt=Models.Ts1.taskAttributes,
RS1=Models.RS1
in seq
choose with apply rule1(Ts1,Tst1,RS1) do
choose with apply rule2(Ts1,At,V1,TAt);
println("All Rules Executed");
}
}
```

A.2 Home Care Domain Modelling in ALLOY

```
abstract sig Subjects {}
abstract sig Component extends Subjects {}
abstract sig StakeHolder extends Subjects {}
abstract sig Sensor extends Component {}
abstract sig Acuators extends Component {}
sig SmokeAlarms extends Sensor {}
sig DoorSwitches extends Sensor {}
sig PressureMats extends Sensor {}
sig MovementDetectors extends Sensor {}
sig swtiches extends Acuators {}
sig MainSupply extends  Acuators {}
sig LightControl extends Acuators {}
```

```
sig AlertMessaging extends Acuators {}
sig User extends StakeHolder {}
sig Family extends StakeHolder {}
sig ComunityNurse extends StakeHolder {}
sig SocailNetwork extends StakeHolder {}
sig HouseWardens extends StakeHolder {}
sig Policy {
appliesTo:some Subjects,
trigger:lone Trigger,
con:lone Condition,
action:some  Action
}
sig Action {}
sig LogEvent extends Action {}
sig DeviceOut extends Action {}
sig HumanOut extends Action {}
sig DataOut extends Action {}
sig Trigger {}
sig TimeExpiry extends Trigger {}
sig HumanIn extends Trigger {}
sig DataIn extends Trigger {}
sig DeviceIn extends Trigger {}
sig Condition {}
sig UnitClass extends Condition {}
sig UnitValue extends Condition {}
sig UnitIdentifier extends Condition {}
sig TimerIdentifier extends Condition {}
```

Bibliography

[1] ”UMC v3.6. Online:”. http://fmt.isti.cnr.it/umc, 2010.

[2] OMG Model Driven Architecture Home Page. http://www.omg.org/mda, Dated: 10-01-2009.

[3] Alloy Analyzer . http://alloy.mit.edu/, Dated: 10-01-2010.

[4] The Viatra-I Model Transformation Framework Users Guide. http://www.eclipse.org/gmt/VIATRA2/doc/viatratut.pdf, Dated: 10-02-2010.

[5] Viatra-2 Model Transformation. http://http://wiki.eclipse.org/VIATRA2, Dated: 10-04-2010.

[6] FUJABA. http://www.fujaba.de/, Dated: 10-06-2010.

[7] OMG Meta Object Facility Specification v2.0, 2002. www.omg.org/cgi-bin/apps/doc?ptc/03-10-04.pdf, Dated: 10-08-2008.

[8] A. Agrawal. GReAT: a metamodel based model transformation language. In *18th IEEE International Conference on Automated Software Engineering, Montreal, Canada*, 2003.

[9] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. UML2Alloy: A challenging model transformation. In *Model Driven Engineering Languages and Systems*, pages 436–450. Springer, 2007.

[10] A. Anderson. An introduction to the web services policy language (WSPL). In *Policies for Distributed Systems and Networks*, volume 4, pages 189–192. IEEE, 2004.

[11] D. Baksi. Model checking of healthcare domain models. In *Computer methods and programs in biomedicine*, volume 96, pages 217–225. Elsevier, 2009.

[12] B. Baudry, F. Fleurey, R. France, and R. Reddy. Exploring the relationship between model composition and model transformation. In *7th International Workshop on Aspect-Oriented Modeling, Montego Bay, Jamaica, Oct. 2nd*, volume 2005.

[13] S. Bensalem, V. Ganesh, Y. Lakhnech, C. Munoz, S. Owre, H. Rueß, J. Rushby, V. Rusu, H. Saıdi, N. Shankar, et al. An overview of SAL. In *Proceedings of the 5th NASA Langley Formal Methods Workshop*, 2000.

[14] L. Bocchi, S. Gorton, and S. Reiff-Marganiec. Engineering service oriented applications: from StPowla processes to SRML models. In *Fundamental approaches to software engineering*, pages 163–178. Springer, 2008.

[15] T. Bray, J. Paoli, C. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (XML) 1.0. volume 6, 2000.

[16] D. Buchs and N. Guelfi. CO-OPN: A concurrent object oriented Petri net approach. In *Proceedings of 12th International Conference on the Application and Theory of Petri Nets, Gjern, Denmark*, volume 18, 1991.

[17] G. Campbell and K. Turner. Policy conflict filtering for call control. In *Proc. 9th Int. Conf. on Feature Interactions in Software and Communications Systems*, pages 83–98. Amsterdam, Netherlands: IOS Press, 2008.

[18] M. Charalambides, P. Flegkas, G. Pavlou, A. Bandara, E. Lupu, A. Russo, N. Dulav, M. Sloman, and J. Rubio-Loyola. Policy conflict analysis for quality of service management. In *Policies for Distributed Systems and Networks*, pages 99–108. IEEE, 2005.

[19] M. Charalambides, P. Flegkas, G. Pavlou, J. Rubio-Loyola, A. Bandara, E. Lupu, A. Russo, M. Sloman, and N. Dulay. Dynamic policy analysis and conflict resolution for diffserv quality of service management. In *Network Operations and Management Symposium*, pages 294–304. IEEE, 2006.

[20] G. Csertán, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varró. Viatra-visual automated transformations for formal verification and validation of UML models. In *17th IEEE International Conference on Automated Software Engineering*, pages 267–270, 2002.

[21] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specification language. In *Lecture Notes in Computer Science*, pages 18–38. Springer, 2001.

[22] S. Davy. *Harnessing Information Models and Ontologies for Policy Conflict Analysis* . PhD thesis, Department of Computing, Mathematics and Physics, Waterford Institute of Technology, Ireland, 2008.

[23] S. Davy, B. Jennings, and J. Strassner. Conflict prevention via model-driven policy refinement. In *Large Scale Management of Distributed Systems*, pages 209–220. Springer, 2006.

[24] S. Davy, B. Jennings, and J. Strassner. Application domain independent policy conflict analysis using information models. In *Proceedings IEEE/IFIP Network Operations and Management Symposium, NOMS*, pages 17–24, 2008.

[25] J. de Albuquerque, H. Krumm, and P. de Geus. Policy modeling and refinement for network security systems. In *6th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2005)*, pages 6–8, 2005.

[26] A. Dersingh, R. Liscano, A. Jost, M. Ahmad, V. Saxena, K. Kurn, M. Baumgarten, M. Mulvenna, K. Greer, and C. Nugent. Context-aware access control using semantic policies. In *Ubiquitous Computing And Communication Journal (UBICC) Special Issue on Autonomic Computing Systems and Applications*, volume 3, pages 19–32, 2008.

[27] N. Dunlop, J. Indulska, and K. Raymond. Dynamic conflict detection in policy-based management systems. In *Proceedings of Enterprise Distributed Object Computing Conference (EDOC02)*, 2002.

[28] N. Dunlop, J. Indulska, and K. Raymond. Methods for conflict resolution in policy-based management systems. In *Enterprise Distributed Object Computing Conference, 2003. Proceedings. Seventh IEEE International*, pages 98–109. IEEE, 2003.

[29] T. Dursun. A generic policy-conflict handling model. In *Computer and Information Sciences-ISCIS 2005*, pages 193–204. Springer, 2005.

[30] H. Ehrig, M. Pfender, and H. Schneider. Graph-grammars: An algebraic approach. In *14th Annual Symposium on Switching and Automata Theory*, pages 167–180. IEEE, 1973.

[31] M. Emerson and J. Sztipanovits. Techniques for metamodel composition. In *6th OOPSLA Workshop on Domain-Specific Modeling (DSM06)*, page 123.

[32] W. Feng and K. TURNER. Policy Conflicts in Home Care Systems. In *Feature Interactions in Software and Communication Systems IX*, page 54. Ios Pr Inc, 2008.

[33] F. Fleurey, B. Baudry, R. France, and S. Ghosh. A generic approach for automatic model composition. In *Models in Software Engineering*, pages 7–15. Springer, 2008.

[34] M. Fowler. *UML Distilled: a brief guide to the standard object modelling language*. Addison-Wesley Professional, 2004.

[35] Z. Fu, S. Wu, H. Huang, K. Loh, F. Gong, I. Baldine, and C. Xu. IPSec/VPN security policy: Correctness, conflict detection, and resolution. In *Lecture notes in computer science*, pages 39–56. Springer, 2001.

[36] S. Gnesi, M. ter Beek, H. Baumeister, M. Hoelzl, C. Moiso, N. Koch, A. Zobel, and M. Alessandrini. D8. 0: Case studies scenario description. In *SENSORIA Deliverables Month*, volume 12, 2006.

[37] S. Godik, A. Anderson, B. Parducci, P. Humenn, and S. Vajjhala. OASIS eXtensible Access Control 2 Markup Language (XACML) 3. 2002.

[38] M. Gogolla and M. Richters. Transformation rules for UML class diagrams. In *The Unified Modeling Language.≪UML≫98: Beyond the Notation*, pages 514–514. Springer, 2004.

[39] S. Gorton, C. Montangero, S. Reiff-Marganiec, and L. Semini. StPowla: SOA, policies and workflows. In *Service-Oriented Computing-ICSOC 2007 Workshops*, pages 351–362. Springer, 2009.

[40] S. Gorton and S. Reiff-Marganiec. Policy-driven business management over web services. In *Integrated Network Management. 10th IFIP/IEEE International Symposium*, pages 721–724. IEEE, 2007.

[41] S. Gorton and S. Reiff-Marganiec. Towards Feature Interactions in Business Processes. In *Feature Interactions in Software and Communication Systems IX*, pages 99–112. IOS Press, 2008.

[42] W. Hassan and L. Logrippo. Governance policies for privacy access control and their interactions. In *Feature Interactions in Telecommunication and Software Systems VIII*, pages 114–130, 2005.

[43] W. Hassan, L. Logrippo, and M. Mankai. Validating access control policies with Alloy. In *Practice and Theory of Access Control Technologies WPTACT'2005*, page 17, 2005.

[44] D. Jackson. Software Abstractions: logic, language, and analysis. MIT Press (MA), 2006.

[45] L. Kagal. The Rein Policy Framework for the Semantic Web, 2006.

[46] B. Kempter and V. Danciu. Generic policy conflict handling using a priori models. In *Lecture notes in computer science*, volume 3775, page 84. Springer, 2005.

[47] Z. Khowaja and S. Reiff-Marganiec. Extending a Policy Language in a Structured way using Model Driven Techniques. In J. Peltonen, editor, *SPLST'09 & NW-MODE'09*, pages 336–341. Tampere University of Technology, 2009.

[48] N. Koch. Automotive case study: UML specification of on road assistance scenario. Technical report, 2007.

[49] D. Kolovos, R. Paige, and F. Polack. Merging models with the Epsilon Merging Language (EML). In *Model Driven Engineering Languages and Systems*, pages 215–229. Springer, 2006.

[50] P. Kumaraguru, L. Cranor, J. Lobo, and S. Calo. A survey of privacy policy languages. In *Proceedings of the 3rd Symposium on Usable Privacy and Security*, volume 48, 2007.

[51] A. Layouni, L. Logrippo, and K. Turner. Conflict detection in call control using first-order logic model checking. In *Feature Interactions in Software and Communication Systems IX*, page 66. Ios Pr Inc, 2008.

[52] A. Ledeczi, G. Nordstrom, G. Karsai, P. Volgyesi, and M. Maroti. On metamodel composition. In *IEEE CCA*, 2001.

[53] E. Lupu and M. Sloman. Conflict analysis for management policies. In *Proceedings of IFIP/IEEE International Symposium on Integrated Network Management (IM1997)*, 1997.

[54] E. Lupu and M. Sloman. Conflicts in policy-based distributed systems management. In *Software Engineering, IEEE Transactions on*, volume 25, pages 852–869. IEEE, 1999.

[55] T. Mens. On the use of graph transformations for model refactoring. In *Generative and transformational techniques in software engineering*, pages 219–257. Springer, 2006.

[56] T. Mens, P. Van Gorp, D. Varró, and G. Karsai. Applying a model transformation taxonomy to graph transformation technology. In *Electronic Notes in Theoretical Computer Science*, volume 152, pages 143–159. Elsevier, 2006.

[57] J. Moffett and M. Sloman. Policy hierarchies for distributed systems management. In *Selected Areas in Communications, IEEE Journal on*, volume 11, pages 1404–1414. IEEE, 2002.

[58] C. Montangero, S. Reiff-Marganiec, and L. Semini. Logic–based detection of conflicts in APPEL policies. In *International Symposium on Fundamentals of Software Engineering*, pages 257–271. Springer, 2007.

[59] C. Montangero, S. Reiff-Marganiec, and L. Semini. Logic-based conflict detection for distributed policies. In *Fundamenta Informaticae*, volume 89, pages 511–538. IOS Press, 2008.

[60] F. Mostefaoui and J. Vachon. Verification of Aspect-UML models using alloy. In *Proceedings of the 10th international workshop on Aspect-oriented modeling*, AOM '07, pages 41–48, New York, NY, USA, 2007. ACM.

[61] A. Muller, O. Caron, B. Carre, and G. Vanwormhoudt. On some properties of parameterized model application. In *Lecture notes in computer science*, volume 3748, page 130. Springer, 2005.

[62] P. Muller, F. Fleurey, and J. Jézéquel. Weaving executability into object-oriented meta-languages. In *Model Driven Engineering Languages and Systems*, pages 264–278. Springer, 2005.

[63] OMG. Meta Object Facility (MOF) 2.0 Core Specification . `http://www.omg.org/docs/ptc/03-10-04.pdf`, 2005.

[64] OMG. Unified Modeling Language (Version 2.2). `http://www.omg.org/spec/UML/2.2/Superstructure/PDF/`, 2009.

[65] L. V. Pedro, V. Amaral, and D. Buchs. Foundations for a domain specific modeling language prototyping environment: A compositional approach. In

Proc. 8th OOPSLA ACM-SIGPLAN Workshop on Domain-Specific Modeling (DSM). University of Jyvskyln, 2008.

[66] I. Ray and M. Toahchoodee. A spatio-temporal role-based access control model. In *Proceedings of the 21st annual IFIP WG 11.3 working conference on Data and applications security*, pages 211–226. Springer-Verlag, 2007.

[67] Y. Reddy, S. Ghosh, R. France, G. Straw, J. Bieman, N. McEachen, E. Song, and G. Georg. Directives for composing aspect-oriented design class models. In *Transactions on Aspect-Oriented Software Development I*, pages 75–105. Springer, 2006.

[68] S. Reiff-Marganiec and K. Turner. Feature interaction in policies. In *Computer Networks*, volume 45, pages 569–584. Elsevier, 2004.

[69] S. Reiff-Marganiec, K. J. Turner, and L. Blair. Appel: The Accent policy environment/language. Technical Report CSM-164, University of Stirling, Jun 2005.

[70] S. Sheidaei. Policy conflict detection using alloy: An explorative study. Master's thesis, Interactive Arts and Technology, Simon Fraser University, 2010.

[71] E. Syukur, S. Loke, and P. Stanski. Methods for policy conflict detection and resolution in pervasive computing environments. In *Policy Management for the Web*, 2005.

[72] G. Taentzer. AGG: A graph transformation environment for modelling and validation of software. In *Applications of Graph Transformations with Industrial Relevance*, pages 446–453. Springer, 2004.

[73] G. Taentzer, K. Ehrig, E. Guerra, J. De Lara, L. Lengyel, T. Levendovszky, U. Prange, D. Varro, and S. Varro-Gyapay. Model transformation by graph transformation: A comparative study. In *Proc. Workshop Model Transformation in Practice, Montego Bay, Jamaica*, 2005.

[74] M. Ter Beek, A. Fantechi, S. Gnesi, and F. Mazzanti. An action/state-based model-checking approach for the analysis of communication protocols for Service-Oriented Applications. In *Proceedings of the 12th international conference on Formal methods for industrial critical systems*, pages 133–148. Springer-Verlag, 2007.

[75] M. Ter Beek, S. Gnesi, C. Montangero, L. Semnini, and P. Isti-Cnr. Detecting policy conflicts by model checking UML state machines. In *Feature Interactions in Software and Communication System X*, pages 59–74. IOS Press, 2009.

[76] M. Toahchoodee and I. Ray. Using alloy to analyse a spatio-temporal access control model supporting delegation. In *Information Security, IET*, volume 3, pages 75–113. IET, 2009.

[77] A. Toninelli, R. Montanari, L. Kagal, and O. Lassila. A semantic context-aware access control framework for secure collaborations in pervasive computing environments. In *The Semantic Web-ISWC*, pages 473–486. Springer, 2006.

[78] G. Tonti, J. Bradshaw, R. Jeffers, R. Montanari, N. Suri, and A. Uszok. Semantic Web languages for policy representation and reasoning: A comparison of KAoS, Rei, and Ponder. In *Lecture notes in computer science*, pages 419–437. Springer, 2003.

[79] K. Turner, G. Campbell, and F. Wang. Policies for sensor networks and home care networks. In *Proc. 7th. Int. Conf. on New Technologies for Distributed Systems*, 2007.

[80] K. Turner, L. Docherty, F. Wang, and G. Campbell. Managing home care networks. In *Networks, 2009. ICN'09. Eighth International Conference on*, pages 354–359. IEEE, 2009.

[81] O. UML. 2.1. 1 Superstructure Specification (formal/2007-02-03). Technical report, Object Management Group. available at www. omg. org 2007, downloaded at May 25th 2009.

[82] D. Varró and A. Pataricza. Generic and meta-transformations for model transformation engineering. In *2004-The Unified Modelling Language*, pages 290–304. Springer.

[83] D. Varró and A. Pataricza. VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. In *Software and Systems Modeling*, volume 2, pages 187–210. Springer, 2003.

[84] D. Varró, G. Varró, and A. Pataricza. Designing the automatic transformation of visual languages. In *Science of Computer Programming*, volume 44, pages 205–227. Elsevier, 2002.

[85] K. Verlaenen, B. De Win, and W. Joosen. Towards simplified specification of policies in different domains. In *Integrated Network Management, 2007. IM'07. 10th IFIP/IEEE International Symposium on*, pages 20–29, 2007.

[86] F. Wang and K. Turner. Towards personalised home care systems. In *Proceedings of the 1st international conference on PErvasive Technologies Related to Assistive Environments*, pages 1–7. ACM, 2008.

[87] M. Wirsing, L. Bocchi, A. Clark, J. Fiadeiro, S. Gilmore, M. Hölzl, N. Koch, P. Mayer, R. Pugliese, and A. Schroeder. Sensoria: Engineering for service-oriented overlay computers. In *At Your Service: Service-Oriented Computing from an EU Perspective*, pages 159–182, 2007.

[88] A. Zito, Z. Diskin, and J. Dingel. Package merge in UML 2: Practice vs. theory. In *Model Driven Engineering Languages and Systems*, pages 185–199. Springer, 2006.