

SOAP-WADI

Specifying Or Analysing Policies With A Decent Interface

Daniel Burrell

June 15, 2010

Abstract

There have recently been several attempts at visual policy analysis and policy specification using natural language. Yet for all the powerful analysis engines these tools may ship with, few of the tools are actually used by the average security administrator in a real working environment. Additionally most of these systems have acted as a front end for a domain-specific analysis system such as just role-based access control, or just file permissions, and there remains much work to make this more general.

Work at imperial has led to the development of a policy analysis system for policies, models and queries expressed in first-order logic. However, as logic is not the language of choice for the everyday user, a simplification of the policy specification interface is needed. The aim of this project is to provide an integrated, extensible set of tools for policy specification and analysis. The emphasis is on a extensible architecture that will accommodate the easy replacement of components. In this document I set out the background, specification, and evaluation criteria for the implementation of such a tool.

Dedicated to Fina and the memory of Mike

Acknowledgments

My greatest thanks to Doctor Robert Craven and Professor Morris Sloman for their commitment to and expert supervision of this project. Your advice throughout the project has been invaluable in helping me develop this system.

I also extend my thanks to Doctor Emil Lupu for his assistance during initial discussions.

Contents

Contents	5
1 Introduction	8
1.1 Motivation	8
1.2 Objective	9
1.3 Requirements	10
1.3.1 Key Requirements	10
1.3.2 Advanced Requirements	11
1.4 User Story	11
1.5 Use Case Diagram	12
2 Related Work	13
2.1 ACCENT Policy Wizard	13
2.1.1 Ontologies	13
2.1.2 Domain Extensibility	13
2.2 SPARCLE	14
2.2.1 Rule guide example	14
2.2.2 Generating Grammars	14
2.3 Policy Authoring	16
2.4 Access control policy analysis and visualization tools for security professionals	17
2.5 Visualization based policy analysis: case study in SELinux	18
2.5.1 Conclusions	19
2.6 Language	20
2.6.1 Policy Types	20
2.6.2 Policy Elements	20
2.6.3 English Policy template	20
2.6.4 Fluents and Predicates	20
2.6.5 Logic Policy Template	21
2.6.6 Consequent Predicates	21
2.6.7 Condition Fluents	23
2.6.8 Sample policy	23
3 Design	25
3.1 Proposed solution outline	25
3.2 Scenario File and relation to UML	25
3.3 System overview.	26
3.3.1 Policy Specification Subsystem	27
3.3.2 Policy Output Subsystem	29
3.3.3 Policy Analysis Subsystem	30
3.4 Tools and Frameworks Used	30
3.4.1 Swing	30
3.4.2 Prolog	30
3.4.3 Interfacing from JAVA	31

4	Architecture	31
4.1	Model View Controller architecture.	31
4.1.1	Model	31
4.1.2	View	31
4.1.3	Controller	32
4.2	EventBus architecture	32
4.3	Observer	33
4.4	Strategy	34
5	Components	35
5.1	Scenario File	35
5.1.1	Headers	35
5.1.2	Body	35
5.1.3	Body-Class	35
5.1.4	Body-Actions	36
5.2	The Importer	37
5.2.1	The XMLLoadAlgorithm	38
5.2.2	The Scenario Model	40
5.3	Variable Factory	40
5.4	Stage1: Class Selection	41
5.4.1	The specification component.	42
5.4.2	The condition component	43
5.5	Stage2: Instantiation	43
5.5.1	The ‘Add As Required’ method	44
5.5.2	The ‘Reference Counting’ method	44
5.5.3	FluentInstance class	46
5.5.4	Instantiating Conditions	47
5.6	Stage3: Binding Fluents	48
5.7	Stage4: Binding Time	51
5.7.1	TimeController	52
5.7.2	ExpressionBuilder	53
5.8	Display	54
6	Testing	56
6.1	Unit Testing	56
6.1.1	Testing Scenario Import System	56
6.2	Beta Testing	56
6.2.1	Beta testing response	57
6.3	Usability Testing	57
6.4	Response to Usability Testing	57
6.5	Summary	57
7	Profiling	58
7.1	Minimum Permissions Consequent	58

8	Evaluation	60
8.1	Comparison With Initial Requirements	60
8.1.1	Completed Requirements	60
8.1.2	Dropped Requirements	61
8.1.3	Additional Features	61
8.2	Usability	61
8.2.1	GUI	61
8.2.2	Learning Curve	62
8.2.3	Inability to edit	62
8.3	Architecture & Design	63
8.3.1	Model View Controller	63
8.3.2	Strategy	64
8.3.3	Extensibility	64
8.4	Choice of Languages & Tools	65
8.4.1	JAVA	65
8.4.2	Swing	65
8.4.3	Event Bus	65
8.5	Expressiveness	66
8.5.1	Capture of Action Relation	67
8.6	Time Criticisms	68
8.6.1	English time descriptives	68
8.7	Future Work	71
8.7.1	Natural Language	71
8.7.2	Visual Policy Design	71
8.7.3	Analysis Functionality	72
8.7.4	Editing	73
8.7.5	Visual Policy Binding	73
8.7.6	GUI Improvements	74
9	Conclusions	76
9.1	Time Management	76
9.2	Appropriate EventBus usage	76
9.3	Policy Specification	77
	References	79

1 Introduction

Policies are rules governing the choices in behavior of a system. They are often used as a means of implementing flexible and adaptive systems for management of Internet services, distributed systems, and security systems [32].

Good computer security is rooted in accurate policies that capture the intentions of their authors; inaccurate policies can lead to denial of service on the one hand and to compromised resources on the other. [30]

In an ideal world of security and management of systems, these good, accurate policies control access to computational resources and fire the actions automatically performed by a system. Recent work within the area has examined the analysis of policies for various properties of correctness and freedom from conflict, to ensure that they are indeed accurate and capture the author's intent. [16]

1.1 Motivation

There have recently been several attempts at visual policy analysis and policy specification using natural language [15], [30], [36]. Yet for all the powerful analysis engines these tools may ship with, few of the tools are actually used by the average security administrator in a real working environment [24] for the reasons listed here:

- Users want tools that closely relate data provided with the tool's output. If it is difficult to see how the two are related, the tool will most likely not be used. [10]
- Users often rely on their own scripts, resorting to bash scripts or other black and white command line tools over sophisticated GUIs for the simple reason that the administrator knows what's in the script, is familiar with the tool, or can tailor it as necessary. [10]
- There is a confidence issue with complex tools, users fear they may be misinterpreting the output of unfamiliar software with respect to what data is being fed in. For example: A graphical user interface that writes configuration files that sometimes don't take effect [10]. (Similar to Nielsen's Software Reliability principles etc.)

Additionally most of these systems have acted as a front end for a domain-specific analysis system such as just role-based access control, or just file permissions, and there remains much work to make this more general. These tools for analysis and specification need to be modular and work round a framework that allows additional modules to be quickly plugged in and out.

1.2 Objective

The aim of this project is to provide an integrated, flexible set of tools for policy specification, analysis, and visualization. The emphasis is on a flexible architecture that will accommodate the easy replacement of components. These may include components to:

- Enable user-friendly specification of policies as well as potential conflicts using application-specific templates;
- Allow various different policy analysis techniques to be used;
- Support graphical visualization of policy conflicts and the proof procedures used to reason about policies
- Control policy learning and simulation.

In this document I set out the background, specification, and evaluation criteria for the implementation such a tool and describe progress made in its realization.

Currently we have a system which will allow us to specify policies and a model on which to check, both these have as their base formalism first-order logic. Since logic is not the language of choice for the everyday user, we require that natural language be used to simplify the input of policies and queries. Specifically we should allow the user to build policies by selected and arranging required elements into some sort of template. The user should be able to specify different kinds of policies as well as different kinds of conditions without being an expert in the base formalism.

We also require the analysis to allow the user to clearly see any conflicts, coverage gaps or redundancies within the policy, as well as trace paths around the policy with a behavioral simulation tool. The diagnosis should be clear and should allow the user to re-specify the policy in a convenient way, and query the system to see if such a modification would solve the new constraint.

1.3 Requirements

I chose a set of requirements which I felt would produce a highly productive and feature rich initial release of SOAPWADI. These requirements were reached from interviews with my project supervisor, comparison with existing projects and features which I believed would provide a more immersive tool.

1.3.1 Key Requirements

Policy Builder Template Build policies from templates. Provide an interface whereby the user can select subjects actions and targets in a structured manner and add or remove conditions as necessary to enable the creation of policies.

Output Logic Output logic policy representation. Provide a display area for the user to view generated logic output.

Permission Policies Support the writing of Permission Policies.

Denial Policies Support the writing of Denial Policies.

Obligation Policies Support the writing of Obligation Policies.

Policy Type Conditions Provide the ability to create conditions based on Permissions, Denials or Obligations.

HoldsAt conditions Support the writing of HoldsAt conditions.

Happens conditions Support the writing of Happens conditions.

Do conditions Support the writing of Do conditions.

Request conditions Support the writing of request conditions.

Request In Between conditions Extend the request functionality to include start and end bounds.

Fulfilled conditions Support the writing of fulfilled conditions.

Violated conditions Support the writing of violated conditions.

Static Predicates conditions Support conditions containing static predicates.

Disk Output The ability to write the output logic to disk.

These basic requirements are what the final solution requires to be a productive tool. The most complicated of these requirements will be the implementation of the specification section which we tackle first.

1.3.2 Advanced Requirements

Policy Comparison The ability to compare two policies to see if they are equivalent or one is subsumed by the other.

Separation of duty conflict analysis The ability to analyse static or dynamic separation of duty.

Modality Conflicts The ability to check for situations such as join authorization and denials, or obligation without permission necessary.

Coverage Gaps The ability to check if no policy exists to dictate the correct response to a request.

These advanced requirements will increase the appeal of SOAPWADI to a wider audience of users by increasing core functionality, improving accessibility to novice users and extending the areas of use for the application from the original specification.

1.4 User Story

From discussions with my supervisor I was able to build a User Story of the typical tasks the software is expected to assist with. People who use the tool would be experts in their domain, but may not be experts with first order logic. We should be able to specify any policy we can express in the underlying engine, however we should not have to deal with first order logic. Once policies have been specified, we should be able to analyse them for different problems, such as coverage gaps and modality conflicts amongst other things. The user should then be able to trace this back to the original policy and made adjustments or refine it as necessary. When they're satisfied with the policy they should be able to save it to disk.

The application would primarily be a prototype proof of concept for use amongst domain experts, but not necessarily academics in the field of logic so it would be acceptable to assume a reasonable level of policy knowledge but not necessarily first order logic.

1.5 Use Case Diagram

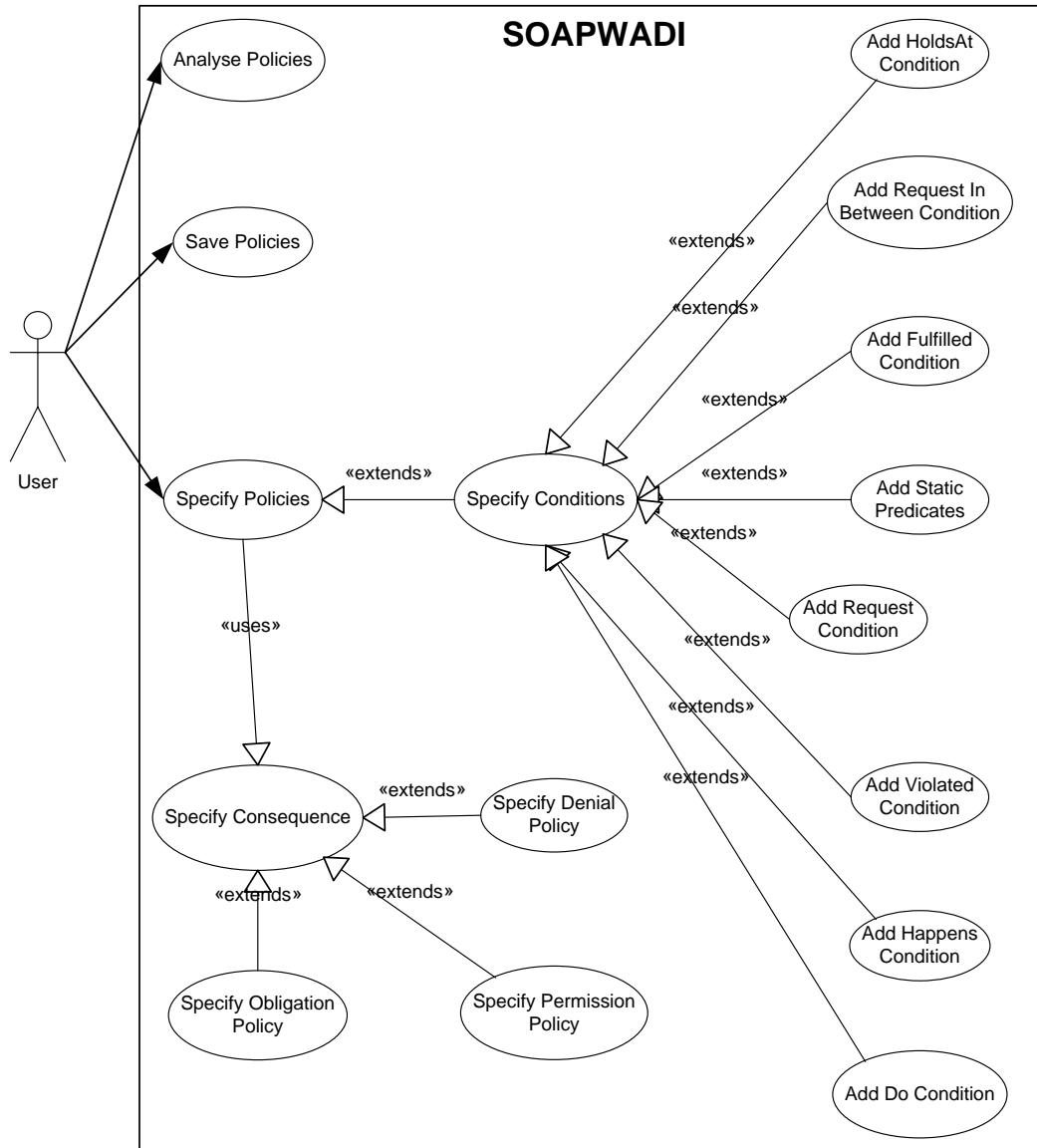


Figure 1: Use case diagram for basic SOAPWADI application modeled in UML

The main functionality and purpose of SOAPWADI should be to enable policy specification. This results in the requirements of having a policy specification template / policy analysis template, the ability to resume sessions at a later date and ultimately to save the policy result in a more standard format such as an XML file.

2 Related Work

There are a number of solutions available that provide parts of our desired solution. I now explore these different solutions and compare their functionality against our requirements at the end in Table 1.

2.1 ACCENT Policy Wizard

The ACCENT (Advanced Component Control Enhancing Network Technologies) Policy Wizard [35] is a web based interface used to allow end users to create policies in XML form. The tool, like our intended tool, attempts to shield the user as much as possible from the underlying XML. The original version facilitated the creation of policies which applied to the areas of telephony and call control, however recent versions are now domain-independent. The wizard provides a great deal of support when defining policies by providing hints for fields and providing tool tip information on certain links. Additionally an on-line help manual is available in the user's preferred language. The wizard has several different views which are available depending on the level of expertise. This allows the system to cater to Beginners, intermediates, experts, and administrators, showing only the minimum of the policy language to a beginner, but revealing the full functionality to an expert. The wizard does not however support 'undo/redo'.

After the user has logged in, the interface provides a series of buttons which enable him to create new or edit existing policies as well as work from a template or select from a policy template repository. POPPET [12] builds a model of the ontology to enable queries from the wizard or other programs, (implemented via JAVA RMI).

2.1.1 Ontologies

The domain independence is achieved through the use of ontologies. The wizard obtains these ontologies from a separate ontology server. Once the wizard has the ontology, it uses the POPPET tool — a neutral interface for retrieval of information held in ontologies.

2.1.2 Domain Extensibility

The wizard can be extended for other application domains, however this is an extensive exercise. However, the authors have outlined the components that need to be created:

1. An ontology for the new domain. The author recommends the use of existing examples provided. There are even ontology editors available [37].
2. Several internal configurations files need to be modified.

We note that although the system does not require expert skill in generating policies, the extensibility certainly requires expert skill. In particular there

seems to be no way of automatically generating the steps required for building a new domain from some High Level Formalism such as a UML diagram.

2.2 SPARCLE

The SPARCLE Policy Workbench [11] tries to find a way of linking the written privacy policies of an organization with the implementation of those policies. It does this by allowing a user to specify privacy policies using natural language. The final policies are then turned into a XACML format for use by an enforcement engine.

Typically this is carried out by following a rule guide, or importing existing text policies and tailoring them using the rule guide, the other method is to use a structured format to define the elements and rule relationships that will be directly used in the machine readable policy and then have SPARCLE generate the natural language for the provided rule.

2.2.1 Rule guide example

```
[User Category(ies)] can [Action(s)] [Data category(ies)] for the
purpose(s) of [Purpose(s)] if [(optional) (condition(s)]
with [(optional)Obligation(s)]
```

As with any tool dealing with natural language, one has to be careful that the intended policy does not get lost in translation. The authors claim figures of 88% to 94% parsing precision, however there is no indication as to which policies were lost in translation. The shallow parser, which provides the “natural language to XACML” functionality, looks like it could be adapted to suit the needs of our project. Since SPARCLE only works for privacy policies, we would only be able to specify and analyze a subset of the policies that our system is capable of.

The shallow parser processes text in a number of stages, beginning with operations that use limited linguistic knowledge to identify syntactic structures such as nouns, noun phrases, verbs, verb groups, and modifying phrases. From this basic speech information the shallow parser then identifies the desired text in a document based on patterns of parts of speech. The SPARCLE group created a set of very specific grammars for identifying the 5 policy elements in each rule: “user categories, actions, data categories, purposes, and conditions/obligations”. Though we don’t deal with ‘purposes’ in our own engine, we can see parallels between the system with ‘actions’, ‘conditions’ and ‘obligations’

2.2.2 Generating Grammars

The SPARCLE team collected samples of privacy policy rules from several organizations across many domains, with the length and level of specificity of each policy varying. On inspecting these, two grammars were generated (see point 2 below). In order to achieve a higher degree of parsing accuracy SPARCLE only parsed constrained natural language rather than completely unconstrained natural language.

1. A policy rule had to be written as a single sentence. This allowed the parser to easily identify the scope of each rule.
2. The policies had to be written in 1 of 2 forms as shown in figures 2 and 3

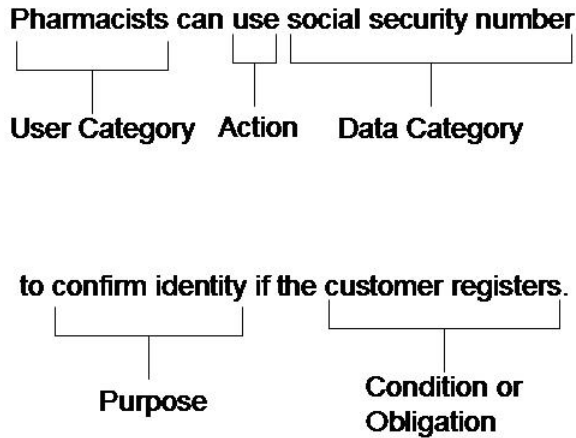


Figure 2: First grammatical structure which SPARCLE accepts

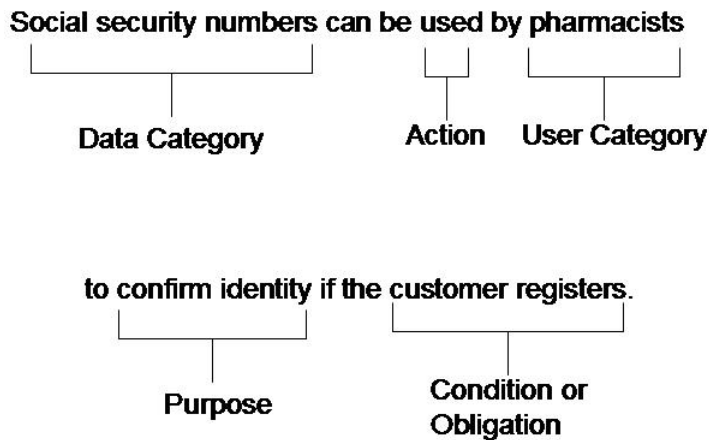


Figure 3: Second grammatical structure which SPARCLE accepts

The SPARCLE group had access to a special information analysis tool (Unstructured Information Management Architecture (UIMA) [22] which was able

to provide future extensibility to the project via natural language parsing capabilities. Despite this, they used a constrained grammar as shown above. It is however unclear as to why they do not achieve 100% parsing accuracy using a constrained language, since if one knows the grammar used, one can expect to achieve 100% accuracy provided the grammar is not ambiguous. If we were to extend the template system in our own project to cover constrained natural language, given the timescale, it would seem that the use of grammars for parsing rather than a sophisticated engine would be more appropriate.

2.3 Policy Authoring

If a user is to be guided, rather than following a set of on screen rules, we should move away from plain text editors to a more user friendly approach to force the user to write the correct policy first time. Looking at the features of any IDE we have the notion of auto-complete and suggestions, additionally we have live syntax error highlighting to indicate when something has been done wrong. Using the grammars similar to those defined in SPARCLE, or else transforming them, by inspecting the model provided we should be able to generate keywords and restrict the path along a sentence to only what should be typed. Indeed studies have shown [38] that the Syntax Highlighting behavior exhibited by an IDE is extremely useful when specifying policies:

1. The purpose of syntax highlighting is to increase the rate at which users can write rules by indicating when something cannot be parsed at design time before the system attempts to process it.
2. Colours can be used to distinguish elements within the policy, (types, actions, subjects etc.)
3. Use of bold, italics and underline can also be used to distinguish between elements in the rule, or subclassify within areas of the rule which are the same colour.

The study claims, however, that being told constantly that the syntax you have provided is incorrect, can be a distraction. To some users it was a minor distraction, while to others it was extremely annoying. Given that one user even wrote out all additions before checking them, this reduces syntax highlighting to its least effective form, where it is used as a post-authoring diagnosis tool, rather than an authoring tool guide. The article suggests that while writing policies, users are either writing or reviewing the policy they have written. Reviewing refers to reading the policy back, and taking in feedback provided to correct the policy to conform to grammar requirements. Generally, feedback should not be provided too early in the process.

The underlying problem here seems to be that the users do not find out what the syntax is until they have attempted to guess it themselves and receive feedback as to whether it is correct or not.

The solution to this would be to revert back to a system of checkboxes where all elements are presented to the user. Indeed this method was preferred by non-native English speakers since everything is presented to them in a format

which is syntactically correct, furthermore they cannot make mistakes with the grammar or values since all these are provided to them, [38]

The disadvantage of this method however, is that it prevents advanced users familiar with the grammar from entering policies quickly, since checking boxes and selecting conditions from a drop-down box may take longer than typing it. [25] The extent to which this time frame becomes an issue depends on the size of the element dictionary; the bigger the dictionary, the longer you can expect the user to spend searching for elements within the list.

A second study in policy authoring with a group of participants who had no background in security or privacy was undertaken by [25]. Their study shows that using Natural Language with a guide or selecting from a structured list are the preferred methods to unstructured natural language, and that the former performs better in terms of quality and user satisfaction. Unstructured privacy policy creation resulted in only 40% of the necessary elements being present in any privacy rules created, however up to 80% of the elements in the scenario were present in the final rules when using a structured list. It's worth noting that the survey which ranks the three authoring methods according to user preference shows that only 3 out of the 19 users from the structured list method would be required to change their opinion in order to swing Natural Language with guide as the first choice. Since their guide was a written guide which the user had to follow and at no point was the user 'forced' to type a certain value via auto-complete, this could suggest that if further steps towards ease of use were implemented (such as auto-complete) then the Natural Language with Guide approach could even be preferred.

A halfway house between these two methods which seems to have been overlooked by the authors of SPARCLE is to make use of auto-complete features similar to that presented in an IDE. This will allow advanced users to type their policies in quickly, whilst ensuring that non-native English speakers have a way of confirming that what they have entered is grammatically correct. The auto-complete works by displaying a list of possible completions to the word the user is typing; this list floats beneath where the user enters their text. By following this approach, the advanced user's typing is not interrupted, whilst a novice is still presented with a list of options; additionally, the novice may start remembering the grammar options over a period of time, as is the case for programmers. It would seem that this method would also conform to the recommendations the paper makes with respect to making a clear and smooth transition between the above three processes, as well as providing a smooth and simple transition for revising a mistake without switching screens (since everything is inline on one page).

2.4 Access control policy analysis and visualization tools for security professionals

The Prisimos tool (a tool proposed by the author, not yet implemented) [36] is designed to provide a visual representation for a given set of policies, as well as the product of any analysis made on those policies. In the authors' example this is specifically conflict analysis. The interface comes in two parts:

1. Specification interface. Lists all the policies together with roles, resources, actions, conditions and purpose. A grid similar to that from expandable grids [30], allows the user to home in on a value by expanding each node and receiving a set of possible values in return, when the bottom level value is reached, a tick in the grid will affirm a value for that part of the policy.
2. Analysis interface. The analysis interface allows the user to focus in on conflicts by listing only the ticked elements in the policy concerned with a particular rules conflict, and then highlighting the specific elements within the rule which conflict.

By using information hiding techniques, large amounts of clutter and noise which are usually present in the specification view are eliminated allowing the user to focus in on only relevant information.

Having seen the interface in 2 halves, where access to resources are specified visually, one wonders whether a design feature for our own tool could be to have the effects read out live. So rather than specify the policy, and ‘compile’ the policy separately, there would be a constant compiling process which would provide the most up-to-date preview of the policy. This would require that the analysis tool be ‘fast’ at analysing, it may be the case that not all (if any) forms analysis we wish to carry out, could be done in the timeframe of a few seconds. Perhaps if processing is a problem we could move processing to a dedicated server with which a dumb client connects, since many enterprises now have access to powerful servers for data analysis.

2.5 Visualization based policy analysis: case study in SELinux

The case study [41] in SELinux discusses an analysis tool, referred to here as the SELPA tool, which is similar to the analysis tool we wish to write for our own Engine tool. The SELinux policy includes over 30,000 statements which makes meaningful analysis very difficult not only because of its sheer size but because administrators must be well versed in the special policy expression language used. Administrators can be expected to carry out deeper analysis where one not only locates policy violations but explores the effects of such violations including its propagation throughout the system, as well as finding the root cause. I now examine this framework in more detail omitting the details on TCB (Trusted Computing Base)

The framework is split into 5 components:

Policy Files A set of files providing policy statements, mappings of the operations between subjects and objects amongst other things.

Policy Parser The policy parser allows a policy graph (the model used for visual analysis) to be compiled by mapping policies into goals.

User Input [Overview Module] allows the user to see the policy graph.

[Content View Module] allows the user to see the policy statements

[Detailed View Module] allows the user to see detailed portions of the policy graph

[Policy Analysis Module] allows the user to perform analysis of and search for policy violation.

Query The query modules allows the specifying, translating, and executing of queries on a policy graph.

[Query Writer GUI] for specifying queries

[Query Translator] translates queries into to path queries for the policy graph.

[Query Executor] executes the query by applying path finding algorithms on the policy graph.

The SELPA tool uses policy graphs as a model for the security policies. Observe that SELinux policy graphs contain 4 node categories: ‘User’, ‘Role’, ‘Domain’ ‘Type’ In order to generate policy layouts for displaying parts of the policy, adjacency-matrices and semantic substrate mechanisms are used [8]; to explain this formalism in detail would take us too far away from the actual work, so we merely refer to the reader to the references above for further information

2.5.1 Conclusions

Feature	SPARCLE	ACCENT	Prisimos	SELPA	This Tool
Ease Of Use	✓	✓	✓	✗	✓
Domain Independence	✗	✓	✗	✗	✓
Analysis	✗	✗	✓	✓	✓
Specification	✓	✓	✓	✗	✓
English->Format	✓	✓	✓	✗	✓

Table 1: A comparison of existing software, each of which provide part of our desired functionality, against the final SOAPWADI release.

The project is looking to create a combination of the functionality these applications. Ideally there should be similar specification functionality of SPARCLE combined with the highlights of the analysis features of the other tools customised for our own analysis needs. Though the ACCENT wizard seems to resemble our intended functionality (excluding the XML representation), it does use a web based interface which is not desirable since the programmers expertise lie in Java and C++ rather than in web based development, furthermore functionality such as undo redo would be easy to code in a language like Java, this is not the case for web based languages such as PHP.

2.6 Language

We now discuss the details of the output language the Specification module tool will produce.

2.6.1 Policy Types

The analysis engine is capable of handling three types of policies, and so we must build a tool which can allow us to express these policy types. The types are:

Permitted Permitted policies express the idea that some element in the policy has the permission to carry out some action at a point in time.

Obligated Obligation policies express the idea that some element in the policy has the obligation to carry out some action within period of time.

Denied Denied policies express the idea that some element in the policy is denied permission to carry out some action at a point in time.

2.6.2 Policy Elements

There are four types of elements in a policy:

Subject The subject is the entity performing the action. Example: “**Bob** can access files before 5pm”

Actions Under an obligation, an action is what must be performed. Under an authorisation, it is what may be performed. Example: “Bob can **access** files before 5pm”

Targets The target is the object on which actions can be performed. Example: “Bob can access **files** before 5pm”

Time The time element is used to express when policies apply, or a range of time under which obligations hold, or events happen in the system. We cover obligations and events in more detail later.

2.6.3 English Policy template

A policy template is the canonical structure we require our policies to adhere to in English. This template is of the form “[Subject] [is permitted] [to perform action] [on target] if [C]*” Note the asterisk indicates 0 or more conditions. Each of these words in [brackets] is sometimes referred to as a ‘slot’ in the template and corresponds to a policy element or a policy type as shown above.

2.6.4 Fluents and Predicates

The analysis engine accepts as input a set of policy files. Each policy file contains a policy written in Prolog notation where fluents are used to represent

policy elements and predicates are used to contain these predicates with respect to time.

In our system, a fluent is an identifier followed by a series of parameters as shown in this example.

$$\text{fluentName}(Parameter1, Parameter2) \quad (1)$$

A predicate takes the similar shape but contains a fluent and a time variable as shown in this example:

$$\text{HoldsAt}(\text{fluentName}(Parameter1, Parameter2), Time) \quad (2)$$

We can use fluents to represent elements within our policies. The following example demonstrates how we would represent the policy elements Project, Project related documentation, and the action transfer.

$$\text{project}(P); \text{projectRelatedDocumentation}(D, P); \text{transfer}(source, destination); \quad (3)$$

We can see that the project fluent contains a single variable P. If another fluent mentions variable P, we can say that the other fluent is associated with that particular Project (as demonstrated in the projectRelatedDocumentation fluent. Effectively this variable P identifies a particular project in a policy with respect to other fluents. Similarly the variable D which appears as the first parameter in projectRelatedDocumentation identifies the projectRelatedDocumentation fluent. We note however that the action transfer does not have a variable to identify it, we will see why this is not necessary later.

2.6.5 Logic Policy Template

Similar to the English policy template, we have the notion of a logic policy template. Policies in our system will take the following form.

$$\text{Consequent} < -[\text{ConditionList}] \quad (4)$$

The consequent is always one of the predicates listed in 2.6.6. The [ConditionList] is a list which contains at least a set of fluents defining the the Subject and Target mentioned in the Consequent. Beyond this, we can have additional conditions which we explain later. Within the consequent, we will have a predicate, this predicate will have parameters to which we assign variables. We sometimes use the word ‘Slot’ instead of ‘parameter’ when we’re dealing with the concept of binding in logic templates, they are however essentially the same concept.

2.6.6 Consequent Predicates

We note the following important predicates and briefly describe the syntax of each one.

Permitted predicate Permitted predicate describe the notion that a Subject can carry out an Action on a Target at a time T and takes the syntax:

$$permitted(Sub, Tar, Act, T) \quad (5)$$

Denied predicate Denied predicate describe the notion that a Subject is denied the ability to carry out an Action on a Target at a time T and takes the syntax:

$$denied(Sub, Tar, Act, T) \quad (6)$$

Obliged predicate Obligated predicate describe the notion that a Subject is obliged to carry out an Action on a Target between Ts and Te, and the policy rule holds at time T and takes the syntax:

$$obliged(Sub, Tar, Act, T_s, T_e, T) \quad (7)$$

The next section covers the above predicates in more detail. The Obligation predicate is linked to two other domain-independent rules: Fulfilled predicate holds when the action which a subject is obliged to perform is executed. The definition for fulfilled is as follows:

$$\begin{aligned} fulfilled(Sub, Tar, Act, T_s, T_e, T) \leftarrow & Obl(Sub, Tar, Act, T_s, T_e, T_{init}), \\ & do(Sub, Tar, Act, T_2), \\ & \neg cease_ob(Sub, Tar, Act, T_{init}, T_s, T_e, T_2), \\ & T_{init} \leq T_s \leq T_2 \leq T_e, T_2 < T. \end{aligned} \quad (8)$$

however we only need to deal with

$$fulfilled(Sub, Tar, Act, T_s, T_e, T) \quad (9)$$

since the former definition is already provided in the analysis engine. Violated predicate holds when the action which a subject is obliged to perform is not executed. The definition for Violated is as follows:

$$\begin{aligned} violated(Sub, Tar, Act, T_s, T_e, T) \leftarrow & Obl(Sub, Tar, Act, T_s, T_e, T_{init}), \\ & notcease_ob(Sub, Tar, Act, T_{init}, T_s, T_e), \\ & T_{init} \leq T_s \leq T_2 \leq T_e, T_2 < T. \end{aligned} \quad (10)$$

similarly we only need to deal with

$$violated(Sub, Tar, Act, T_s, T_e, T) \quad (11)$$

2.6.7 Condition Fluents

The following predicate will always appear as part of the condition, if they appear at all:

HoldsAt The holds at predicate takes a fluent and associated time to say that at a particular time, that fluent holds.

$$holdsAt(fluent(), T) \quad (12)$$

Happens The happens at predicate holds true when a certain event happens at time T. For example, though there are internal events that the policy engine can control, we focus on external events outside of the policies control. For an analogy, let us take the weather as an example. An event might be rains(). If this even happens at time t, then the happens predicate would hold.

$$happens(Event, T) \quad (13)$$

Do The do predicate represents the execution of some Action by a Subject on a Target at a time T.

$$do(Sub, Tar, Act, T) \quad (14)$$

Request The request predicate holds if a request for a subject to carry out an action of a target was made at time T.

$$req(Sub, Tar, Act, T) \quad (15)$$

RequestIB The request in between predicate builds on the request predicate by holding if a request for a subject to carry out an action of a target was made between time T_1 and T_2

$$reqInBetween(Sub, Tar, Act, T_1, T_2) \quad (16)$$

Static Fluents Static predicates are other custom predicates that do not evolve with time. They are always part of conditions.

2.6.8 Sample policy

Let's now look at an example policy.

Project Managers can transfer Project Related Documentation from One stakeholder to another if the destination stakeholder has permission to read the Project Related Documentation.

The final output for this policy would look something like this:

$$\begin{aligned}
& \text{Permitted}(\text{Sub}, \text{Tar}, \text{transfer}(\text{Source}, \text{Target}), T) : - \\
& \text{HoldsAt}(\text{projectManager}(\text{Sub}, P), T), \text{HoldsAt}(\text{project}(P), T), \\
& \quad \text{HoldsAt}(\text{stakeholds}(\text{Source}, P), T), \\
& \quad \text{HoldsAt}(\text{stakeholds}(\text{Destination}, P), T), \\
& \text{HoldsAt}(\text{projecRelatedDocumentation}(\text{Tar}, P), T), \\
& \quad \text{permitted}(\text{Destination}, \text{Tar}, \text{read}, T);
\end{aligned} \tag{17}$$

We see that the time throughout the policy is the same. Note also that the fluents in the condition such as `projectManager` have been made the subject by taking the variable `Sub` and placing in both the `Permitted` predicate and the `projectManager` fluent. This process of associating two expressions in a policy is known as `Binding`.

Explicit element An explicit element is an element that is explicitly mentioned in the English (canonical/template) form. For example in the above policy, ‘Project Managers’ and ‘Project Related Documentation’ are examples of explicit elements since they appear explicitly in the policy.

Peripheral/Implicit element A peripheral element (sometimes called an implicit element) is an element that is required in the logic representation but is not explicitly mentioned in the English template. For example ‘Project’ is a peripheral or implicit element, since at no point in the policy template do we mention a project, even though we understand that the project manager and the project related documentation are bound by the same project. When we talk about explicit or peripheral elements, we usually refer to their logical representation and not their English representation.

3 Design

In this section I explain the overall system architecture and design. This requires giving an overview of the various subsystems, however these will be explained in greater detail later in the report.

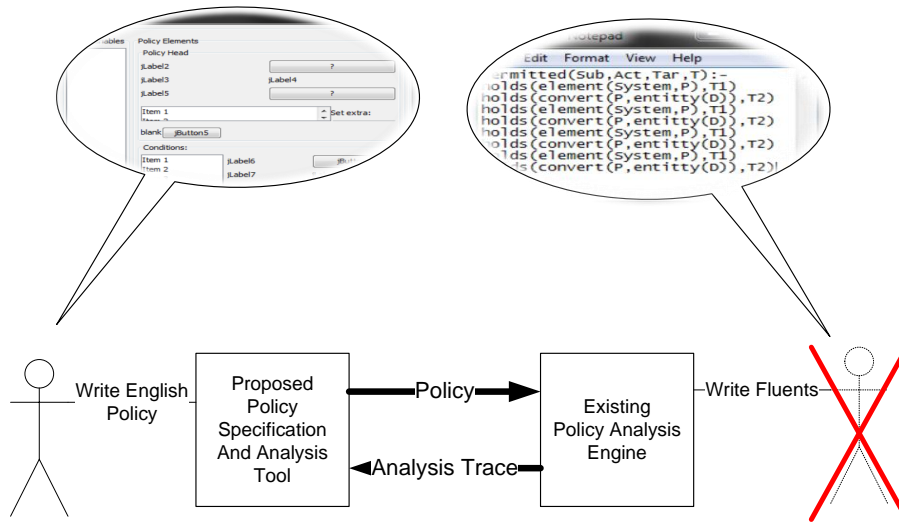


Figure 4: Diagram representing the intended outcome of the system.

Figure 4 describes the problem from the user interaction point of view. A single user must be able to specify policies in an English-like template via a GUI without having to deal with the intricacies of a first-order-logic based language such as the one used with the analysis engine.

3.1 Proposed solution outline

The user loads in a description of the system on which policies are to be defined. The user specifies policies by selecting from a set of elements laid out in a template. Initially the user sketches the policy shape, then via a series of stages, the user iteratively refines the policy by providing more information via the binding of time elements and cross associating elements within the policy, until a policy is finally generated. The user is free to save or analyse these policies further. We should avoid allowing the user interact directly with the fluents or underlying logic as much as possible.

3.2 Scenario File and relation to UML

In order to test and analyse policies, there needs to be a representation of the domain to which they apply. This high level formalism could manifest itself, for example, as a UML diagram, though it could take other forms and we should not assume that a UML diagram specifically will always be available. If we assume that the ability to import a domain model from a UML diagram is necessary, then to avoid being arbitrary about our format choices, it is also

necessary to cope with many other HLFs. Our project has a time limit and we would be encumbered with the decision of selecting a particular UML file format to accept. It is therefore impractical to try to handle all HLFs, so we offer an interface instead: A Scenario File. The scenario file describes the domain model in our own clearly documented manner, and only comprises the data and structures necessary to build policies. By doing this we can keep at bay the development of tools to convert or combine HLFs to our format which may be complex in nature. To this end, the scenario file contains a description of the domain in the form of a structured list of all elements in the system (Subjects Actions etc). It defines both an English and logical representation for each including the number of parameters, actions etc... We cover the exact implementation of such a file later.

3.3 System overview.

Overview of the components of the system.

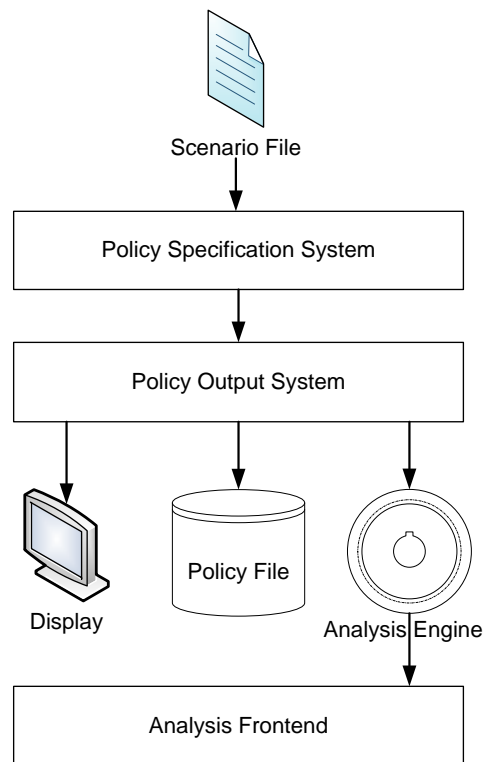


Figure 5: A high level representation of how the 3 main proposed subsystems interact

Figure 5 shows how the three main components of the proposed system interact with the three main outputs. The policy specification subsystem allows the user to specify policies in template format containing English elements based on the definitions provided in the scenario file. The Policy output system displays the current policy, the set of all policies created in the current session and enables the submission of these policies to an analysis engine or to disk for manual loading into an analysis engine. The analysis system receives a trace output

from the analysis engine, and displays this trace along with the conclusion of the analysis. (e.g. for conflict analysis ‘conflict found’ or ‘no conflicts’). We now make a closer examination of the design of each subsystem.

3.3.1 Policy Specification Subsystem

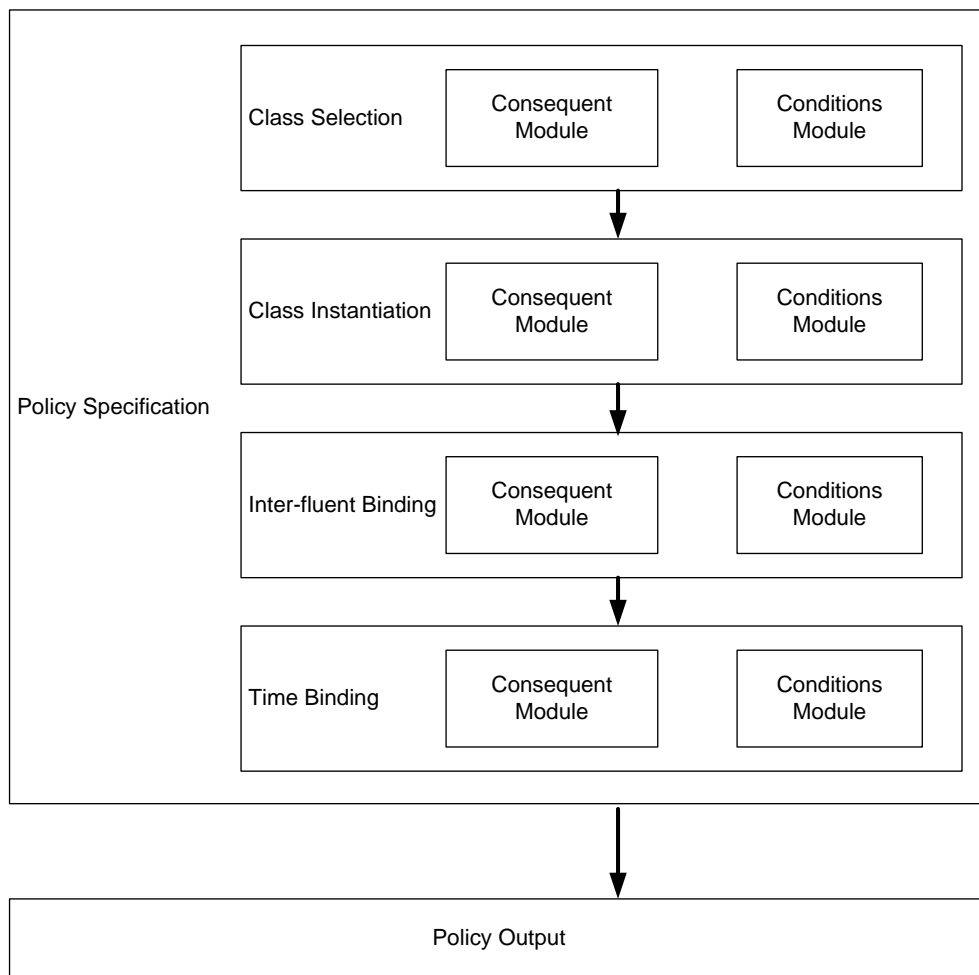


Figure 6: The Policy Specification subsystem overview and its internals.

The policy specification subsystem is built of 5 internal layers shown in Figure 6. Each contains a ‘consequent’ module for manipulating the (English) consequent aspects of a policy, and a ‘conditions’ module for manipulating (English) conditions of the policy. We manipulate these aspects at every layer, which I now describe in detail:

Blueprint Selection: Blueprint selection is a process where the user implicitly defines the overall shape of the policy in terms of how many conditions (if any) there are, the types of conditions (section 2.6.6), as well as explicitly defining the Policy Types (section 2.6.1), and the specific class of the element to be used.

A **blueprint** object is an object which internally describes the fluent or predicate structure, (e.g. the name, arity, parameter types etc...) while externally providing an English representation for display in a template.

The underlying system does not create any objects to represent fluents at this stage, instead it creates blueprint objects for instantiating fluents as part of the next stage. This is a finite process whose length depends on the complexity of the policy since the number of element classes to be selected depends upon the length of the policy being built.

Blueprint Instantiation: Blueprint instantiation is a process where the user instantiates fluents or predicates based on the blueprints selected in the previous stage. The instantiation result is a fluent or predicate representing a policy element along with a unique identifying variable where necessary. We count the number of references made to a particular element in the previous stage, and use this to evaluate the number of element instances required for this stage. The method of distinction between instances is not important at this stage (we can use colour, or append a number after the instance name). Enabling the user to select a particular instantiation is a very good way to insulate the user from dealing with the underlying logic since the user does not deal with ‘variables’, only with distinct instances.

*Scenario: Assume a room with 2 laptops. Both laptops draw 5A of current. The room’s mains fuse box is currently fitted with a 6A fuse. Clearly to avoid blowing a fuse **only 1 laptop may be switched on at any one time**. Additionally a laptop can be in 1 of 2 states: On or Off, and we would like to express that **a laptop can only be switched on, if it was previous switched off**. To express the first policy, in stage 1 would say: ‘A laptop is permitted to be switched on if A laptop is off’. In stage 2 we would set the first reference to the laptop to be instance 1, and the second reference to the laptop to be instance 2. Later if we specify that instance 1 is not the same as instance 2, then we have essentially expressed that ‘A laptop is permitted to be switched on, if another laptop is switched off’. To express the second policy, stage 1 would look identical, however, in stage 2 we would select instance 1 for both the first and second reference. This would allow us to express ‘A laptop is permitted to be switched on, if said laptop is currently off’.*

Inter-fluent Binding: Inter-fluent binding allows the user to bind existing or peripheral fluents (section 2.6.8) to the current policy by selecting existing fluents or instantiating new ones. The user is forced via a type filter to select from or create only relevant fluents, that is they can create fluents for which there are gaps in the policy - implicit elements. The following example makes it clear that this process is necessary to allow peripheral elements to be added.

Scenario: Take this policy: ‘A project manager may transfer project related documentation to a customer’. In this policy, the elements are ‘Project Manager’, ‘Transfer’, ‘Project related documentation’, ‘Customer’.

Implicitly using human intelligence, we automatically assume a relationship between a project and the documentation, the project and the manager. That is, on first glance we infer that throughout the policy, there is a single implicit project to which the manager is presumably related to. This again may not be the case since the manager may be transferring somebody else's documentation pertaining to another project other than his own. We cannot be explicitly clear about this policy without mentioning the project, and there was no opportunity to include a project when building the policy (since the template does not accommodate it) hence the need for this step.

Time Binding: Time binding allows the user to introduce temporal factors into the policy. This is a requirement of the system since the underlying language supports time elements. The user can bind time variables to indicate when the policy holds and to each condition in the policy to stipulate when it should hold. In the case of Obligations we require that the user state the start and end time of the obligation as well. The user can provide any constraints on the time variables using standard operators $<, =, >, \leq, \geq$.

3.3.2 Policy Output Subsystem

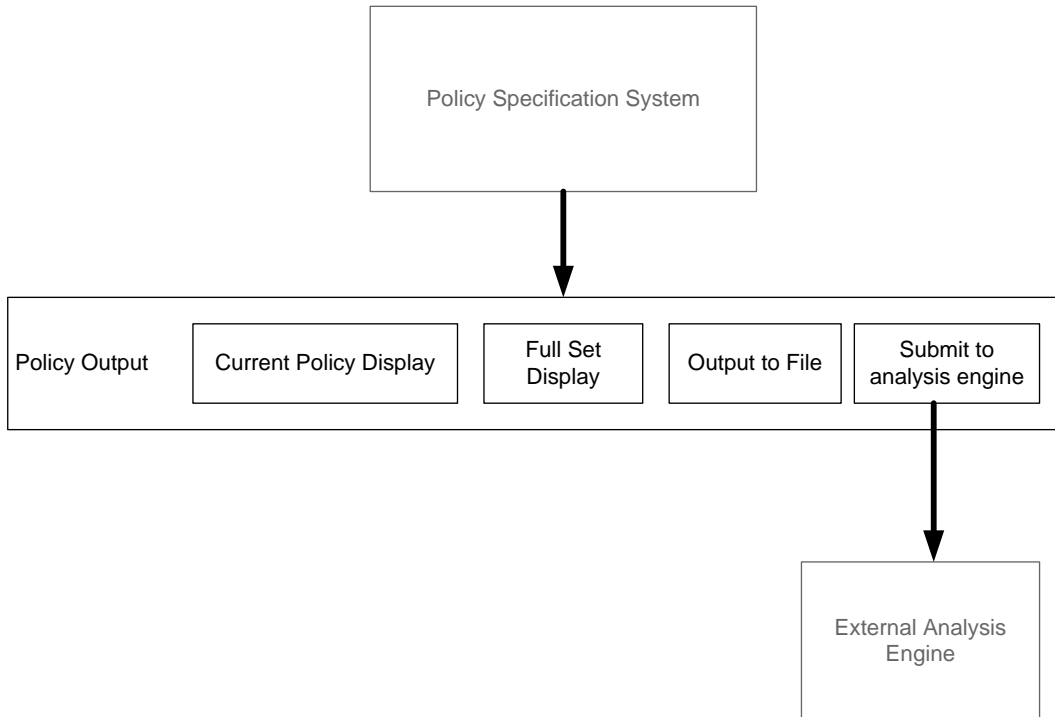


Figure 7: The Policy Output subsystem overview and its internals.

The policy output subsystem receives the final policy model from the policy specification subsystem and stores it in memory together with all other policies

present in the current session. It provides the components necessary to display both the current policy and other session policies on screen, as well as the ability to output the file to disk or to submit all session policies to an analysis engine.

3.3.3 Policy Analysis Subsystem

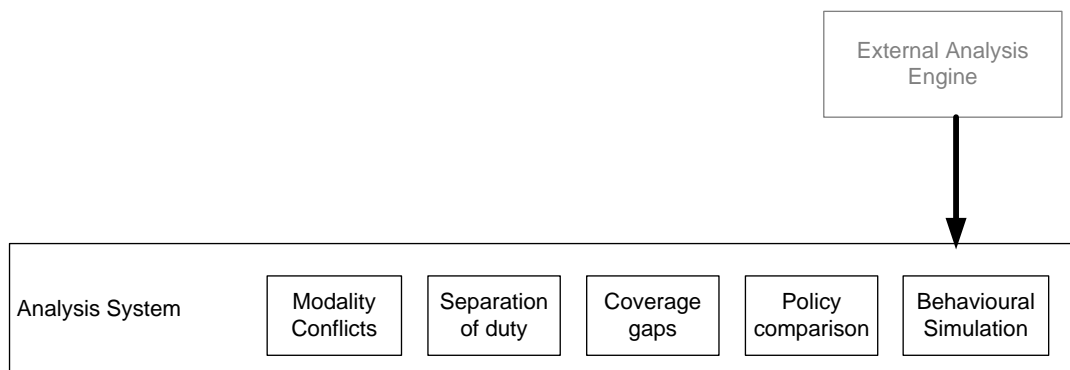


Figure 8: The Policy Analysis subsystem overview and its internals.

3.4 Tools and Frameworks Used

3.4.1 Swing

Swing provides a large range of standard widgets that we used to create the GUI. These widgets include everything from simple push buttons to tree view widgets for conditions. Layout managers were used to arrange the widgets and also perform automatic positioning and resizing depending on the contents or window size. These widgets support the MVC model, a fundamental pattern adopted throughout the project which is further discussed in Section 4.1 One of the most important features of swing is the way it handles input from the GUI. When a GUI component is clicked, it automatically fires an event. This event is received by any Listener objects bound to the component. The Listener objects are objects which have a reference to the controller and can call the appropriate method to allow the controller to handle the situation.

3.4.2 Prolog

Prolog [21] is a declarative language with roots in formal logic. Both the analysis tool's source code, and the policy code we generate with our system are written in prolog. Prolog allows us to query the system with a set of facts, rules and have an answer returned. This is done through a process of matching [14] and unification [9]. For example given the rule $a :- b$. For example, given $a :- b, c$ (a if (b and c)). If the prolog engine finds that b and c hold, then it will return a.

3.4.3 Interfacing from JAVA

Since our tool is written in JAVA, we need to find a tool to interface with Prolog. Jasper is a bi-directional interface between programs written in Java and Sicstus Prolog [26] it will allow us to send policies into the analysis engine along with any queries as part of the analysis tool.

4 Architecture

4.1 Model View Controller architecture.

Every subsystem uses the model-view-controller pattern to some degree. There are many variations on a theme of model-view-controller [29], [40], [33]. Our choice of MVC is based on Swing's heavy rooting in this pattern. Though the swing documentation goes into great detail on the actual MVC architecture [18], I provide a brief outline of our use of it here. Specifically note that Swing's method is known as 'Modified-MVC' which implements a separable model architecture and pluggable look-and-feel architecture.

4.1.1 Model

Every model in the system contains some data or information relevant to the policy specification system. This could be a list of subjects, or a list of conditions. The model cannot be accessed directly by external controllers since the model is encapsulated by its own controller. Most of our models will contain information which is displayable to some degree, to facilitate this display our custom models must implement a model interface from swing such as ListModel. While we can also do this by extending DefaultListModel, this is not desirable since Java only supports single inheritance and our models may wish to extend other predefined classes such as Observable. Our models notify any data listeners (typically views) when the contents changes, this causes the view to update its display. This implementation of this notification system is already available, provided we wire the view to the model, though we still need to trigger the update using some method (typically setFired()).

4.1.2 View

The view acts a display for the model by receiving notifications on data changes. the view typically manifests itself as a single class file with many swing components attached to it. Views receive input from the user and fire events (specifically Listener objects) which are associated with the view. The listeners contain a reference to the controller and call back the controller on specific methods (typically handleEventX()).

NB. Views are designed using the Netbeans form designer for speed. As a result, a single view class reference may be shared amongst several controllers. This is down to the fact that the Netbeans form designer encourages the user to design the entire view as a series of composite components present in one class by locking (read only) the generated form code. This prevents the user from

constructing a view from different components and adding them at runtime. The community response to this is typically “Don’t use a form designer”, which given the time limit is not an option. Though using this form designer may contradict a single function, responsibility principle, one must balance this with the need to maintain the GUI in a form designer since previous project experience has demonstrated that the trade off is worth it [17]; GUI’s built during the design phase are susceptible to a great number of changes which it is simply impractical to hand code but which a GUI design tool handles exceptionally well.

4.1.3 Controller

The controller acts as an interface between the model it protects and the rest of the system, it handles the logic for manipulating the model, and handles user events sent by the view via listener objects.

4.2 EventBus architecture

An initial examination of the design suggests there will be many controllers which may need to communicate with each other, though it’s difficult to ascertain how many controllers, which controllers will communicate with each other, and how this will change over the lifetime of the project. For two controllers communicating with each other, wiring the two controllers together is a quick and cheap solution. However for 20 controllers we would (potentially) need 20 references in every controller to every other controller, this would mean that if we tried to extend our system by adding another controller, we would need to modify every other of the 20 controllers to be able to communicate with our controller. This violates the open-closed principle [27] not to mention the difficulty in maintaining 400 references across 20 classes. Temptation is to make all classes singleton to gain global access. However this is not what the Singleton pattern is intended for. Additionally, use of the singleton pattern will hamper unit testing. We could use a registry whereby we register each controller with a singleton object together with setters and getters. This would solve the reference hell in each controller, since we could maintain our references in one place, however we would still be in violation of the open-closed principle with the registry which is not actually an interested party from the view of functionality. There is another alternative which allows us to extend the system (add a controller) without modifying the existing registry (or even requiring a registry for that matter). The EventBus system [28] as shown in Figure 9 allows for communication between objects and even broadcast of information via the publication of objects rather than directly calling methods on objects.

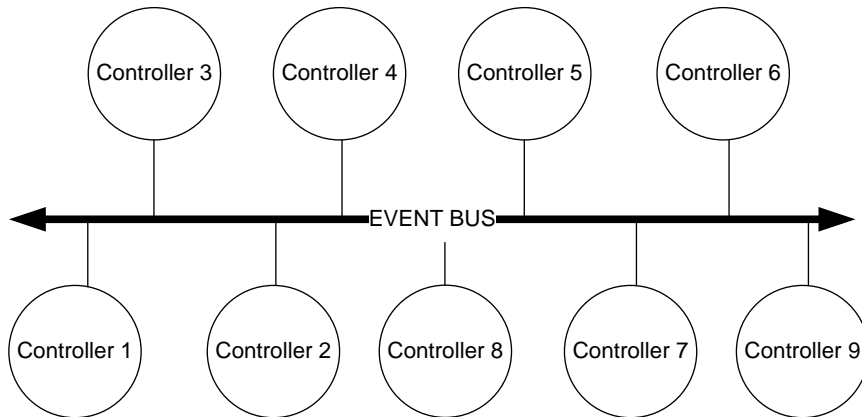


Figure 9: The Policy Analysis subsystem overview and its internals.

By simply using annotation like so:

```
@EventSubscriber
public void handler(HandlerMessage msg){
    //code handling
    ...
}
```

and including a processing directive in the constructor like so:

```
public Constructor(){
    ...
    Annotations.process(this);
    ...
}
```

we can enable one controller to call another without them ever referencing each other directly. This system of communication does not even require us to use setters to wire up the target object’s method to the message object, furthermore the message object can carry a payload if required. Since message passing is the lowest form of coupling available, this system is highly desirable since we can now add a controller by creating the controller class and enable communication to interested parties by publishing a payload object, only interested parties need to be altered. This system now obeys the open-closed principles. Whilst being very powerful, it is not a panacea to be used everywhere as I explain in the evaluation.

4.3 Observer

The Observer pattern [34] has been used in several places to allow models which don’t have a direct visual representation to notify observers (typically controllers) of their state change. This is particularly useful for automatic ‘enabled state’ checking on widgets - enabling or disabling buttons according to the state of a model. The observer is an alternative publish-subscribe system to the eventbus system it is more controlled since we must tie an observer to an observable at runtime, and the observation is restricted to these two objects. Contrast this with the eventbus where anybody can listen in, and the binding

exists as soon as a listener is initialised and is persistent until the object is garbage collected. We note there are implications of having an object in memory which is no longer required or referenced, but has not yet been collected by the GC. One implication is the ability of the object responding to signals meant for other objects. In particular the observer is more suited to the objects used to represent a policy model, since if we were to build two policies one after the other, we may observe the elements of the first policy interfering as we try to build the second policy, since they would still be in memory and permanently subscribed to objects in use in the second policy.

4.4 Strategy

The strategy pattern [19] has been a key pattern for extensibility. This pattern enables us to define various condition generation classes and dynamically interchange their instances at runtime whilst still observing the open-closed principle. It has essentially been a replacement for if-statements by using runtime polymorphism.

5 Components

5.1 Scenario File

The scenario file, which the importer must parse, is written in XML for portability, readability, but mostly because the problem we are trying to solve is similar to that solved by the spring framework's dependency injection system [7] which uses XML to represent objects to be injected [39] into a system. The next few sections cover an explanation of the file structure.

5.1.1 Headers

We begin with the standard XML header required for all XML documents.

```
1 <?xml version="1.0" encoding="utf-8"?>
```

In addition to the standard XML header, we require a way of ensuring we process correct versions of this format with correct library versions as the software is developed or at least to ensure we reject versions we cannot handle.

```
1 <Document type="Specification File" version="1.0" comments="">
  </Document>
```

5.1.2 Body

The body of the document is made of components which define the individual elements within the policy, there are exactly 2 components we are concerned with:

1. Objects: Elements to be placed into subject and target lists.
2. Actions: Elements to be placed into action lists.
3. Static: Represents static predicates.
4. Attribute: Represents an attribute of an object.
5. AttributeValue: Represents a possible value of an attribute.
6. Event: Represents an event.

We cover only Objects and Actions here, since the other elements are implemented in much the same way. A component requires a name. The name should be descriptive and must be unique since as in Spring we will be using it to describe dependencies between components.

5.1.3 Body-Class

```
2 <Object name="meeting">
  </Object>
```

Within the object we find 2 required elements and 1 optional element.

Required elements:

- English: Used for display purposes within the GUI
 - Display: The value given here should be exactly as it is to be displayed in the Subjects/Targets list
- Logic: Used as output, specifically this is the logical representation of an element to be generated.
 - Parameter: Each parameter is specified in the order it is expected.
 - * Name: The name attribute is used to identify the the argument for readability with the exception of 2 names: ‘id’ and ‘parent’
 - id: The value ‘id’ allows the system to infer that this is the declaring argument for this fluent.
 - parent: This is an indication of relation between this object and another object (A meeting may relate to a project). The system infers that a relation lies here and may use this information for display purposes when binding fluents in stage 3.
 - * Variable: The variable attribute is used to suggest a friendly variable starting point onto which the system should generate values from (e.g. suggest M , the system will generate M_1, M_2 etc... this is to avoid producing policies with non-descriptive output logic.
 - * Type: Type is used to indicate the class of the parameter expected. The type is referenced by the unique class name.

Optional Elements:

- Actions: Actions tag define a set of referenced actions which this class can have performed on it.
 - Reference: reference tag denotes a reference to another (action) class which must exist in the document. The ref attribute should be the same as the name tag of the referenced class.

5.1.4 Body-Actions

Actions are the other kind of component in the system. Similar to classes they require a unique name and have several properties, however all these properties are required. Since actions can be parameterised (for example a transfer of something from somebody to somebody else by a third party), we permit the same parameter based logic tags as given in classes.

- English: as above in classes.
- Logic: as above in classes.

It should be clear that while display names do not necessarily have to be unique, class/action identifier names do. The following snippet shows a sample XML scenario file.

```

2 <?xml version="1.0" encoding="utf-8"?>
  <Document type="Specification File" version="1.0" comments="">
    <Class name="projectDocumentation">
4      <English display="Project Related Documentation"/>
      <Logic fluentName="projectDoc">
6        <Parameter name="id" variable="D">
          </Parameter>
8        <Parameter name="parent" type="project">
          </Parameter>
10       </Logic>
      <Actions>
12        <Reference ref="read"/>
        <Reference ref="transfer"/>
14       </Actions>
    </Class>

16    <Action name="read">
18      <English display="read" />
      <Logic fluentName="read">
20        </Logic>
    </Action>

22    <Action name="transfer">
24      <English display="transfer" />
      <Logic fluentName="transfer">
26        <Parameter name="id" variable="R" />
        <Parameter name="source" type="projectPartners" />
28        <Parameter name="destination" type="projectPartners" />
        </Logic>
30    </Action>
  </Document>

```

5.2 The Importer

The importer is responsible for parsing and generating a model suitable for use by our template system. The method of import and the file to be imported is interchangeable at runtime via the strategy pattern by implementing an importer interface and an `execute()` method. Though our importer is capable of handling multiple strategies, we demonstrate only one strategy which reads the XML scenario file.

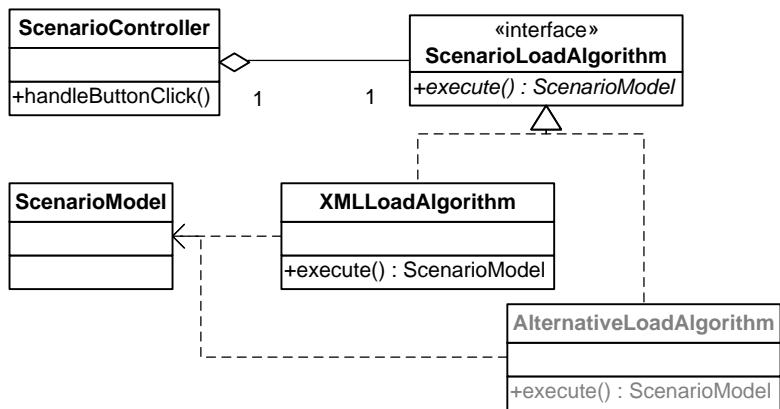


Figure 10: Class diagram demonstrating various import strategies

Figure 10 shows the architecture of our import system. We see the ScenarioController contains a ScenarioLoadAlgorithm interface (supertype strategy) which is implemented by the XMLLoadAlgorithm and produces a scenarioModel on output. AlternativeLoadAlgorithm class demonstrates how we can extend this system to handle a different format than XML at a later date by simply plugging in the class which implements the execute method in the interface.

5.2.1 The XMLLoadAlgorithm

The XMLLoadAlgorithm takes a fileURI and parses it into a ScenarioModel. We instantiate a custom builder and assign this builder to the the object implementing the IXMLParser interface from the nanoXML library [31] before calling the parse() method on the parser. When we return from this call, the builder has built a model based on the tags it encountered. A Handlers package contains the classes responsible for parsing, there is a one to one mapping between XML elements and Classes in this package, that is for each tag element encountered in the XML file, we have a separate Class designed to instantiate the appropriate ScenarioModel element. This allows us to change the structure of our XML file and easily adjust the handling system by adding/removing/editing the appropriate classes in isolation. We can do this again by using the strategy pattern where all handlers implement H_Abstract class, and delegating Element and Attribute calls to appropriate methods in each class as shown in figure 11. Furthermore by implementing the IXMLBuilder interface we can have different Builders depending on the version of the file format that we wish to parse. This will be useful for backward compatibility over time as the file format evolves.

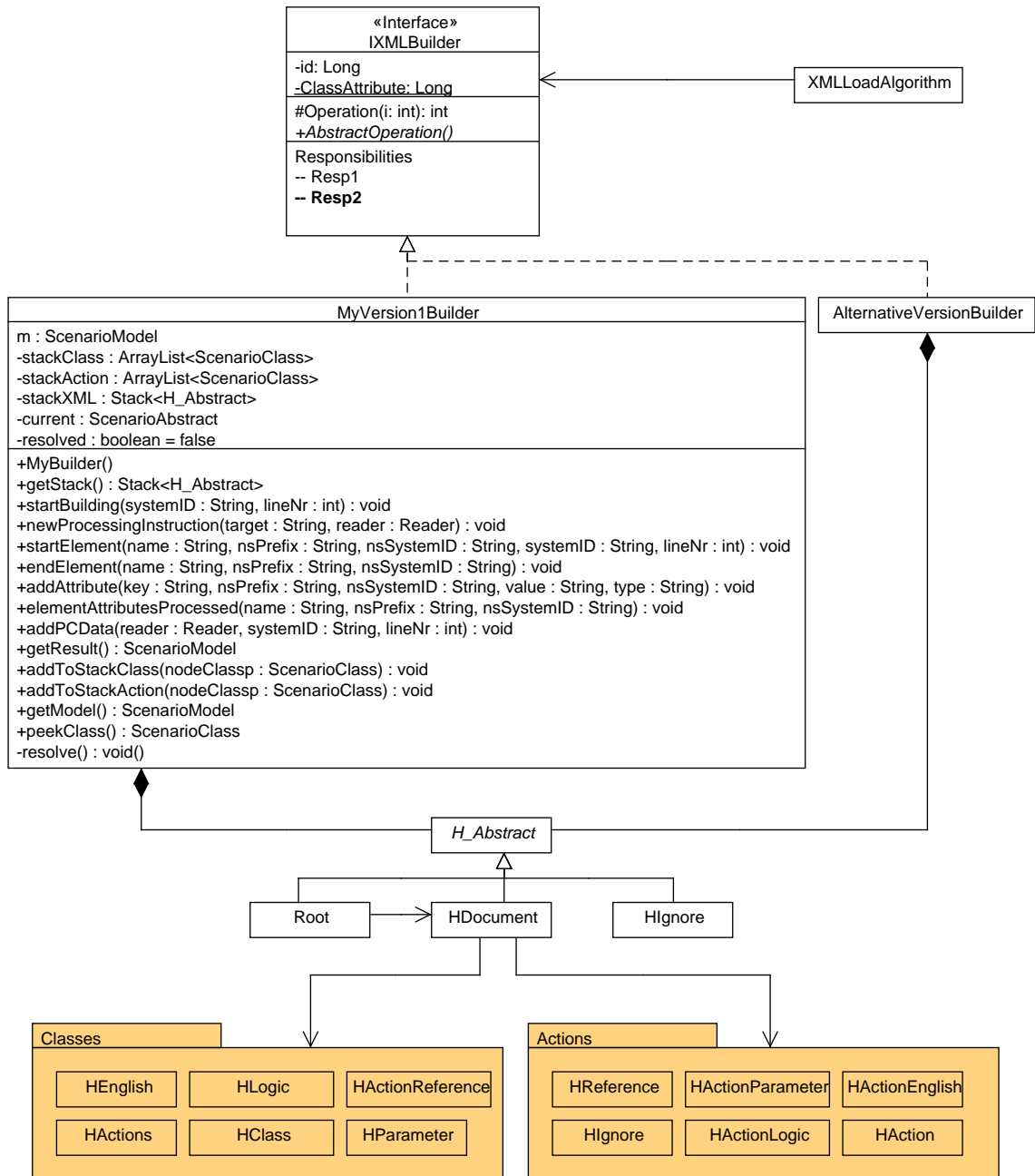


Figure 11: Class diagram demonstrating the XML handling system

5.2.2 The Scenario Model

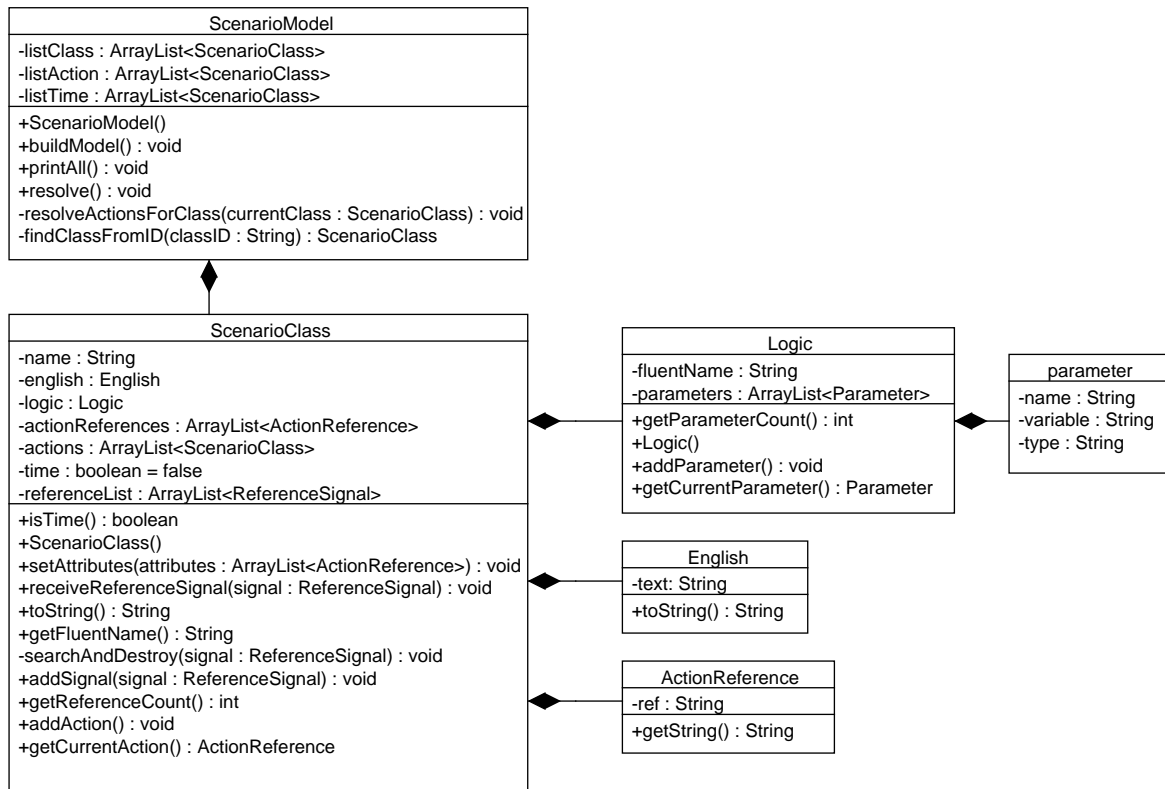


Figure 12: Class diagram demonstrating scenario model and a single ScenarioClass

The scenario model which is created by the custom Builder object consists of a group of `ArrayList<ScenarioElement>` segregated by whether the particular scenario element is an Action, Subject/Target, or Time. We do not distinguish between these concepts at a lower level since there is no additional functionality provided by actions subjects or targets; There is no reason to subclass for the sake of subclassing. The scenario model's elements are built up at each policy refinement stage as we gather more information from the user. Initially however the scenario model serves as a reference for future models which derive their information (such as subject lists) from this single reference. In particular we note that each ScenarioClass has the ability to clone itself. This lets us support the modeling of multiple policies in a single session, which is favorable over alternatives such as recording the final policy logic output as a string, since by keeping it as a model, we allow ourselves at a later date to implement editing of existing policies.

5.3 Variable Factory

During the course of building a policy we will need to generate unique variables. These may be variable for binding to fluents which identify the fluents,

anonymous variables where we don't wish to bind to any particular fluent, or time variables to be attached to special Holds() fluents and policy types such as Permitted, Obligated, Denied. To do this we will use a singleton class VariableFactory. Singleton pattern will provide us with a global point of access to the object, and ensure that only one instance of a class is created. This is useful since variables need to be unique, and by having a single global instance we will be able to retrieve variables from a system we know was the last one we used.

5.4 Stage1: Class Selection

The Class selection stage allows us to define the overall shape of the policy by selecting elements of the policy in their class form from various list widgets. The minimum input required to form a policy is selecting Subject, Action and Target classes along with a policy type (i.e. permitted/denied/obliged). We can optionally add conditions at this stage too.

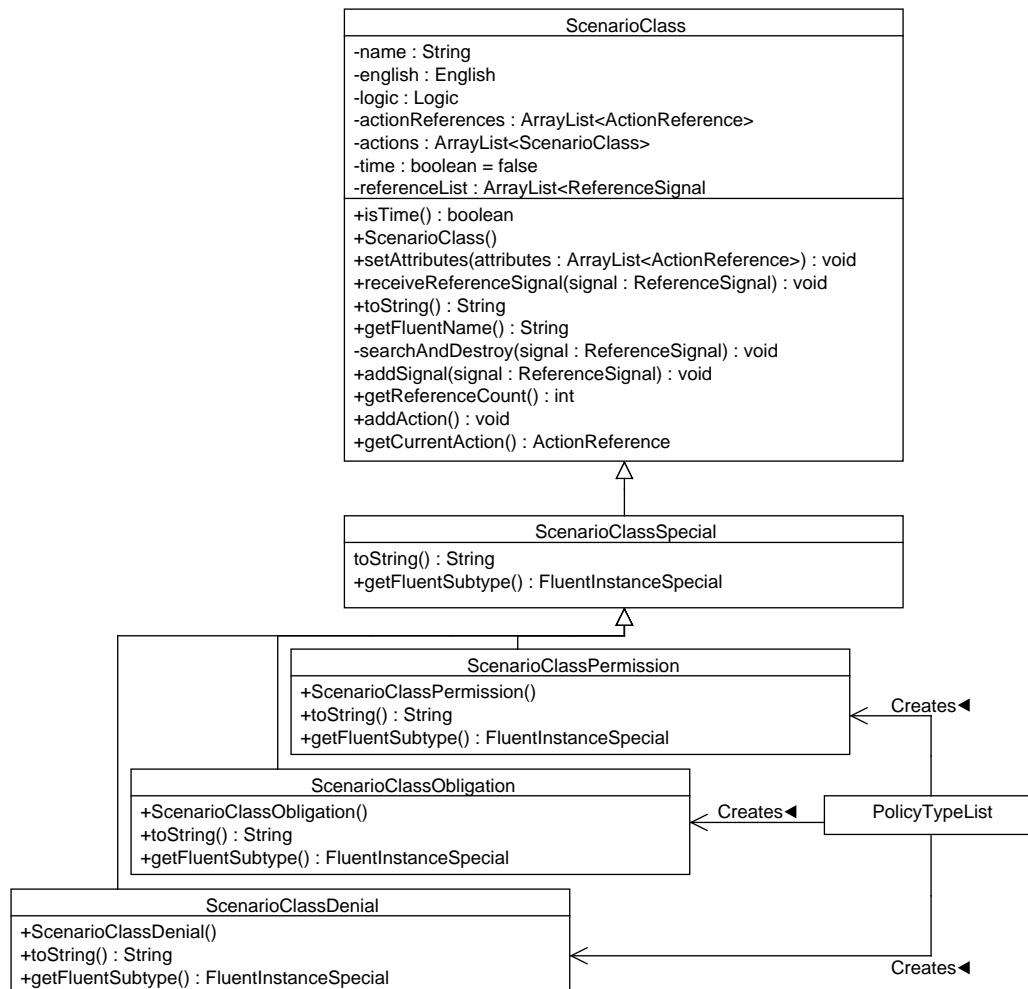


Figure 13: Class diagram demonstrating the scenario class hierarchy

Each Subject, Target, Action is represented by a ScenarioClass object and stored in a container object. Since ultimately all policy elements result in fluent creation, it is desirable to treat ScenarioClass objects in a similar way. Since we already have Subjects, Targets and Actions as ScenarioClass, we need only find a way to represent Policy types (*Permitted()*, *Denied()*, *Obligated()*) as ScenarioClass types. These elements are not derived from the scenario file and all contain special time values which we need to bind later. In particular we note that the Obligated fluent is a special case which carries 3 time fluents. To solve this problem of representation we subclass ScenarioClass into an abstract type ScenarioClassSpecial, from this we can derive any special cases to represent policy types. We provide concrete implementations ScenarioClassPermission, ScenarioClassObligation, ScenarioClassDenied. The instantiator of these types is PolicyList since the PolicyList type will contain these objects. By subclassing these special cases we can define the custom structure (parameters, display etc..) in the constructor, yet to external users, they will appear as simply ScenarioClasses. We note that if at some point in the future a new policy type is invented we simply plug in a new policy class. See figure 13.

5.4.1 The specification component.

The specification controller is responsible for maintaining the models and views which focus on the English consequent of the policy at the Class Selection stage. The model maintains a list of the available subjects actions and targets along with the current selections. Since the action list is always filtered by target, we must notify the controller of any changes made to the target list. We use listeners to notify the controller which rebuilds the action list according to the actions associated with the current target. As the user selects policy elements from the various lists, the controller is notified of the selection index and sets the respective ‘current’ element in the model. (e.g. If the user selects ‘project manager’ as the subject, the view fires a listener which calls the controller on a handleSubjectChange method. The method body sets the currentSubject field to reference the ScenarioClass returned on an get method of an arraylist of subjects. See the left-hand panel of 14

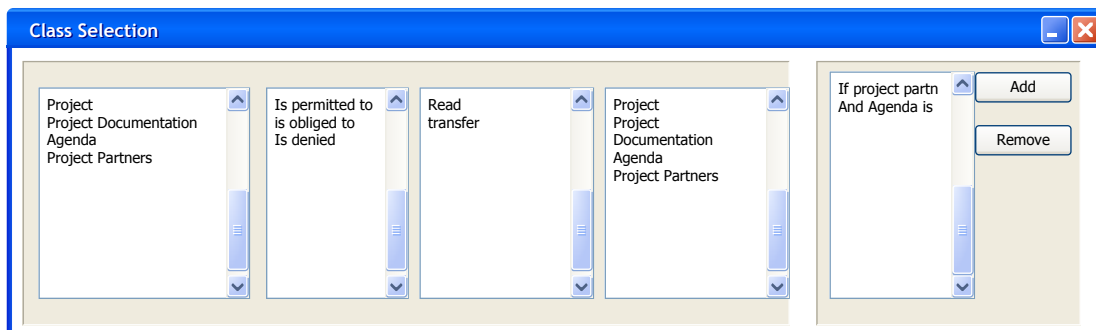


Figure 14: Design screenshot of the View handled by specification and condition Controllers

5.4.2 The condition component

The condition controller is responsible for maintaining the list model of all conditions in the Class Selection stage. The input of data for individual conditions is implemented via a tab system as shown in Figure 15. Each tab represents a condition and has a respective concrete controller strategy to manage it. So we see the condition controller itself does not handle every type of condition, instead each strategy for handling a particular type of condition is encapsulated in a concrete subclass of the ConditionBlock class. When we click add, the condition controller causes a tabbed window to appear. As the user changes the tabs, the controller used to manage that particular condition strategy is changed. The user fills out the fields and the particular condition controller handles the construction of the model. When the user clicks OK the window is closed and the new model of a condition is added to the ConditionController's model to be displayed by the view. This use of the strategy pattern appears at every stage of this project since every stage has a condition module.

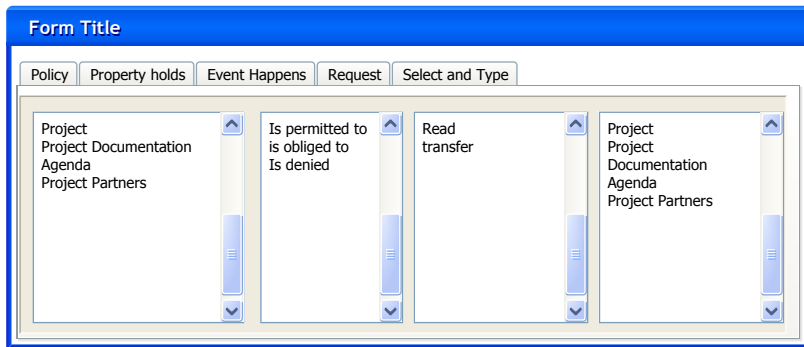


Figure 15: Design screenshot of the view handled by various condition strategies

5.5 Stage2: Instantiation

Under a manual process, instantiation of a ScenarioClass involves writing down a fluent and assigning it a Variable with which to identify it. Suppose we want to instantiate a 'Project Documentation' fluent, we would write something like $projectDocumentation(D, -)$. This is the minimum requirement for fluent instantiation. If we later say $holds(projectDocumentation(D, -), T_1)$ and $holds(projectDocumentation(D, -), T_2)$ We know that at both instances of time T_1, T_2 we are referring to the same 'project documentation' fluent, however if we also instantiate $projectDocumentation(E, -)$ and later say $holds(projectDocumentation(D, -), T_1)$ and $holds(projectDocumentation(E, -), T_1)$ and $D \neq E$ then we can say that at time T_1 , we are referring to two different pieces of project documentation. We wish to take this responsibility of manually binding an identifying variable to each fluent away from the user and now present two approaches to this: the 'Add As Required' and 'Reference Counting' methods.

5.5.1 The ‘Add As Required’ method

We initially present the user with a listbox containing just one instance. We then allow the user to click a button to add more instances as they feel necessary. A problem arises if the user adds lots of instances, makes a selection, and then tries to reduce the number of instances. The behavior which should arise from this situation is not clear and there are better alternatives.

5.5.2 The ‘Reference Counting’ method

By examining several policies and using intuition about the structure of policies in our template system, we realise that the (maximum) number of instances we need depends on the number of times we have references the element in stage 1. We therefore use reference counting as a method for evaluating the maximum number of elements in any template slot for a particular ScenarioClass. See 16

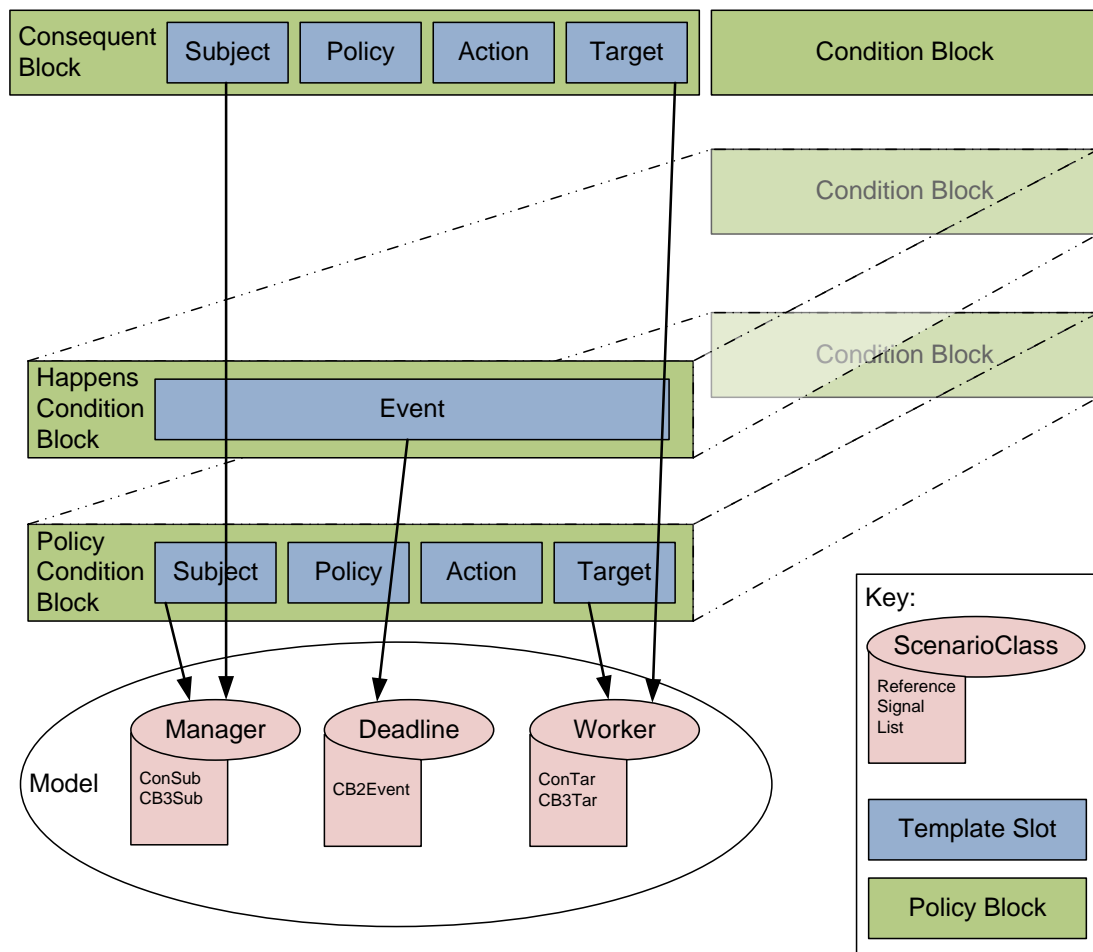


Figure 16: Shows the implementation of reference counting system with respect to template slots

To implement reference counting every ScenarioClass object contains a list of ReferenceSignal objects, the number of ReferenceSignal objects in the list

determines the number of references made to that ScenarioClass object in the entire policy. Note not all elements need to be tracked, for example PolicyType elements such as Permitted, Obligated, Denied do not need tracking. ReferenceSignal in detail: ReferenceSignals contain a string which identifies the template slot uniquely as follows: For consequent elements Subject, Target, Action, we publish ReferenceSignals with an id of 'csubject' 'ctarget' 'caction'. For conditions we publish the element as above and append a hash of the ConditionBlock object. This will allow us to uniquely identify any ReferenceSignal. Now that we know how to unique identify ReferenceSignals with respect to template slots, the process of adding is as follows: As per MVC system, the view informs the controller that a ScenarioElement has been selected or changed. We create a ReferenceSignal object with the appropriate ID, and broadcast this over the EventBus. Every ScenarioClass receives this, and checks in it's referencesignal list for a matching ID. If one is found, this element is removed (We are changing the template slot element) If none is found in the system, this must be a fresh selection. Worth noting that the broadcaster does not know if a match was found or not, since this search operation is effectively delegated elsewhere. Finally we add the reference signal to our current ScenarioElement. Here follows a code snippet.

```

class SpecificationController {
    ...
    public void handleSubjectSelectionChange(int lastIndex) {
        ReferenceSignal r = publishReferenceSignal("subject");
        ScenarioClass current = specificationModel.getSubjectsModel().getElementAt(lastIndex);
        current.addSignal(r);
    }
    private ReferenceSignal publishReferenceSignal(String signalString) {
        ReferenceSignal r = new ReferenceSignal(signalString);
        EventBus.publish(r);
        return r;
    }
    ...
}

class ScenarioClass {
    ...
    @EventSubscriber
    public void receiveReferenceSignal(ReferenceSignal signal) {
        //search for the signal in the list, if it's there, delete it.
        searchAndDestroy(signal);
    }

    private void searchAndDestroy(ReferenceSignal signal) {
        String newOwner = signal.getOwner();
        int i = 0
        while (i < referenceList.size(); i++){
            String oldReference = referenceList.get(i).getOwner();
            if (oldReference.equalsIgnoreCase(newOwner)){
                referenceList.remove(i);
                return;
            }
        }
    }
    ...
}

```

This is a clever method since we don't need to know who is currently referencing a scenarioelement, we do not need to manually search the entire model, instead we publish a reference and delegate the searching of reference lists to each ScenarioClass object who has the knowledge to act (Expert principle). It's also extensible, since any extensions to the policy template will not be affected by this system, and the ScenarioClasses themselves do not need to be modified. (Open-Closed principle).

5.5.3 FluentInstance class

In order to represent instantiated fluents we define the FluentInstance class which takes a ScenarioClass as a constructor. Recall the ScenarioClass is a blueprint object for building fluents. The fluentInstance examines the properties of this blueprint (the fluent name, number of parameters, and ID location) and sets it's properties accordingly. In particular the fluentInstance generates a single Variable to identify the fluent (if necessary). It does this by going through all parameters in the ScenarioClass searching for one tagged as being an ID. It then inserts this variable into a two-fold data structure. Firstly it inserts it into:

```
HashMap<Integer,Variable> parameters.
```

The key used is the position of the ID variable in the ScenarioElement. However at some point in the future we may need to retrieve a variable using information from the ScenarioClass such as the Parameter object which described the variable. So we also add the Parameter object from the ScenarioClass together with the index into a

```
HashMap<Parameter,Integer> parameters.
```

hashmap. This will make future lookups faster. We apply the architecture used in Stage 1 for ScenarioClasses to FluentInstances, see Figure 17

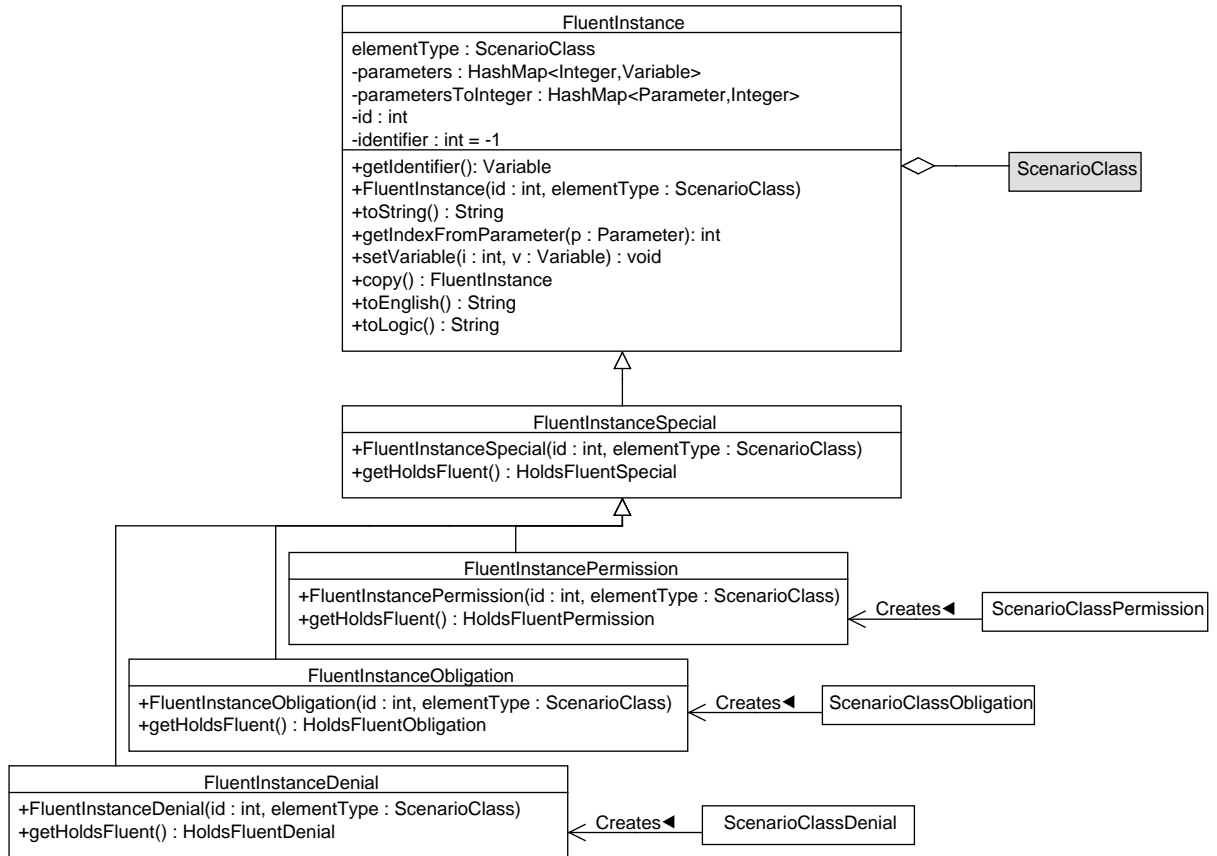


Figure 17: Class diagram demonstrating the FluentInstance class hierarchy

Again by doing this we allow the programmer to treat all fluents in the same way, whilst allowing the model to maintain knowledge of any special subtypes such as permitted denied obliged etc...

5.5.4 Instantiating Conditions

In order to instantiate conditions, we need to be able to present the user with different views depending on the type of condition whose fluents are being instantiated. As in the previous stage, we use the strategy pattern to define our conditions. When the user selects a condition from the list, we populate the relevant model with that condition's data, and display the appropriate tab for fluent selection. The method of selecting the right model makes use of the EventBus. Essentially the InstantiationPackage.ConditionController calls a method on the ConditionBlock which has been selected. This method instantiates a ChangeTabSignal object of a concrete type depending on the concrete type of the conditionblock. (Expert principle). This signal object contains a reference to the concrete type of the ConditionBlock and is broadcast on the eventbus. One of the subtypes of AbstractConditionController will have a method accept() with the concrete Signal. This method then unpacks the

object, and fills in the model as appropriate. Again this method has extremely low coupling since the ConditionController does not need to know about every possible concrete ConditionTab controller, nor does it need to know the exact subtype of ConditionBlock and is able to treat these objects in a uniform way.

5.6 Stage3: Binding Fluents

Stage 3 allows us to associate fluents with one another. The first stage is to enable the user to select the element whose parameters need to be bound. We present the policy in a structured manner showing the consequent section, along with a list of conditions and an area to show a current condition as shown in figure 18.

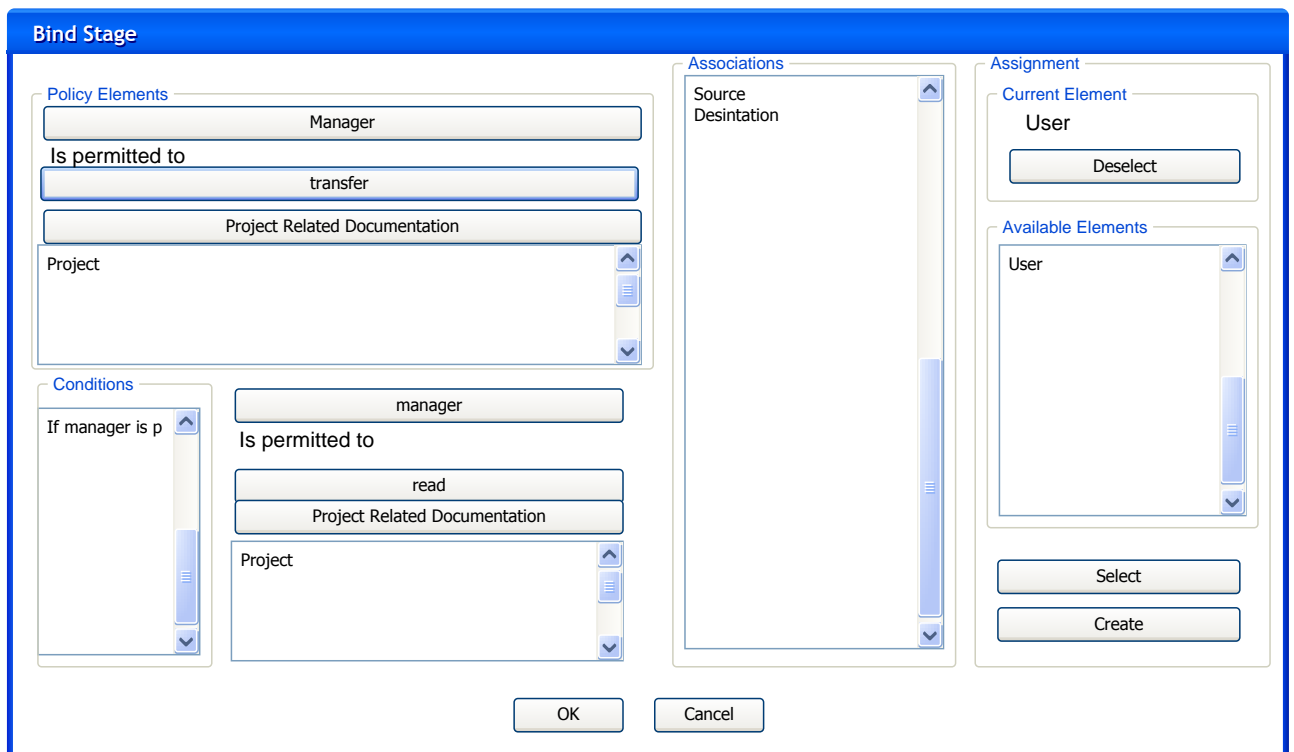


Figure 18: The view of the bindstage showing ‘transfer’ as the selected

The bindstage ConsequentController, ConditionController both receives a model which was published at the end of stage 2 on the EventBus. They both import this model and look at the relevant components which need to be imported. The controller then calls the view, setting each view component individually with an English representation of the fluent. When the user clicks a particular widget, the controller responsible for that widget tells the BindController to set the current policy element to be whatever is represented by the widget. The bindcontroller then looks up all the parameters of the FluentInstance, and lists these by name (except for the ID), whilst noting the type in the background. We use a TypeEnglishWrapper to display an English type:

```

private void rebuildTypeList(FluentInstance currentFluent) {
    typeList.clear();
    view.clearTypeSelection();
    ScenarioClass currentClass = currentFluent.getElementType();
    ArrayList<Parameter> paramSet = currentClass.getLogic().getParameters();
    for (int i = 0; i < paramSet.size(); i++){
        String currentName = paramSet.get(i).getName();
        if (!currentName.equalsIgnoreCase("id")){
            //add to list
            typeList.addElement(paramSet.get(i));
            TypeEnglishWrapper d = typeList.getElementAt(0);
            view.repaintType();
        }
    }
}

```

When the user selects a particular parameter, the BindController then delegates a search through the policy model for all FluentInstances of the selected type. It calls back the controller by broadcasting a FoundMatchingFluentsMsg result. We then display the result in the Available Elements list:

```

void handleTypeSelectionChange(int selectedIndex) {
    currentType = typeList.getElementAt(selectedIndex);
    FluentSearcher algorithm = new FluentSearcher();
    ArrayList<FluentInstance> searchResults = algorithm.search(currentType.getCorrespondingClass());
    for (int i = 0; i < searchResults.size(); i++){
        this.assignmentList.addElement(searchResults.get(i));
    }
}

@EventSubscriber
public void acceptFoundMatchingClassMsg(FoundMatchingFluentsMsg msg){
    for (int i = 0; i < msg.getMatchingFluents().size(); i++){
        this.assignmentList.addElement(msg.getMatchingFluents().get(i));
    }
}

```

If the result set is empty, it indicates that there has not been a mention of this type of object in the policy yet. The user can choose to leave the assignment blank (in which case the anonymous variable is left in tact underneath), or the user can click create, which causes a new fluentInstance of the selected type to be instantiated and added to the list. If the user then selects this element, the current assignment selection changes to this fluent. If the user then hits the select button, the AssignmentController inspects the element, derives the ID Variable, and assigns it to the current fluent. Essentially we have bound two fluents together by referencing the same variable in both fluents. See figure 19

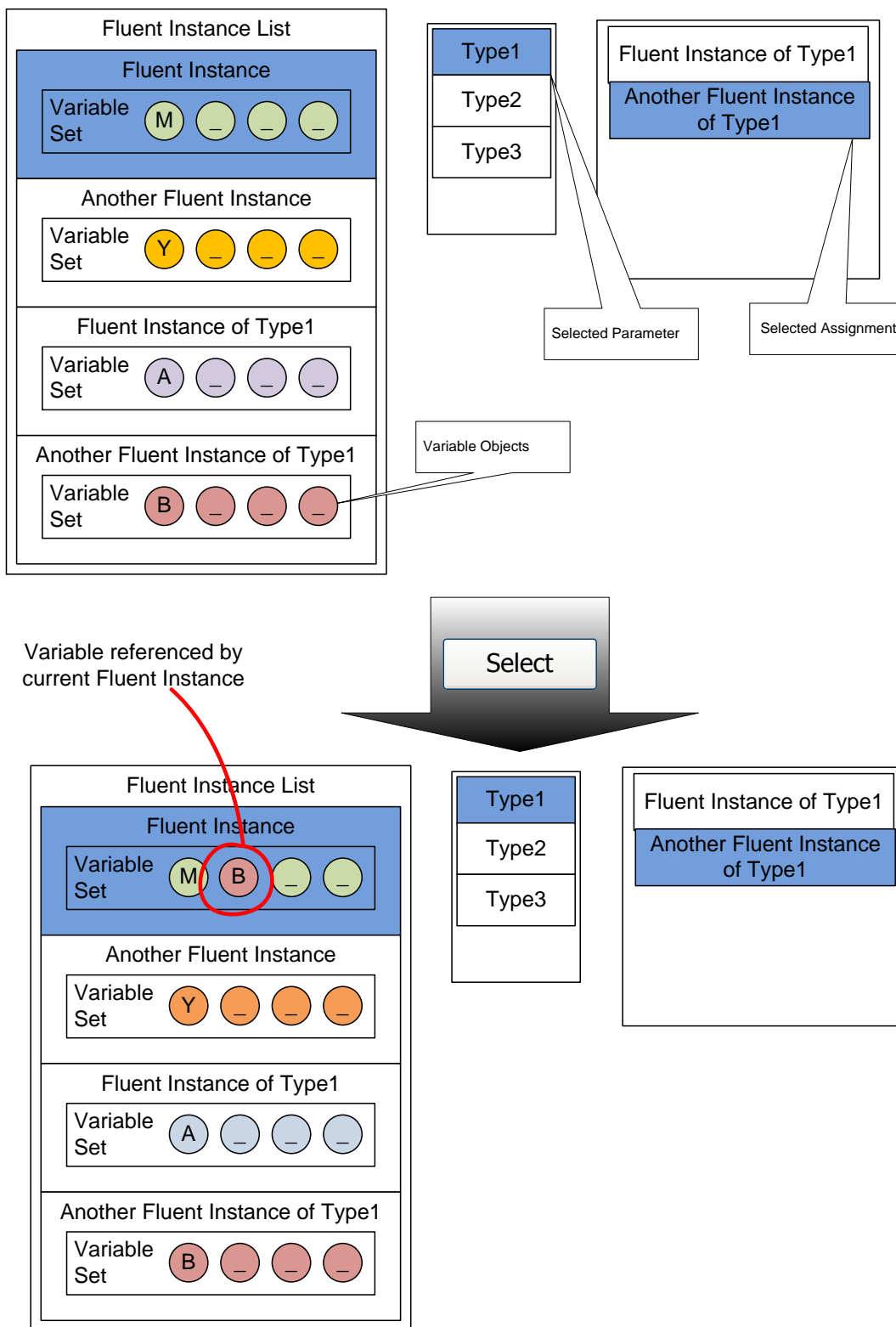


Figure 19: Diagram showing how the bind system state changes when select button is clicked

When we create a new element, we are essentially adding a peripheral element to the policy. There is no place for peripheral elements in our policy template since they are only context elements. Instead we maintain a list of ‘extras’ which is associated with each Policy Block (i.e. an extras list in the consequent, and one for each condition). Whenever we create a peripheral element, we add it to the extras list. In order for the BindController to treat every controller that contains an extras list in the same way, we implement the Extras interface:

```
public interface ExtraContainer {
    public void addExtra(FluentInstance f);
    public ArrayList<FluentInstance> getExtras();
}
```

This allows the Bindcontroller to add extra fluents and other controllers to get such fluents if necessary. Note that since the extras list is just a list of FluentInstance objects, we can in turn bind any parameters of these peripheral fluents to yet other peripheral fluents.

The condition controller is implemented as in previous stages whereby a selection on a condition causes the appropriate Controller to handle it via a combination of the strategy pattern and EventBus system.

When we are done binding fluents, the user hits the OK button, and the BindController broadcasts a model of it’s final state to any controllers which are interested.

5.7 Stage4: Binding Time

Unlike in the other stages where we have been able to completely insulate the user from the complexities of the underlying language, our binding time system falls somewhat short of this. Ideally we would like to create a system where the users describes the temporal aspects of the policy in English like this example:

‘User is permitted to read documentation if at some point in the past the documents were transferred’

Here we see that the temporal aspects prefix the condition and are embedded in the grammar of the policy. However due to the extensive expressibility of the policy language, a user interface to cope with such a system would be extremely complex, certainly too complex for the timeframe at hand. We did attempt to tackle this problem directly and my initial design (which was not carried forward) for an English approach to the problem is covered in section 8.6.1, I present the design which was carried forward to implementation here:

The user will have full control over the binding of time variable in the policy, and will still not have to directly read fluents. However the process of binding is a direct one, that is the user creates a time variable (e.g. T) and binds this time to a policy fluent by clicking a button associated with a policy element. The button associated with the policy element will changes it’s caption to match the variable selected. In this way the user still does not have to write fluents manually, but we can see they still have to be aware of the relationships between Time elements. Once again we segregate responsibility for consequent specification (ConsequentController) from condition specification (ConditionController) and have a separate TimeController to manage the creation of time variables and expressions.

5.7.1 TimeController

The user creates time variables on demand by clicking the create button. This via the usual MVC channels calls a handler method which uses a variable factory to create a new variable and add it to the TimeModel. There are several models which observe use the controller as a middleman for update propagation. For example if the underlying (non-visual) model changes, the controller is notified, and in turn notifies all listening models that the underlying model has changed. In this way we can synchronize the view of the TimeVariable list, and the expression builder list. Once we have created our list of variables it is best to bind them to the policy elements first before creating expressions on them (though it is of course possible to do it the other way round, testing in section 6.3 demonstrated that users find it easier to assign them to a policy element first). Once again we have laid out the policy elements which can be bound to time in the same template format as we did in the previous Binding stage. However instead of having a list of types, we know there is only one variable (type ‘time’) which needs to be added. At the end of the last stage, a signal was broadcast containing a model which represented the final state of the Binding model. Each component (Consequent, ConditionController) imports this model and creates a set of HoldsFluent objects passing the relevant fluent as a parameter. In the case of PolicyType fluents (represented as concrete subtypes of FluentInstanceSpecial), we call a method on the FluentInstanceSpecial supertype which delegates the call to the concrete type and allows them to create their own concrete type of HoldsFluent (by Creator and Expert) see Figure 20, (This is required because whereas normal policy elements are surrounded by a holds fluent and paired with a time variable, theses PolicyType fluents are not surrounded by a holds fluent and instead have a time variable inline, furthermore, the PolicyType elements vary amongst themselves - Obligated fluent takes 3 time parameters, where as Permitted and Denied only take 1).

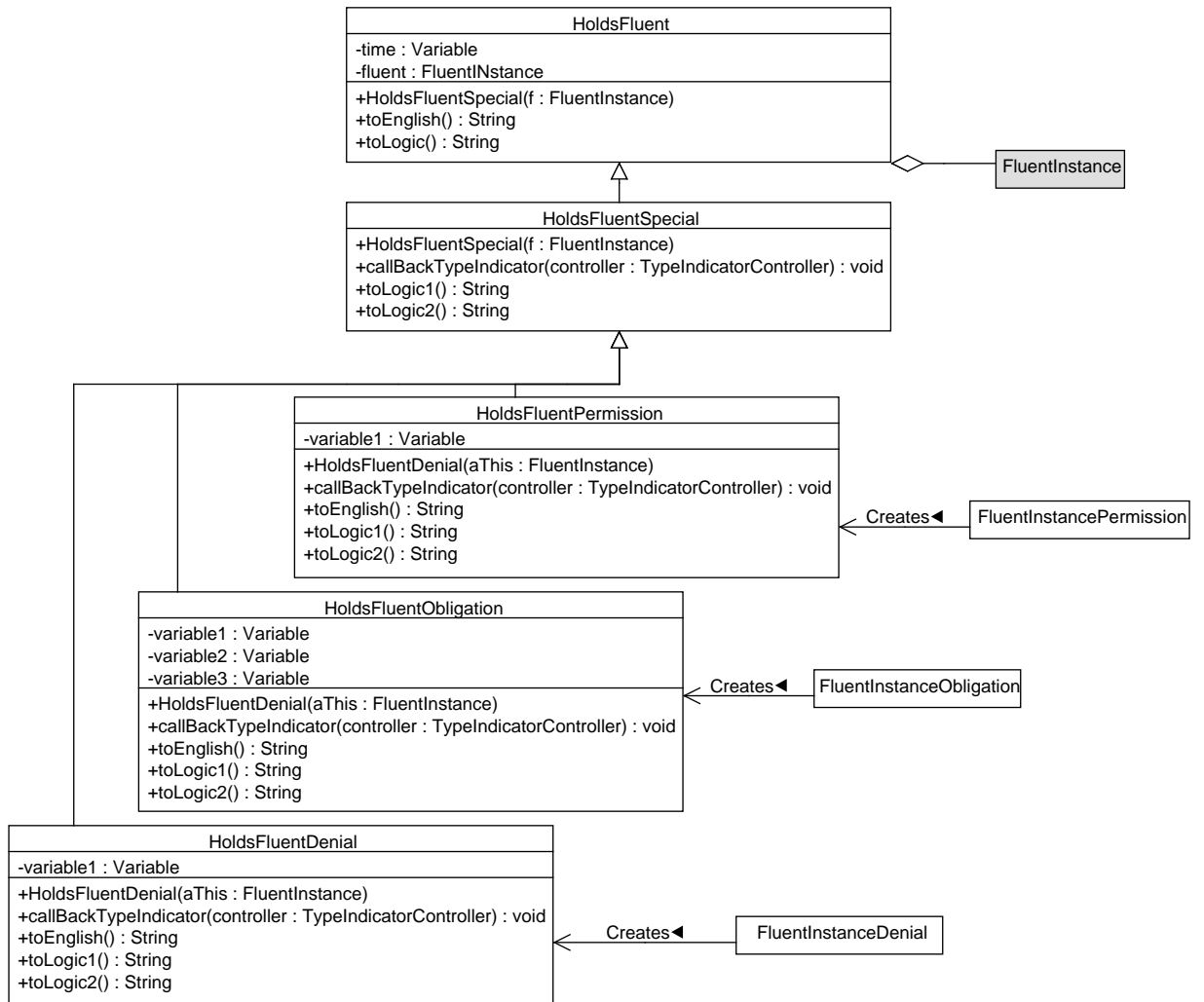


Figure 20: Class diagram demonstrating the HoldsFluent class hierarchy

This class hierarchy allows us to treat all holdsfluents in the same way regardless of whether they are standard HoldsFluents or a specific subtype. As in the previous stage, each button has a listener attached which calls the controller on a set variable method. The currently selected time variable is bound to the holds fluent and the view is updated to reflect this.

5.7.2 ExpressionBuilder

The expression builder (see Figure 21) allows the user to place constraints on the time variables in the time variables list. The result is a triple of the form $T_1 R T_2$. Where T_1, T_2 are times, R one of the following relations ($=, <, >, \leq, \geq$).

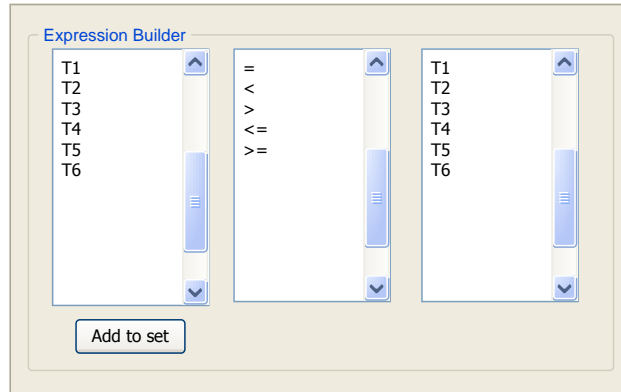


Figure 21: Expression View showing 6 variables and the different operators

When the user clicks add to set, a Triple object is created containing a reference to the two selected Variable objects, and a reference to the selected operator. The Triple object is added to the expression set list (see Figure 22) and the model notifies the view of the changes via the usual MVC/data listener channels.

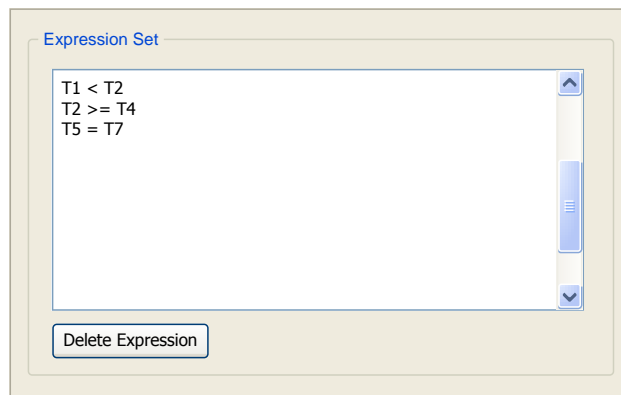


Figure 22: Expression View showing 3 constraints on time variables.

The user can easily delete expressions using the delete Expression button which removes the selected expression from the expression model, however we do not allow the user to delete variables since this complicates the Variable factory (difficult to tell the variable factory to re-issue old variables safely without implementing yet another reference counting system for the variables), furthermore it is unnecessary since only variables that are bound in the policy or by an Triple in the expression set will actually appear in the policy.

5.8 Display

The display controller is responsible for importing the final output from stage 4 and turning it into a readable form. When OK button is clicked in stage 4, a model is broadcast which the display controller picks up. The display controller then calls on the model to produce a string representation of itself which is then set on a view widget. Since every fluent object implements the *toLogic()* method, the process is fairly straight forward. There are two exceptions to the

print method *toLogic()* which we shall see shortly. The call order is as shown in figure 23

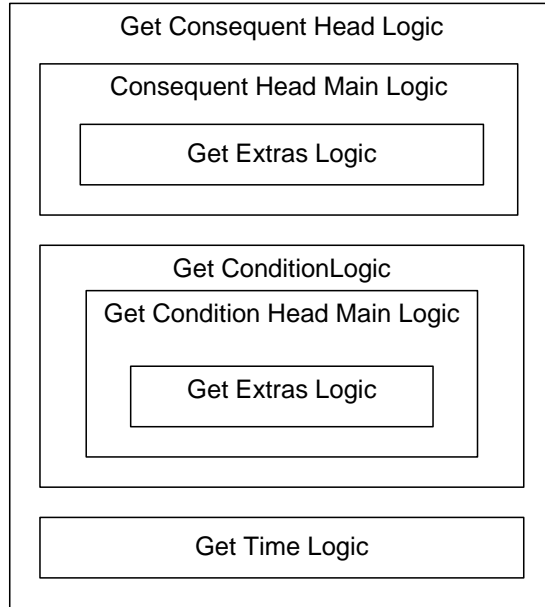


Figure 23: Diagram illustrating the containment of the policy model and thus the calling order required to return a policy as a String.

The printing of standard HoldsFluent objects is done via the *getLogic* method, however, due to the fact that actions are always printed inline, special cases of PolicyTypes such as Permitted, Obligated, Denied, will need to be broken up in order to inline the action, we do this by splitting the *getLogic()* into *getLogic1()* and *getLogic2()*. We use a series of functions to generate a string by concatenating the output of sequential calls to conditions, finally when the main bulk of the policy has been added, we append the time constraints and the string is returned to the caller.

6 Testing

6.1 Unit Testing

We tested the more algorithmic elements of the application via a series of unit tests. We used JUnit 4 [2] as a testing tool since it uses annotations to find test functions. (cf. JUnit 3 [1] which requires every method to be prefixed with test). The main focal point of unit testing was the Scenario Import system.

6.1.1 Testing Scenario Import System

I built a scenario file containing one of every element to be parsed. Where components were optional, there would be one instance where the component was provided, and one instance where the component was omitted. I built a version of the model by hand coding one based on what the correct model should be, and made assertions about the types of classes generated by the model builder and their contents. The tests passed as expected. However we did not experiment with extreme cases where the scenario model was broken or malformed. Were this project to be developed further, one would expect a full time team of testers to built a full testing suite. This would cover extreme cases that we require.

6.2 Beta Testing

For the remainder of the system, we found that unit tests were:

1. very complicated
2. crossing class boundaries
3. fragile.
4. harder to write than the code being tested

On the advice given by Jeff Canna [13], we looked at some functional testing systems such as Jemmy. However these tools were either commercial [23] or lacking in documentation [3].

Since there is no such thing as exhaustive testing [20] (or at least there is only such as thing as exhausting testing), I decided the best way to test would be to assemble a small team of beta testers to test the software, rather than try and deal with half-baked, poorly integrated functional testing tools. I asked 3 Computing students at Imperial College to try and break the tool for an hour. Initial beta uncovered a number of bugs, a sample of which I list here:

- Attempting to load a file which was not a scenario file did not produce an error.
- Attempting to load a malformed scenario file did not produce an error.
- Conditions box could only be closed with X button - cancel button was not wired.

- Selecting a target, then action, then a target, and not reselecting an action results in OK button being incorrectly enabled.

6.2.1 Beta testing response

Most of the bugs were trivial to fix. Fixes included inserting code to catch exceptions, wiring in a listener objects, or resetting ‘current’ fields to ‘null’ when a new target was selected.

6.3 Usability Testing

As well as the BETA test which aimed to uncover functionality issues in the application, we also went through a usability testing phase where the usability of the application was tested. Users were asked to give worded feedback, and a sample of comments is given here:

- “There is no ‘first-glance’ method of telling whether a fluent has been bound or not. Some sort of colouring scheme for when elements have been bound. Especially in the case of time bindings where every element must be bound by time.”
- “There is no English preview as the policy is built, though I can see what the policy is by reading off the highlighted elements, it would be nice to have a separate plain text preview of the policy clear of any widgets.”
- “The way of indicating different elements in stage 2 is user-friendly when making the initial specification, but not so user-friendly when making alterations. Having brackets containing the instance number is not very clear, again colouring could make a difference.”
- “The wizard is very effective, however it would be nice if the overall shape of policies on screen were consistent across the different stages.”

6.4 Response to Usability Testing

The result of usability testing uncovered some features which could be brought in as part of future versions such as colouring policy elements that have been bound in green whilst leaving others in red. We use the feedback given as part of our evaluation later.

6.5 Summary

Beta testing was an effective short term way to discover bugs in the current system, however it is not particularly effective at regression testing. This means that if we alter the code to fix these bugs, we would have to release it to beta testers again for testing which could uncover new bugs introduced by the fixes. For a longer term project it is not practical to do this, and functional testing tools would be better suited. However, given the time constraints beta testing suited the purpose well.

7 Profiling

We implemented the storing and display of multiple policies in memory. We now examine the cost of policies in memory, and look at various statistics such as the maximum number of policies etc... Using a profiler we measured the size of each of the objects created when a single policy is built in the system. From this we can work out some important statistics relating to aspects such as the number of policies our system can handle.

Type	Size (Bytes)
HoldsFluent	32
FluentInstance	48
ScenarioClass	72
HoldsFluentPermission	40
ScenarioClassPermission	72
FluentInstancePermission	48
ClassEnglish	48
ClassLogic	64
Variable	72
FinalMode	48
Parameter	80

Table 2: A table listing the various objects and their memory requirements along with how many are required in a single policy.

Table 2 shows the memory requirements for each of the elements in the policy as derived from the profiler. We now present the working for several policy scenarios.

7.1 Minimum Permissions Consequent

Assumptions:

1. No inter-fluent binding between subject and target.
2. Only the FinalModel object and it's contents is used to store a policy.
3. Subject and Target are distinct.
4. Action has no parameters
5. No constraints on time variables.

Every ScenarioClass carries one ClassEnglish object and one ClassLogic object. A ClassLogic object will contain exactly 1 parameter for a Subject or Target, 0 for actions, 0 for PolicyTypes The total size of a ScenarioClass object is therefore $72 + 48 + 64 + 80 = 264$. (184 for Actions/PolicyTypes)

- Subject (264), Action (+184), Target (+264) represented by a Scenario-Class

- “Permitted to” (+144) statement represented by a ScenarioClassPermission (72) inheriting ScenarioClass (72)
- The above elements brought forward into Fluent Instantiation. (+48), (+48), (+48), (+48), (+48)
Subject and Target create a Variable ID. (+72), (+72)
- The above elements brought forward into HoldsFluent (+32), (+32), (+32), (+40)
Each element binds at least a time variable. (+72), (+72), (+72), (+72)

The total size in bytes for a single policy is:

$$(4*72)+(3*32)+40+72+72+(5*48)+(72+144+184+264+264) = 1736. \quad (18)$$

Assumption 2 is very important since it means we can only store the policy and not edit it. If we want to edit the policy we would have to extend our model to include all other submodels which were created. This would include lists of Scenario elements, lists of fluent instances that were not selected, and would make the model substantially larger since we would be carrying significant amounts of dead weight required solely for editing such as alternative choices, instantiated but unused fluents etc.

On a 4gb machine the above result suggests we can accommodate just under 2.5 million policies in memory. Obviously this is the absolute minimum and it is highly unlikely that we would expect to see a system with 2.5 million policies without a single condition in. However this gives us a rough indication as to the size of our model, and the order of magnitude of policies we can hold in memory. If we were to say that 1000 policies is a reasonable number of policies to maintain in memory. We would have to create policies with an increase of 3 orders of magnitude in order to reach this limit. Clearly no reasonably condition would possible increase the size of a single policy by 1000 times.

8 Evaluation

8.1 Comparison With Initial Requirements

	Requirement	Completion
Key	Pluggable & Extensible architecture.	✓
	Build policies from templates	✓
	Output logic policy representation	✓
	Permission Policies	✓
	Denial Policies	✓
	Obligation Policies	✓
	Permission Denial Obligation Conditions	✓
	HoldsAt conditions	✓
	Happens conditions	✓
	Do conditions	✓
	Request In Between conditions	✓
	Request conditions	✓
	Fulfilled conditions	✓
	Violated conditions	✓
	Static Predicates conditions	✓
Write policy to disk	✓	
Advanced	Interface with Prolog engine	X
	Policy Comparison	X
	Separation of duty conflict analysis	X
	Modality Conflicts	X
	Coverage Gaps	X
Extra	Store multiple policies	✓
	Write policies to disk	✓

Table 3: Comparison of initial requirements against completed project, ordered by importance to the project.

8.1.1 Completed Requirements

By the end of the project's development I fulfilled all the key requirements as laid out in Section 1.3.1. In doing so I have produced an application which meets the original specification for SOAPWADI. I am satisfied with the implementation of the key requirements, including specification, reduction from a high level formalism, and the extensible nature of the architecture. I believe that these operations are implemented in a clean manner and are easily accessible to the end user.

8.1.2 Dropped Requirements

During development I re-evaluated the advanced requirements and selected several which were dropped from development owing to time constraints. These features, were seen as having the least impact on functionality of the project given the time required to implement them. I would like to see these features implemented in a later version of the project and believe that through the design patterns used in the project they would be relatively simple extensions. Over the course of this evaluation and conclusion section, I cover implementation and design ideas for the dropped requirements.

8.1.3 Additional Features

During the initial development of SOAPWADI, I had not decided upon a standard Scenario File, and in fact attempted to build a model directly from a UML diagram built in ArgoUml. Once it became clear for the reasons laid out in section 3.2 that this was not practical, this route to model building was dropped, however the legacy code remains in the codebase and serves to demonstrate the ease with which one can build a model from an ArgoUml. In addition to this there were several features which we realised would be useful to the project during development. There had been no explicit mention of handling multiple policies. As part of a drive for extensibility, I built the system to cope with multiple policies in memory, and the ability to view multiple policies at the same time in a policy repository. This policy repository essentially has become a hub for any future tools to gain access to the policy set. For example any analysis tool wishing to obtain the set of policies in the current session can simply request them from the policy repository. Since we require the ability to save a single policy to disk, we also require the ability to save multiple policies to disk, this feature was quick to implement but again was not explicitly mentioned.

8.2 Usability

8.2.1 GUI

The GUI provides all the basic functionality that SOAPWADI needs to meet the requirements. To live up to the name of our project, we expect the interface to be decent, that is easy to use, and intuitive. Our adoption of the wizard interface was very effective since it forces the user to perform the stages in the correct order, and secondly makes sure only relevant areas of the view were displayed. We payed particular attention to the enable and disable state of widgets, ensuring that a user could only proceed with an operation if all the relevant fields were filled out. This is something which a lot of applications lack, not necessarily in the field of security. We observe an example of this in Figure 24.

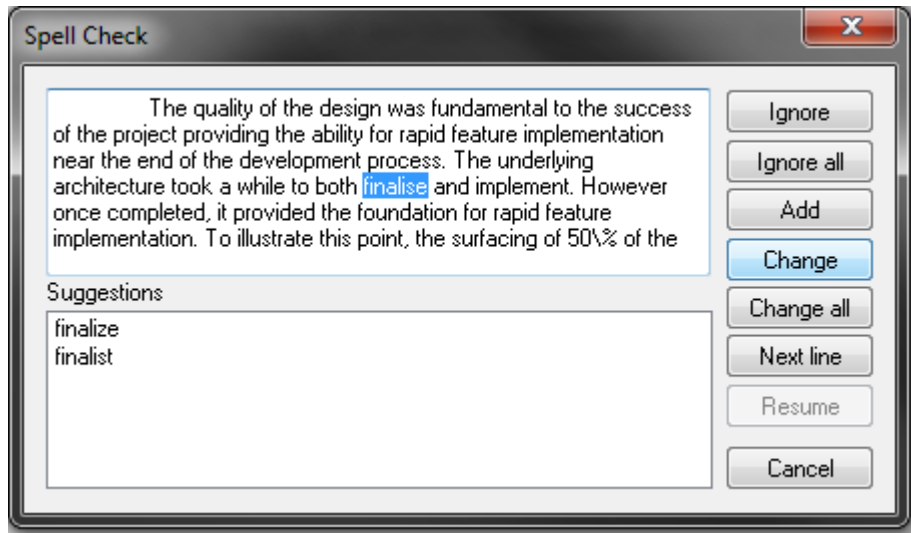


Figure 24: TeXnicCenter illustrates bad GUI design.

We see that the change button is enabled, yet no selection has been made on the suggestions. This is confusing since in some systems this causes the first element to be selected, in this element, no change occurs when the Change button is clicked. The change button should be disabled. We have been careful not to overload the user with functionality. The wizard once again serves to break up the process of specifying policies rather than forcing the user to manage all aspects of the specification process at the same time. We are consistent in our presentation of conditions throughout each stage. We present the user with a list of conditions, and upon selecting a particular condition, the appropriate card appears.

8.2.2 Learning Curve

SOAPWADI was intended to have an intuitive, easy to use, stable, interface. The target users would be experts in the domain of the policies they were specifying, would have some experience in security, for example in specifying policies in English, however would not necessarily be experts in first-order logic. We have seen from user testing that the tool requires some initial explanation. However once this explanation was given, the user was happy using the system. Given that one of the users had no grounding in policy specification, We believe this demonstrates a very shallow learning curve which is the ideal scenario. Of particular importance to this was again the wizard usage, which made it very quick to understand since most computer users are familiar with the concept of a wizard. Areas of difficulty included the concept of instance creation, however once the concept was understood, the system itself was easy to use.

8.2.3 Inability to edit

One of the main problems that arises from having stages 2-4 is the inability to edit. Suppose we specify our scenario classes in stage 1, go through to stage 2 and specify instances, we then bind several fluents in the condition to elements

in the policy head as part of stage 3. Suppose we then decide to go back to stage 1 and add a condition, our system requires that we then go through stages 2 and 3 and set the new condition up appropriately. Suppose however, that we decide to delete a condition from stage 1, our models in stages 2 and 3 are unstable since we have selected and bound elements which may no longer exist. For safety reasons, we discard the entire model, though we need not do this. In the conclusion we outline how we can do better, but for now we simply observe that this inability to properly edit a system is at least time consuming and not what one would expect from a ‘decent interface’. This functionality is regrettable and was never part of the original design plans since stages 2-4 were unforeseen.

8.3 Architecture & Design

This project inherently has a strong Software Engineering bias, relative to mathematical or algorithmic content. I therefore prioritised system architecture and design throughout the development cycle, with large quantities of time spent developing the underlying system. I believe the project has a successful and strong underlying architecture because of this, ensuring an easy to maintain source base with clear modularity providing for future extensibility.

The quality of the design was fundamental to the success of the project providing the ability for rapid feature implementation near the end of the development process. The underlying architecture took a while to both finalise and implement. However once completed, it provided the foundation for rapid feature implementation. To illustrate this point, the surfacing of 50% of the requirements occurred in the short space of a week after spending months on design and architecture. In addition, the extension of the application in numerous ways (e.g. providing free typing natural lanaguage parsing) would be quick to implement because of the patterns implemented.

I found the isolation of different components helped as I made modifications to the design. In particular, the final decision on how to model policies during the different stages took a long time to reach, during which the implementation was very volatile. Despite the fact that the policy model subsystem changed several times the other subsystems remained relatively stable, a sign of relatively low coupling between the different components.

I reduced coupling between controllers by use of the Eventbus system. Methods were called by firing signal objects rather than calling them directly. Also I applied the expert principle wherever possible, for example if a HoldsPermissionFluent object needed to be created, the FluentInstancePermission object which had the necessary information would create it.

8.3.1 Model View Controller

By adhering to the model view controller, we have isolated our views from our models. This means that at some point in the future, we should decide to change the GUI framework from Swing to Flex to attain a more graphically enticing experience, this could easily be done by swapping out the view components.

Should we find the need to change the model as the applications tool set grows and requirements change, we can also do this. This independence of view and model is highly desirable.

8.3.2 Strategy

The strategy pattern has appeared in several places throughout the project with the purpose of providing extensibility and flexibility. This pattern has been crucial to rooting out unnecessary ‘if’ statements and to using runtime polymorphism to change behavior rather than cheap alternatives such as assigning string names to objects and comparing them later. This has made the project not only extensible, but more robust, and maintainable (The string solution for example carries errors on matching case in strings and accidental namespace collisions of strings.) The pattern is used not only to change tools at the outset, but also, crucially, to permit the runtime interchanging of conditions. To an outsider, all conditions appear the same way, they are all printable, they are all displayable, however the details of the condition, the construction of fluents is all delegated to the concrete strategy, which the rest of the system need not concern itself with. This is highly desirable behavior since we can now freely plug in additional condition strategies without opening existing code. That is, we have adhered to the open-close principle.

8.3.3 Extensibility

One of the objectives was to attain a pluggable architecture. There are varying levels of pluggable architectures. If we look at Eclipse [4], for example, a provider can publish a plug-in, which any user can plug in and use straight away without having to recompile. This for us would have been the ideal system, however we did not achieve this. What we did achieve however is not that far off: provided the user implements a simple ‘Tool’ interface, all that needs to be done to import a new tool, is copy the packages over to the project, and add 1 line of code to the ToolsController before recompiling:

```
addTool(new MyTool());
```

This I consider to be a very simple operation, though not as flexible as being able to drop pre-compiled code into a folder, it has been deemed acceptable by the user. We note that JAVA provides a ServiceLoader system which could as part of a future modification be used to provide eclipse like plug-in functionality. We simply load the pre-compiled classes in at runtime under the guise of a Tool interface (which we already have) by using the following snippet.

```
private ServiceLoader<Tool> loader;
public ToolsController() {
    loader = ServiceLoader.load(Tool.class);
}
```

We can then access any tool we like from the loader array. Although we have not implemented this, we can see that our design is very close and minimal changes would be required to implement this system, since we already provide a tools interface.

8.4 Choice of Languages & Tools

8.4.1 JAVA

The decision to use Java helped to speed up development of the application. From past experience, development in languages such as C++, has resulted in more time spent dealing with the intricacies of the language than with the nature of the problem. Java helped with this by managing memory via automatic garbage collection. Since our application is not particularly memory hungry and does not require direct control over memory (it is not a resource/time critical application), this was useful. The cross platform nature of JAVA will be of some benefit since any expansion into an analysis tool will require the use of Sicstus Prolog [5] which itself is available on multiple platforms. This would allow any serious further development of this tool as a commercial project to target multiple platforms or at least Windows and Linux.

8.4.2 Swing

Swing offers a tight integration with Netbeans, this allowed rapid development of the GUI and subsequent easy modification and maintenance. My previous experience with GUI management has been with Qt [6]. Qt at the time offered a Beta version of a form designer which although useful for designing forms, was not very effective at maintaining them. This resulted in manual design of the forms which was a tedious and slow task. In this project however, the Netbeans form designer has been one of the greatest assets since not only could initial designs be made, but future alterations were easy to carry out. I believe the adjustments made could not have been carried out in the timeframe under a manual system. The swing layout managers were particularly difficult to deal with at first, since there are few examples of how to layout widgets under a gridbag system. However I managed to bypass the use of gridbag system via a combination of Border layouts, and static Grid layouts, which resulted in a nice manageable layout system which scaled to any reasonable screensize while keeping widgets visible. One of the disappointments in swing is the difficulty in customizing widgets. For example if I wanted to seek out an alternative to instance representation in stage 2 by colouring matching elements instead of writing an identifying number, this would be difficult to achieve, further to this is an issue with indicating which widgets are selected. However overall, given the time constraints, swing was an ideal choice since it has enabled me to strike a balance between the time taken to learn the framework versus the feature rich nature of a framework. Choosing a different GUI Framework such as flex would have entailed a steeper learning curve, and though the interface may have been better looking, it would probably have resulted in even less of the requirements being met.

8.4.3 Event Bus

The event bus has been a particularly useful software implementation component. It has solved the problem of how to get a large number of controllers

to communicate with each other in a decoupled manner whilst maintaining a flexible approach to development. It is important to note however that the event bus must be used appropriately and it is not a panacea to be used as a replacement for the observer pattern or to replace standard method calls. We note that overuse of the eventbus can be dangerous, we cover this in more detail in section 9.2

8.5 Expressiveness

One of the most important aspects of this tool is its ability to express policies so we must make sure that the tool meets the expressibility of the underlying system. We attempt to reason informally about why the front end is capable of expressing any policy we can express by directly writing fluents.

To show that any policy can be written using the tool can be written directly and visa versa it is sufficient to show that:

1. We can write down any literals we want
2. We can write down any constraints we want
3. We can bind any fluents to any slot.

If a policy element is present in a policy, it is there because it is

- A) An explicit part of the policy. (see Section 2.6.8)
- B) An implicit part of the policy. (see Section 2.6.8)
- C) A policy type predicate (Permitted, obliged, denied)

As part of stage 1, we provide the capability to produce all explicit policy elements. These will manifest themselves in the head of the policy, or in some condition. Since every element explicitly mentioned has a scenarioclass associated, and scenarioclass objects always result in the creation of a fluent, we conclude that we can justify the existence of any explicit fluents present (Part A).

As part of stage 2, we provide for the binding of identifying variables (Project(P), or Project(P1)), to the fluents. We restrict the number of instantiations to the number of references made in stage 1. This will provide all the elements required for explicit policy elements. This shows that we can bind at least the first parameter of any fluent as we see fit.

As part of stage 3 we list all the bindings slots and their respective descriptions to the user (except for ID slots which were bound in stage 2). The list of fluents available to bind is initially restricted to those elements in the policy, however we also allow the user to add additional Peripheral fluents. And so we can say that any fluent variable can be bound to any other fluent variable. This shows that we can bind the remaining non-time slots of any fluent.

On time, the process of binding is a more direct one, there is a 1-1 mapping between the fluents in the system requiring a time binding, and the widgets

visible on the screen. On this basis the user has the opportunity to bind time to every required element.

By the above information reasoning I conclude that though the user is shielded from the logic, the interface does not limit the user in terms of expressibility. That is, anything you could write by hand, can be written using the tool.

8.5.1 Capture of Action Relation

The actions listed in any action box depend on the currently selected target. If we look at the real world model for actions, we see that the action isn't an ability of a target, but rather a shared ability between a Subject and a Target. We define the following terms:

An actions list is **complete** if every possible action which can be done on the target is a member of the list.

An actions list is **(strongly) accurate** if there are no actions in the list which cannot be done by a subject on a target.

An actions list is **weakly accurate** if there is at least 1 action not in the list which cannot be done by a subject on a target.

Figure 25 shows a ven diagram where section A represents all actions which a subject can perform, C represents all actions which we can perform on a target, B represents the intersection of these actions (those which can be performed by a subject and can be performed on a target). We see that our model is complete, and weakly accurate. This means it is possible for us to select an action which makes no sense in the context of the subject, these are areas B and C in Figure 25. Ideally we would like our model to be strongly accurate and complete. For this to occur we need to find a way to related actions to both subjects and targets. To do this we could leave our action definitions in the scenario file as they are, but instead when making references from subjects or targets, have a pair of actions list. 'Actions To' and 'Actions From'. 'Actions To' list contains the actions which can be done on another object. 'Actions From' list contains the actions which can be done on an object. We would then only display the interesection of 'Actions To' and 'Actions From' elements on screen - area B only in Figure 25.

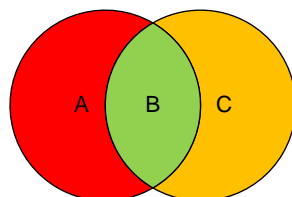


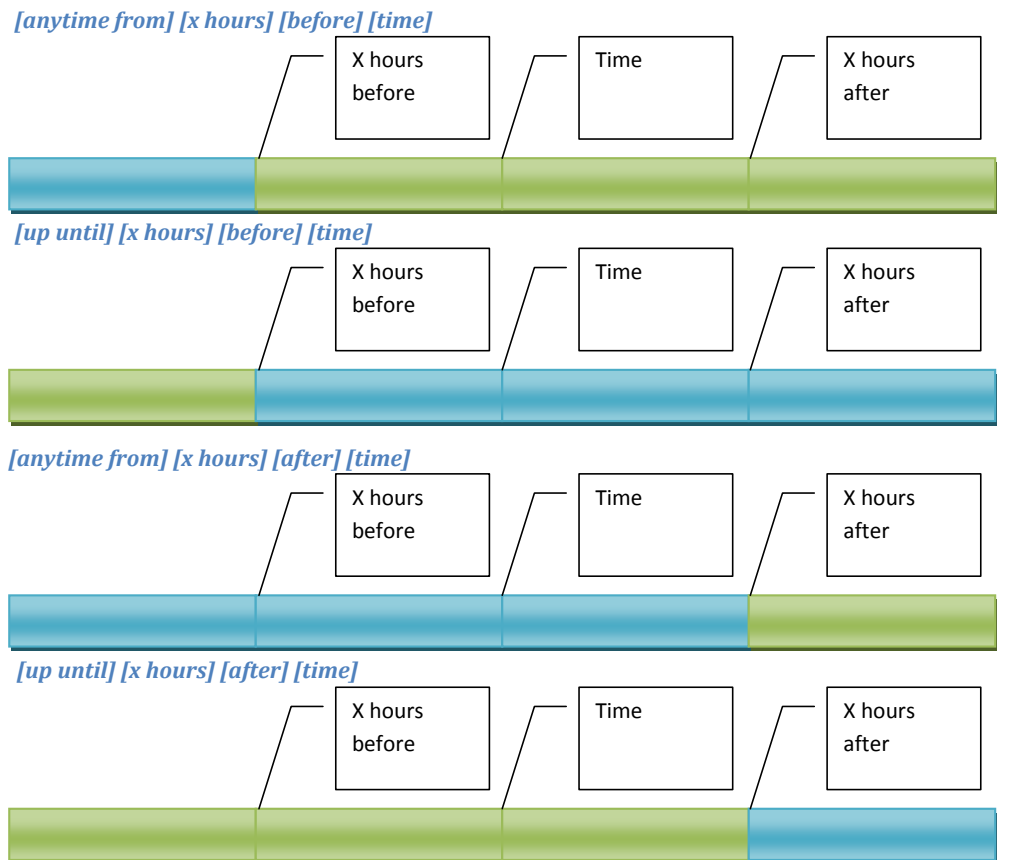
Figure 25: Ven diagram showing A - actions from subject, B intersection A-C, C - actions to target

8.6 Time Criticisms

One of the areas I am most critical about this project is the way we were unable to completely insulate the user from the underlying logic in the final stage. Having successfully insulated them at stages 1-3, it seems a shame that the last hurdle of time binding resulted in direct exposure to Time variables and Time expressions. I outline a system which was intended to be integrated with stage 1, until we realised that stages 2-4 were required.

8.6.1 English time descriptives

First we try to enumerate all the types of time descriptions we could possibly want. We do this informally by looking at a time line, and colouring in sections of it, then thinking of the equivalent english descriptive. The first set shown in Figure 26 are the bounded at one side time descriptions.



at the latest 24 hours after

T1
T2
T3

Before which, subject is not permitted

Figure 26: Time descriptions which are bounded at one side.

The second set set shown in Figure 27 are the point in time based descriptions.

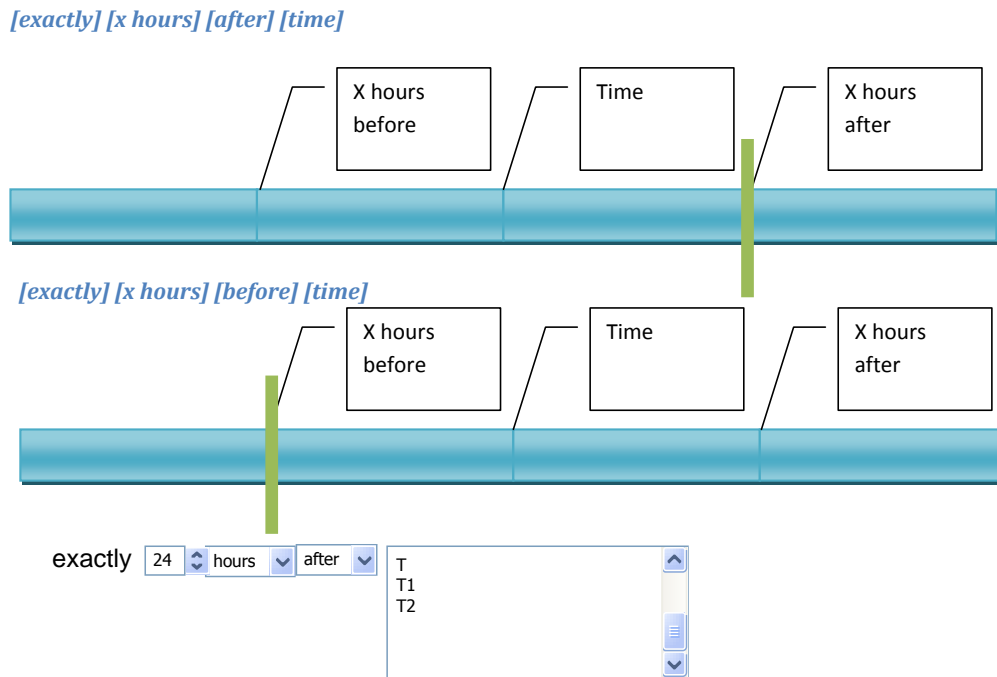


Figure 27: Time descriptions which are based on points in time.

The second set set shown in Figure 28 are the double bounded descriptions.

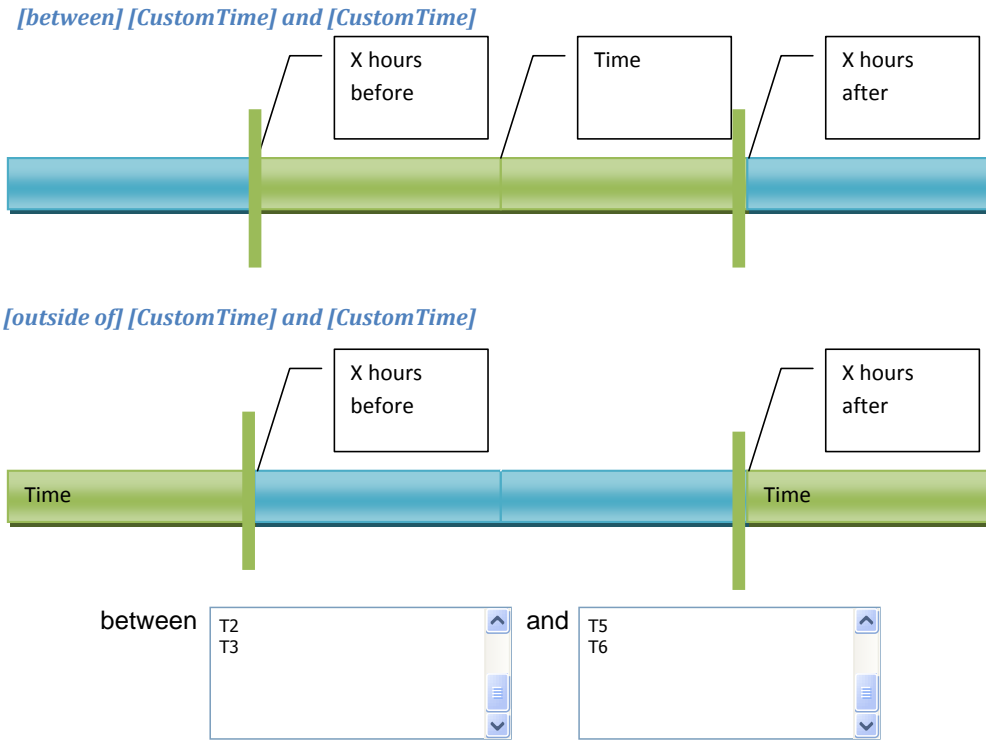


Figure 28: Time descriptions which are bounded at both sides.

We could build a template system based on the descriptions given in the above diagrams and this would certainly allow us bind a time variable to a predicate without exposing the user to the underlying logic.

For example in 28 the first example would bind variable T_1 to the predicate in question, and produce the following constraints: $T_3 < T_1 < T_2$

However the problem comes when relating other time variables to each other. For example, if we later pick the first example from 28 again, we would end up with another binding, this time on T_4 and the following expressions: $T_6 < T_4 < T_5$

However we have not related T_2, T_3 and T_5, T_6 . We need to do this, yet it will be difficult to provide a template to the user which allows them to do this.

For example how would we enable the user to say that $T_2 < T_5$? We could present the user with figures 28, 26, 27 and allow them to move the bound lines relative to each other. Effectively showing the different times on a timeline and allowing the user to move them relative to each other. However it will be difficult for the system to interpret the users input, since such a diagram would describe absolute time constraints, and we may wish to constrain T_2 and T_5 , but may wish to leave some other pair T_3 and T_6 unconstrained. We leave this as an idea for future work.

8.7 Future Work

We have designed and developed a framework containing a specification tool, which I regard to be a great success. However, we were not able to find time to develop other aspects of the project such as analysis tools or extensions to support the input of policies using natural language. I now present a detailed description of how such tools could be designed for a future project.

8.7.1 Natural Language

The Natural Language tools would enable a user to enter policies using the keyboard rather than by selecting from elements on screen. The grammars would not be completely unrestricted, however there would be a greater variety of grammar structures available, in the hope that this would make the English more readable. As part of the original design plans in the specification tool, we did not foresee the need for stages 2-4, we hoped that these could be somehow automatically derived from the structure of a template. However it became clear that the degree of variation in templates made this impractical. Had it been the case that we only needed stage 1, then Natural Language would have been a simple extension whereby we parsed the text, identified the elements, and mapped them directly to our template system, essentially we would be performing a transformation on the Natural Language text to turn it into our Specification tool template form. Once in this form, we could use the existing engine to automatically generate the policy. However we do have stages 2-4, and we have shown they are required since we cannot infer identical instances and distinct instances just from specifying the class alone, and this means that any natural language system would have to be augmented somehow with special parsing techniques which recognize the use of ‘it’ and ‘they’ in reference to identical policy elements and ‘another’ when making distinctions.

8.7.2 Visual Policy Design

One of the problems with stage 2 is that distinguishing between instances is done via a text indication i.e. an instance number after the class name (1) (2) (3) etc... One could help the user get round the problem of identifying distinct and identical elements in a policy by providing a graphical representation of the elements, perhaps using a combination of different shapes and colours for the elements as shown in Figure 31.

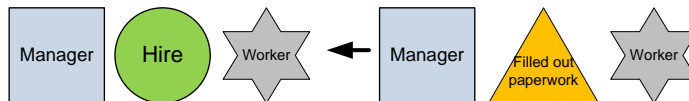


Figure 29: A policy displayed in a visual form using different shapes and colours.

Identical classes would be represented by the same shape, Identical instances would be represented by the same colour. This would allow the user to visually confirm that the policy instances are correct. Of course one has to take into account problems such as running out of shapes, and colour-blind users, however

this would not be a replacement for the existing system, only an extension to aid able users. Identical instances would be represented by the same colour. This would allow the user to visually confirm that the policy instances are correct. Of course one has to take into account problems such as running out of shapes, and colour-blind users, however this would not be a replacement for the existing system, only an extension to aid able users.

8.7.3 Analysis Functionality

Regrettably we were unable to implement the analysis tool. I now outline the steps required to implement such a tool to satisfy the advanced requirements:

Interface We would create a view with several tabs. Each tab would represent a view strategy for a different kind of analysis. Behind each view, a controller with a corresponding strategy would be responsible for acquiring the policies from the policy hub, and communicating with the Prolog interface 3.4.3 in such a way as to query the analysis engine appropriately.

We use modality conflict as an example and show skeleton code for checking the following conflict:

$$\forall T(\neg(\text{permitted}(\text{sub}, \text{tar}, \text{act}, T) \wedge \text{denied}(\text{sub}, \text{tar}, \text{act}, T)))$$

“For all times, it is not true that an action is both permitted and denied.”

```
public class ModalityConflictController {
    ...
    public void submitConflict(){
        SICStus sp;
        SPPredicate pred;
        SPTerm from, to, way;
        SPQuery query;
        int i;
        try {
            PolicyHub.save("policySet.pl");
            sp = new SICStus(argv,null);
            sp.load("policySet.pl");
            sp.load("domainModel.pl");

            pred = new SPPredicate(sp, "permitted", 4, "");
            pred2 = new SPPredicate(sp, "denied", 4, "");
            sub = new SPTerm(sp).putVariable();
            tar = new SPTerm(sp).putVariable();
            act = new SPTerm(sp).putVariable();
            time = new SPTerm(sp).putVariable();

            query = sp.openQuery(pred, new SPTerm[] { sub, tar, act, time });
            query = sp.openQuery(pred2, new SPTerm[] { sub, tar, act, time });
            while (query.nextSolution()) {
                System.out.println(way.toString());
            }
        }
        catch ( Exception e ) {
            ...
        }
    }
}
```

The above code snippet queries the policy system for our modality conflict and prints a set of query results. We can then use the query results to indicate

to the user the subject, target, action and times which were violating correct modality properties. By searching our policy hub we could be able to highlight policies involved.

8.7.4 Editing

In the evaluation we mentioned that the specification tool does not support editing particularly well, also as part of the background work, we commented that instant feedback, and the ability to go back and change things is an important aspect of any tool. One of the problems we have faced is that the existence of stages 2-4 prevents effective editing. Suppose we specify the classes in stage 1, instantiate in stage 2, and enter stage 3 before realising we want to make changes to stage 1, for example to add a condition, we must redo stage 2, and stage 3 for the new condition. Worse yet is the removal of a condition in stage 1, since if any fluents have been bound in stage 3, it is not safe to keep the stage 3 model. Due to time limitations we have done the safest thing which is to discard the model made in stage 3 and force the user to rebuild it. However by noting the dependencies in the system we make the other models safe by removing variables from elements when the parent fluent is discarded.

8.7.5 Visual Policy Binding

Further to the section on visual policy specification, we look at the binding stage and how this could be made into a more visual process. Instead of selecting fluents as text labels on widgets, why not allow the user to draw lines between policy elements to indicate a relationship. For example if our system used the shapes system from section 8.7.2 we could then allow the user to bind elements together using some sort of draw tool to link elements together in the form of a graph.

8.7.6 GUI Improvements

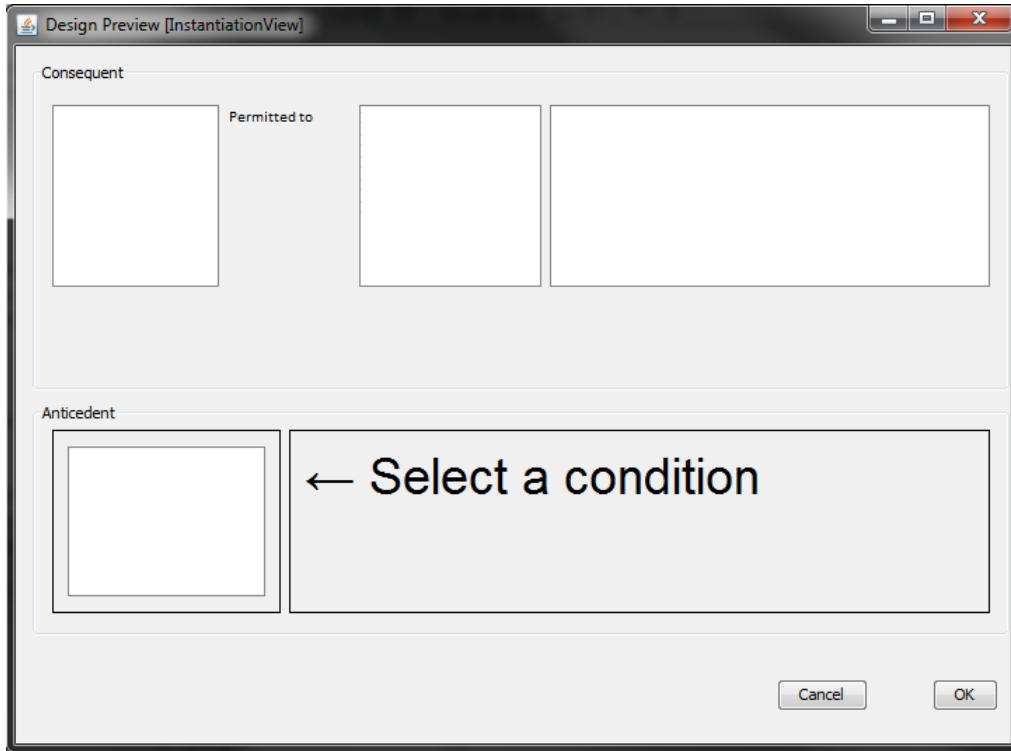


Figure 30: The current policy specification interface.

The GUI was deemed satisfactory during testing, however aspects such as changing page on the wizard could be made smoother. Apple have a reputation for making easy to use devices, and one particular strength noted is their record on good interface design. Widgets on products such as the iPhone bare little or no resemblance to those found on a windows system. We could improve the interface of SOAPWADI by choosing components better suited to represent the objects they are. For example if we take any of the windows which contains policy elements as in Figure 30 We note that there is nothing about this window which suggests that a policy is contained. The top half doesn't look like a policy head, and other than the words Consequent and Conditions, there is nothing on first glance to suggest that we are looking at a policy. Let's ask ourselves "What does a policy look like?" It's very difficult to answer this since a policy doesn't take physical form, but we can certainly try to move away from the "Windows" style nightmare. Let's look at an idealistic alternative:

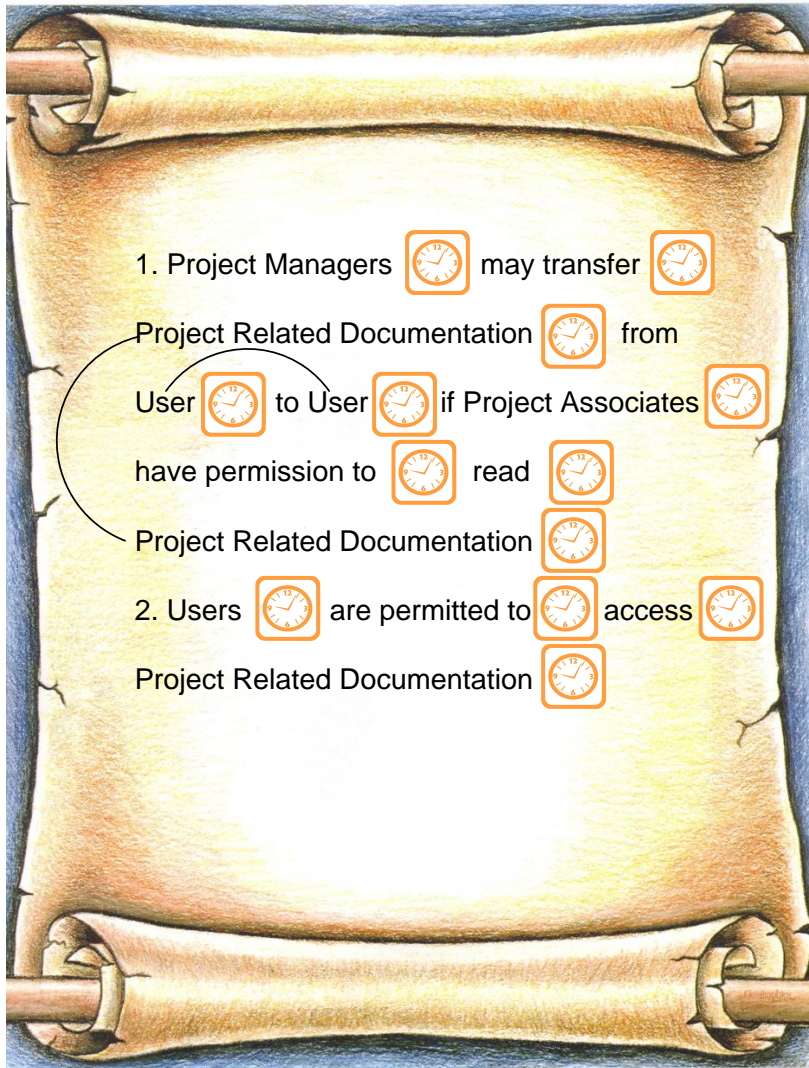


Figure 31: The perfect policy specification interface.

in Figure 31 we have a text based system where each element of text is interactive. Writing out the policy is the stage 1 definition. Double-clicking an element causes the element colour to change, which distinguishes it from other similarly named elements in the policy. This provides us with stage 2. Clicking on an element causes a line to be drawn from it, clicking again on another element causes the line to link the two elements, this effectively performs stage 3 for us. Note the use of clocks after each element which results in a fluent bound by time, this allows us to set stage 4. We have effectively made our system look more like a policy specification system by avoiding ugly windows based components such as lists selections and buttons. The user should feel more at home editing something which looks like pen and paper.

9 Conclusions

9.1 Time Management

We see from the requirements that all of the analysis requirements have been pushed off the implementation radar. A number of factors contributed to this which I list here:

- Extensive blackboxing of certain aspects of the specification system. These blackboxed components such as time integration were deemed trivial but turned out not to be so. Many of these black boxed components eventually resulted in the creation of Stages 2-4.
- The uncovering of stages 2-4.
- Underestimate of the time taken to build conditions.
- Wrong design approach to stage 4.
- A misunderstanding of the underlying logic which led to the belief that fluents bound to a permission predicate did not required parenting with a HoldsFluent and time variable.

The uncovering of stages 2-4 can effectively be shown as the doubling or tripling of the project duration, since entire algorithms for the same areas covered in stage 1 (including all conditions to be handled) needed to be designed. A mistake made in the design of stage 3 where it was thought that fluents could be visually grouped into a single model for binding, when in fact we needed to maintain segregation in order to allow the user to distinguish between similar elements in different areas of the policy. Lastly on point 4 the result of this misunderstanding was that when we mentioned a particular class in stage 1 we would create exactly one fluent, instead of creating one fluent for each reference in the stage 1 template. This led to unnecessary complex code being written, most of which was scrapped. It could be said that this was a misunderstanding in requirements, and this only goes to demonstrate that the cost of fixing requirements at implementation stage is much greater than at the design stage.

9.2 Appropriate EventBus usage

The event bus library is not a panacea for all coupling problems, it requires careful use. In particular one should only use it on single instance objects or objects where duplicate responses do not matter. Wiring is done at design time, and it is not possible to unwire objects without turning the EventBus pattern into a glorified Observer pattern. In particular since we cannot delete objects and must rely on garbage collection instead, it is important to note that old models and objects can interfere with new objects by responding to messages broadcast on the Event Bus. For this reason we recommend only using the event bus for its original purpose of decoupling controllers from each other. We also recommend that models should never contain Event Bus methods, since models are objects which are likely to be persevered throughout the life time of

the application but yet inactive at the same time (Lists of policies other than the current policy etc...)

9.3 Policy Specification

This project shows that there are 4 main unavoidable aspects to translating an English template into logic.

1. Class Selection
2. Fluent Instantiation
3. Variable Binding
4. Time Variable Binding

However, the way in which the user performs these tasks need not be segregated as in our own system. The key to making specification as easy as possible is avoiding the segregation. This will achieve a better editing process, and hence make policy writing easier. From our feedback section, editing (or lack of) is clearly the greatest issue affecting the application closely followed by differentiating between different instances of fluents in stage 2. However we have achieved what we set out to do with respect to specification requirements and requirements for building an extensible framework. The careful use of design patterns and good overall architecture will allow any future extensions to be built rapidly and integrated easily.

References

- [1] Junit3. <http://junit.sourceforge.net/junit3.8.1/index.html>.
- [2] Junit4. <http://junit.sourceforge.net/>.
- [3] Test driven, 2001. <http://www.testdriven.com/modules/mylinks/singlelink.php?lid=911>.
- [4] Notes on the eclipse plug-in architecture, 2003. http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html.
- [5] Sicstus prolog 4, 2003. <http://www.sics.se/isl/sicstuswww/site/index.html>.
- [6] Qt framework, 2010. <http://qt.nokia.com/>.
- [7] Spring source, 2010. <http://www.springsource.org/>.
- [8] Aleks Aris and Ben Shneiderman. Designing semantic substrates for visual network exploration. *Information Visualization*, 6(4):281–300, 2007.
- [9] F. Baader and W. Snyder. Unification theory, 2001. <http://lat.inf.tu-dresden.de/research/papers/2001/BaaderSnyderHandbook.ps.gz>.
- [10] David Botta, Rodrigo Werlinger, André Gagné, Konstantin Beznosov, Lee Iverson, Sidney Fels, and Brian D. Fisher. Towards understanding it security professionals and their tools, 2007.
- [11] Carolyn A. Brodie. An empirical study of natural language parsing of privacy policy rules using the sparcle policy workbench.
- [12] Gavin A. Campbell. Overview of policy-based management using poppet, 2006. <http://hdl.handle.net/1893/1608>.
- [13] Jeff Canna. Testing, fun? really?, 2001. <http://www.ibm.com/developerworks/library/j-test.html>.
- [14] Alison Cawsey. More about prolog matching, 2001. http://www.macs.hw.ac.uk/~alison/ai3notes/subsectionstar2_3_3_2.html.
- [15] Juri Luca De Coi. Controlled natural language policies.
- [16] Robert Craven, Jorge Lobo, Jiefei Ma, Alessandra Russo, Emil C. Lupu, and Arosha K. Bandara. Expressive policy analysis with enhanced system dynamicity. In Wanqing Li, Willy Susilo, Udaya Kiran Tupakula, Reihaneh Safavi-Naini, and Vijay Varadharajan, editors, *ASIACCS*, pages 239–250. ACM, 2009.

- [17] J. Hosang D. Lawrence A. Slater D. Burrell, M. Gist. Distributed doodling. <http://www.daniel-burrell.com/Tabula-DistributedDoodlingFinalReport.pdf>, 2010.
- [18] Amy Fowler. A swing architecture overview, 2010. <http://java.sun.com/products/jfc/tsc/articles/architecture/#separable>.
- [19] David Geary. Strategy for success, 2002. <http://www.javaworld.com/javaworld/jw-04-2002/jw-0426-designpatterns.html>.
- [20] Dorothy Graham, Erik Van Veenendaal, Isabel Evans, and Rex Black. *Foundations of Software Testing: ISTQB Certification*. Intl Thomson Business Pr, 2008.
- [21] J.P.E. Hodgson. Prolog iso standard. <http://pauillac.inria.fr/~deransar/prolog/docs.html>.
- [22] IBM. Unstructured information management architecture. www.research.ibm.com/UIMA/.
- [23] IBM. Functional testing, 2001. <http://www-01.ibm.com/software/rational/offerings/quality/functional.html>.
- [24] Eser Kandogan and Eben M. Haber. Security administration tools and practices., 2005.
- [25] Clare-Marie Karat, John Karat, Carolyn Brodie, and Jinjuan Feng. Evaluating interfaces for privacy policy rule authoring. pages 83–92, 2006.
- [26] Intelligent Systems Laboratory. Mixing java and prolog, 1998. http://www.sics.se/sicstus/docs/3.7.1/html/sicstus_12.html.
- [27] Bertrand Meyer. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997.
- [28] michaelbushe. Event bus, 2010. <http://www.eventbus.org/>.
- [29] Mike Potel. Mvp: Model-view-presenter; the taligent programming model for c++ and java, 1996. <http://www.wildcrest.com/Potel/Portfolio/mvp.pdf>.
- [30] Robert W. Reeder, Lujo Bauer, Lorrie Faith Cranor, Michael K. Reiter, Kelli Bacon, Keisha How, and Heather Strong. Expandable grids for visualizing and authoring computer security policies, 2008.
- [31] Marc De Scheemaeker. Nanoxml, 2008. <http://nanoxml.sourceforge.net/orig/>.
- [32] Morris Sloman. Policy driven management for distributed systems. *J. Network Syst. Manage.*, 2(4), 1994.

- [33] Ph.D. Steve Burbeck. Applications programming in smalltalk-80(tm): How to use model-view-controller (mvc), 1987. <http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html>.
- [34] Sun Systems. Class observable, 2010. <http://java.sun.com/javase/6/docs/api/java/util/Observable.html>.
- [35] Kenneth J. Turner. The accent policy wizard, 2009. <http://hdl.handle.net/1893/1608>.
- [36] Nachi Ueno, Ryota Hashimoto, Michio Shimomura, and Kenji Takahashi. Soramame: what you see is what you control access control user interface, 2009.
- [37] Stanford University. Protege ontology editor. <http://protege.stanford.edu/>.
- [38] Kami Vaniea, Clare-Marie Karat, Joshua B. Gross, John Karat, and Carolyn Brodie. Evaluating assistance of natural language policy authoring. pages 65–73, 2008.
- [39] Lars Vogel. Dependency injection, 2008. <http://www.vogella.de/articles/SpringDependencyInjection/article.html#dependencyinjection>.
- [40] IBM W3C John M. Boyer. Xforms for html, 2008. <http://www.w3.org/TR/2008/WD-XForms-for-HTML-20081219/>.
- [41] Wenjuan Xu, Mohamed Shehab, and Gail-Joon Ahn. Visualization based policy analysis: case study in selinux. pages 165–174, 2008.