An Interoperability Framework For Security Policy Languages

Amir Aryanpour

An Interoperability Framework For Security Policy Languages

By

Amir Aryanpour

A thesis submitted to the University of Bedfordshire in partial fulfilment of the requirements for the degree of Doctor of Philosophy

Submission date: 01-04-2015

Abstract

Security policies are widely used across the IT industry in order to secure environments. Firewalls, routers, enterprise application or even operating systems like Windows and Unix are all using security policies to some extent in order to secure certain components.

In order to automate enforcement of security policies, security policy languages have been introduced. Security policy languages that are classified as computer software, like many other programming languages have been revolutionised during the last decade. A number of security policy languages have been introduced in the industry in order to tackle a specific business requirements. Not to mention each of these security policy languages themselves evolved and enhanced during the last few years.

Having said that, a quick research on security policy languages shows that the industry suffers from the lack of a framework for security policy languages. Such a framework would facilitate the management of security policies from an abstract point. In order to achieve that specific goal, the framework utilises an abstract security policy language that is independent of existing security policy languages yet capable of expressing policies written in those languages.

Usage of interoperability framework for security policy languages as described above comes with major benefits that are categorised into two levels: short and long-term benefits. In short-term, industry and in particular multi-dimensional organisations that make use of multiple domains for different purposes would lower their security related costs by managing their security policies that are stretched across their environment and often managed locally. In the long term, usage of abstract security policy language that is independent of any existing security policy languages, gradually paves the way for standardising security policy languages. A goal that seems unreachable at this moment of time.

Taking the above facts into account, the aim of this research is to introduce and develop a novel framework for security policy languages. Using such a framework

would allow multi-dimensional organisations to use an abstract policy language to orchestrate all security policies from a single point, which could then be propagated across their environment. In addition, using such a framework would help security administrators to learn and use only one single, common abstract language to describe and model their environment(s).

Contents

Acknowledgement

First and foremost, I would like to express my gratitude to my first supervisor Prof. Yan. His encouragement, patient guidance and indeed critical feedback shaped the entire research.

I would also like to offer my deepest appreciation to my second and third supervisors Prof. Parkash and Dr. Aydin who unconditionally supported me at crucial stages of my PhD study. Without their help and support, I could not have finished this study, without a doubt.

I am grateful to Capgemini UK, for granting me a research sponsorship. In particular, I owe a great debt to my managers and colleagues including Mr. Scott Davies, Mr. Andrew Harmel-law and Mr. John Arnold, who helped me throughout this research. Their invaluable feedback, discussions and support encouraged me to overcome majority of the challenges that I had to face.

Last but not the least, I would like to thank my family for their continuing moral and emotional support and being patient with me throughout the last few years.

Declaration

I declare that the work described in this thesis is my own unaided work for the degree of Doctor of Philosophy at the University of Bedfordshire. It has not been submitted for any degree or in any other University or college of advanced education.

This thesis has been written by me and produced using LaTeX.

Name of candidate: Amir Aryanpour Signature:

Date: 01-04-2015

List of Figures

List of Tables

Chapter 1

Introduction

This introductory chapter presents:

- The research motivation,
- The problems statement,
- The research context,
- The thesis statement (Aims and Objectives),
- The research scope and constraints,
- The thesis contribution,
- Related publications,
- The thesis outline.

1.1 Background

There is no doubt that computer networking has been revolutionised during the last decade. Computer networks, including the internet, are expanding at a rapid pace that make it almost impossible to predict what will be introduced and added to this

phenomenon in the future. Less than a decade ago, it would have been inconceivable that in the near future a user would be able to browse networks using a mobile phone whilst sitting in a car or that people would seamlessly, but securely, store their information on the internet without knowing where their information had been physically stored (i.e. the cloud). The expansion appears to be multi-dimensional. However, irrespective of the type and directions of the expansion, people always have had one concern in common; the *security* of their networks.

Although the expansion of networking is fascinating, there are a number of scenarios in which network users would typically decide to restrict access to and from their networks by other users. For example, when backing up information such as pictures on a cloud based storage, one might consider who can access that information and under what circumstances. Another example is restriction of a personal network router to block delivery of age-restricted content to the home network in order to protect the safety of children. In the above-mentioned examples, and in many other similar scenarios, one is effectively thinking in the same way that a security architect in a multi-domain organisation thinks in regard to restricting access to and from network resources. Whilst on a router, for instance, a few clicks on a pre-defined admin-page could achieve the goal of restricting access to certain websites, the same may not be as easily achievable in multi-domain organisations. Generally speaking, that is why security policies have been introduced.

Very similar to programming languages that facilitate orchestration of a series of actions to achieve a goal by programmers, security policy languages have been introduced to allow coders or rather security administrators, to define their goals to protect a specific domain by taking certain actions in a specific order. Many access control models and policy languages have been proposed in order to address the above mentioned concern. These languages, which have evolved during the past decade, usually come with different specifications and aim to tackle different business requirements. However, irrespective of their type and specification, all security policy languages, from a single Role Based Access Control (RBAC) to a highly sophisticated policy language that is capable of negotiating over the network,

with their own advantages and disadvantages, have one aim in common, that is *to secure resources*.

As with programming languages, it is not easy to distinguish the advantages of one specific security policy language over another. However, unlike programming languages that are used to code an unlimited number of scenarios, the majority of scenarios covered by security policy languages can usually be modelled and described at an abstract level. That summation for access request is simply stated as: 'who can access what, under which circumstances?'

1.2 Motivation

The motivation behind this research can be divided into different categories that are described in the sections below:

1.2.1 Industry Research

Policy-based access control systems are now well established. One way or another, different devices, applications, system users etc. are all restricted by these security policies. The management of these security policies across a multi-domain environment is a challenging task that would require tools/frameworks to assist system security administrators to achieve their goals.

There are similar frameworks provided by different vendors to cover different aspects of enterprises system securities like *Security Authentication Language*. Gluu [21], that provides the industry with a framework for identity and access management control utilising a single point of management, can be given as an example. However, industry suffers from lack of such tools for security policy languages, hence, this research is focused on security policy languages and aims to fill-in the identified gaps within the industry.

In other words, very similar to other frameworks such as Gluu [21], this re-

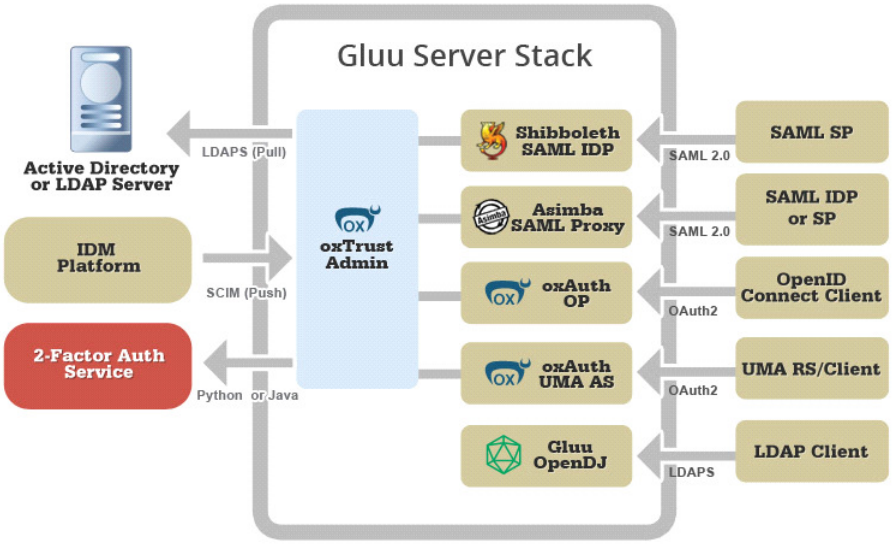


Figure 1.1: Overview of Gluu, a Framework for Identity Access Management [21]

search's goal is to provide security policy users with an abstract standard security policy language, that can be translated to other specific security policy languages. Indeed, in order to achieve this, an *Interoperability Framework for Security Policy Languages* that understands the abstract language and is capable of translating it to specific security policy languages is needed.

1.2.2 Industry Support

In addition to the above, in order to justify the present research, it is necessary to focus on two different sets of facts: on one hand, in todays economy, whilst corporations seek to control costs yet drive productivity, the cost of acquiring and maintaining a company's software is closely scrutinised and controlled. IT departments are under constant pressure to deliver more services in a short span of time with ever decreasing budgets. Hence, IT departments are willing to choose and invest in technologies that provide them with more business values at a lower cost.

On the other hand, the high demand of distributed computing and its multi-dimensional expansion necessitates the increase of security policies and in turn,

security policy language usages in the coming years. As stated above, these policy languages come with different formalisms, specifications, advantages and disadvantages. Yet, while too many of them may be available within the industry, it seems that the usability of these languages cannot easily be challenged. Having said that, and as mentioned before, the majority of the scenarios covered by these languages can be modelled at an abstract level.

To summarise, on one hand, there are a wide range of security policy languages available and needed within the industry, while on the other hand, IT departments are under pressure to control their costs and hence, they are willing to invest in technologies that helps them to achieve such a goal. A combination of these two could justify this research intently that provides the industry with one generic security policy language that helps the industry to manage heterogeneous security policies from single point of management. As we will see in the coming sections, the framework that is provided by this research will help multi-dimensional organisations to significantly reduce their costs.

1.2.3 Research Support

As we will review them in great details in the Chapter 2, it was also noted that few other researches and academic projects aimed to tackle the very same issue, during last few years. Inspired by these contributions, this research aims to improve their work and provide other researchers with a more advanced framework for security policy languages.

1.2.4 Industry Sponsorship/Encouragement

Above all, a series of informal interviews were conducted to obtain more detailed information in the field of this research. A series of highly professional experts have been selected to obtain their invaluable insights on the research. In addition to above it has also been decided to have a mix of expertise from different backgrounds,

with the hope to cover different aspects of the framework. These expert views and guidance paved the way to start this research. A list of these experts who constantly shaped the IT industry is as follows:

- Professor of Security Engineering at the Computer Laboratory of Cambridge University.
- Chief Security Architect at Capgemini UK.
- Manging Software Architect at Capgemini UK.
- Senior Security Consultant at Capgemini UK.
- Senior Security Consultant at IBM UK.

These individuals are referred to as *experts* throughout the research and will reappear in Chapter 3, The framework Overview and Design challenges and in Chapter 7 Implementation of the Framework.

1.3 Problem Statement

Considering the range of security policy languages that are currently available to choose from (more than 20 security policy languages recognised by W3C [24]), inevitably, multidimensional organisations have to use a collection of these languages in their environment. Taking the fact into account that these languages are often used in a single domain, a standard security policy language, in addition to a security policy language framework, is needed to help security architects and security administrators to manage and maintain their secure domains from a single point of administration.

Lack of such a framework as illustrated in Figure 1.2, which has already been noted by other researchers in the past and as a result of that, afew studies and researches have already been carried out for the development of a multipurpose security policy language such as - SecPAL [44]. These researches, however, aimed to

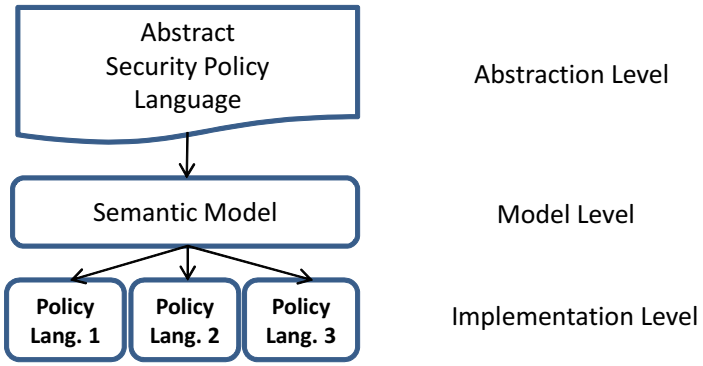


Figure 1.2: Overview of Interoperability Framework for Security Policy Languages

provide a completely new and enhanced security policy language by analysing and improving previous security policy languages. Utilising a completely new security policy language can be considered by new projects within the industry, but this approach is not always welcomed with regards to legacy applications, especially in a secured domain.

Taking the above fact into account, the interoperability framework for security policy language should not only force a legacy system to change or modify the underlying security infrastructure, but it should also use that to a great extent. The usage of this framework should help security experts to achieve their goals more easily and should not impose more constraints on their domains.

Fair and challenging questions can be presented as: *Why is such an abstract language needed? Do policy language users want a standard at all? Can none of the existing languages be adapted more generally?*

In order to answer these questions, the following few facts must be taken into account:

1. Security policy languages have evolved over the last decade and the process is still ongoing.

2. Security policy languages are popular but they are not a compulsory com-

ponent of every infrastructure, hence, they have their own distinctive lists of users.

3. A handful of well-known organisations that have already implemented their security infrastructures are using them to the greatest possible extent. Assuming the infrastructure works as expected and is secure enough, there would be no motivation for the client to change the infrastructure.

Now considering all these facts, these questions can be answered as follows:

Q: Do policy language users want a standard at all ?

A: Probably not at the moment, as the usage of security policy languages is limited to certain organisations and relatively large companies. However, researchers need to be able to predict the demand of predictable tools, technologies, products etc. and to standardise them in order to help both future researchers and industry. Security policy languages can be one of those tools. At the moment, security policy languages come with no unique industry-wide standard, but they do have their own demands in the security arena.

If a standard for security policy languages that provides specific advantages to its users would be desirable in the future, the development of that standard should begin in the present day to encourage both new and existing users. Encouragement of new users could be considered less challenging compared to the existing users. It is believed that by using the framework, even existing users of security policy languages would be able to benefit from the security policy standard without changing their security infrastructure.

Q: Can none of the existing languages be adapted more generally?

A: Generic and academic security policy languages could be considered for such an approach. However, it must be borne in mind that the framework, in addition to its advantages that have already been described has to achieve a very specific goal: to provide future AND current policy language users with an abstract standard policy language.

Using an existing policy language and using it more generally would probably work for new users. However, that would not be acceptable for existing security policy language users.

1.4 Research Context

Protecting networked resources came to life at the very same time when computer networking was introduced. Many access control models and security policy languages have been proposed in order to address the above-mentioned concerns. These languages, which have undergone a revolution during the last decade, usually come with different specifications aiming to tackle different business requirements.

However, these policy languages have often been designed independently and are not interoperable. This lack of interoperability of security policy languages on a distributed network, where different domains use different policy languages, affects the main benefit of policy-based security management the enabling of resources and services be controlled and managed at a high-level regardless of the adopted under-lying policy language.

This research intends to provide security policy language users with a framework to make them interchangeable. The industry would benefit from the usage of such a framework as outlined below:

1.4.1 Short-Term Benefits

Security policy languages are often used in multidimensional organisations that have different requirements for different parts of their networks or so-called sub-domains. These sub-domains often use heterogeneous policy languages because different policy languages are designed and developed to address different business requirements. Policy languages, like programming languages, are typically not easy to learn and understand for occasional users. Thus, the use of different policy lan-

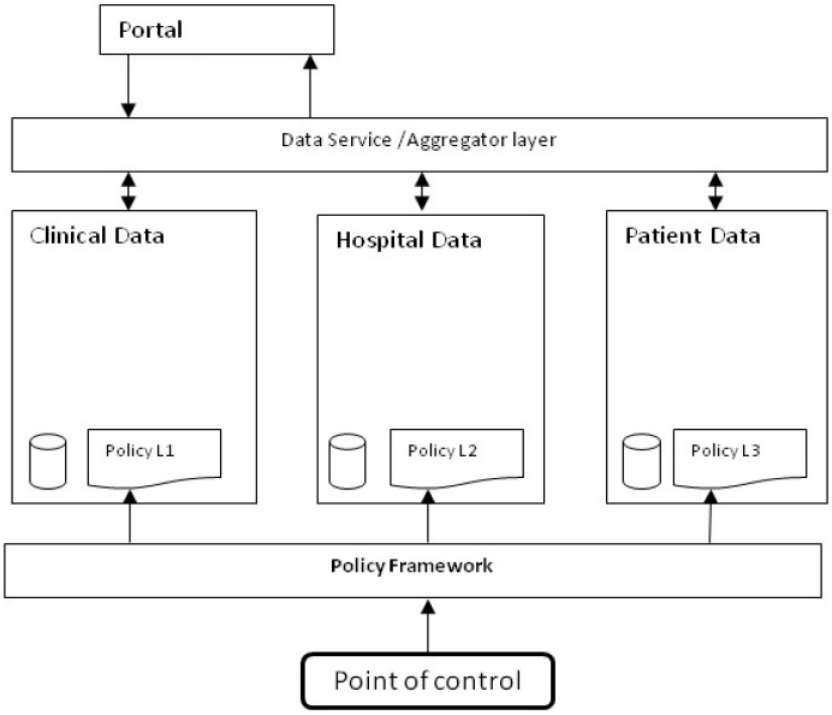


Figure 1.3: A Typical Secure Distributed Environment

guages in a multidimensional organisation imposes the need for security experts, who can code security policies in those security policy languages. In addition, the sub-domains of multi-dimensional organisations are often managed locally. This, in turn, implies that the management of these security domains from a single point is not an easy task to achieve.

Figure 1.3, which has been presented for illustration purposes, shows a typical distributed network that utilises heterogeneous security policy languages across different sub-domains. In such an environment, the management of all sub-domains at an abstract level is challenging if not impossible, unless an abstract management policy framework to control the environment from a single point is introduced.

Using such a framework would allow multidimensional organisations to use an abstract security policy language to orchestrate all the security scenarios at an abstract level and from a single point, that can then be propagated across the environment. In addition, using such a framework would help security administrators

to learn and use a single and common abstract language to describe and model their environment(s).

1.4.2 Long-Term Benefits

Rapid expansion of computer networks, necessitated the introduction of a set of different security policy languages with different formalisms and models to the market [24]. Taking the mentioned facts into account, it can be predictable that the future expansion of computer networks and their requirements will demand even more new security policy languages to be introduced. As a result, in the near future, there will be too many policy languages without any intersection and commonality that are all needed and required within the industry.

Using the proposed framework will help security architects and security administrators by eliminating the requirement of learning how to code security policies in different policy languages. The proposed framework, which understands the security domains, provides users with a much simpler language that maintains the orthogonality of the security system. The standard security policy language that works in conjunction with the framework for security policy language will gradually become more mature and indeed popular and hopefully will lead the industry to use an abstract standard security policy language in future.

1.5 Thesis Statement (Aims and Objectives)

Aims

As stated in the previous section, security policies are often scattered over different domains and environments on multidimensional organisations. These security policies come with their own models and domains of administration. Moreover, some applications within these organisations often need to cross several of these security domains. Such a task would be very difficult to accomplish when these policies are

scattered over the domains. Having the infrastructure of the above organisations in mind, the primary aim of the present research is to provide an interoperability framework for management and administration of heterogeneous security policy languages from a single point, across such an organisation.

The way that users interact with the framework would be via an abstract security policy language perhaps through a user interface. Hence, the secondary aim of the research will be defined as to provide security experts with a human-readable abstract and standard security policy language that is independent of underlying security infrastructure.

Objectives

The research objectives can be categorised in two sections:

A) Theoretical Objectives:

The diversity of the security policy languages urges to shortlist a handful of these languages, as candidate languages. Hence, the first set of objectives can be defined as:

1. To investigate commonalities and differences of security policy languages from different perspectives and categorising them accordingly.
2. To define the project-specific criteria for shortlisting security policy languages. The task will be executed following the review of security policy languages (i.e. task 1).

   A similar task also has to be performed against algebra for security policy languages:
3. To provide formalism for the framework and evaluate it against security policy language candidates. The task will be executed following the literature review of presented algebra for security policy languages.

B) Practical Objectives:

1. To design a framework for security policy languages using appropriate software development methodologies.
2. To implement a Proof of Concept (PoC) using open source components according to software development best practices.
3. To design, develop and enhance an abstract security policy language for the framework.
4. To evaluate the framework in accordance with well-known test strategies.

1.5.1 Scope and Limitations

As has been mentioned in previous section, this research will provide a framework for security policy languages. Although security policy language candidates will be briefly reviewed, their technical details including infrastructure, architecture, formalisms and specifications will not be discussed in detail in this paper and indeed, would be beyond the scope of this research. Existence of these security policy languages has been assumed in advance.

In addition, various open source software application and frameworks have been used throughout the design and development of the framework. A complete list of these components, including their versions used will be provided in Appendix A. Any improvement or enhancement of these components by their vendors that becomes available after the version that was used which directly or indirectly impacts the performance and/or behaviour of the proposed framework will be excluded from this research and would be beyond the scope of this study.

Moreover, a great deal of attention has been given to the design and implementation of the system in accordance with best software development practice. Having said that, the entire design, implementation and evaluation of the framework was performed in laboratory strictly for academic purpose. As a result, the framework should not be used in real world production environments without proper testing

and perhaps improvement.

1.5.2 Research Methodologies

Throughout this study the following research methodologies have been used:

I. **Interview**

A series of simple yet effective interviews were conducted before the research commenced and throughout the research. The majority of interviewees were well-known individuals within the Information Technology (IT) industry with invaluable insights on the future of IT. The results of these interviews shaped the structure of this research from the outset.

II. **Literature Study**

At an early stage of the research, it was decided to mathematically prove whether transition/conversion/translation of security policies is doable. Logically, the most appropriate way to translate many-to-many languages is to translate them to and from an abstract language. Algebra was chosen as the abstract language, or rather the framework, throughout this research. Hence, a literature review of existing algebra for security policy languages was performed.

In addition to that, it was necessary to categorise security policy languages. Accordingly, a literature study of existing security policy languages was also carried out.

III. **Proof of Concept**

Proof of concept, which is also known as prototyping, was used to implement a limited version of the framework in order to evaluate its outcome and its behaviour against real world scenarios. Proof of concept has also been used to test the framework with real users.

IV. **Experiment**

Real users have been asked to interact with the framework prototype and their interactions with the system was observed and surveyed. Their suggestions and comments have also been fed back to the system developers in an iterative manner, in order to improve system performance, behaviour and user-friendliness

V. **Survey**

In order to make the users' feedback procedure easier and indeed more formal, a simple but effective survey, with a combination of multiple-choices questions and suggestion box, was designed and distributed to the users.

1.6 Research Contribution

The main contribution of this research, as appears from the title, is *to implement an interoperability framework for security policy languages*. However, the following can also be considered as contributions of this research:

1. A literature review of security policy languages is provided. It includes a complete comparison of security policy languages backed up by up-to-date discussions and references. In addition to that, it was shown how to define a completely new set of requirements for a customised security policy language comparison.

2. A literature review of algebra for security policy languages and their characteristics, formalisms and specifications is provided.

3. A new algebra for security policy languages is presented and evaluated against real world scenarios.

4. A completely new definition for an abstract standard security policy language is presented. As the abstract language evolves, it will improve the usability of security policy languages.

1.7 Related Publications

1.7.1 Conferences

1. Amir Aryanpour, Song Y. Yan, Scott Davies, Andrew Harmel-law. Towards Design an Interoperability Framework for Security Policy Languages. In Proceedings of 12th International Conference on Security and Management(SAM12). 16-19 July 2012, Las Vegas, USA.

2. Amir Aryanpour, Edmond Parakash, Scott Davies, Andrew Harmel-law. An Interoperability Framework for Security Policy Languages. In Proceedings of 14th International Conference on Security and Management (SAM14). 21-24 July 2014, Las Vegas, USA.

1.7.2 Books

As the result of a successful presentation at SAM14, the respectful representative of the Elsevier Publication at the conference initiated a negotiation to transform the outcome of this research to a book (booklet). The task is ongoing and continuous.

1.8 Thesis Outlines

In this thesis, the outline of the remaining chapters is as follows:

- **Chapter 2: Related Work and Background**

  In this chapter, the history of security policies will be examined in detail, including the history of security policy models and its relation to security policy languages.

  Related work and research in the same area as the present research will be reviewed and areas for improvements highlighted.

- **Chapter 3: The Framework Overview and Design Challenges**

  Chapter 3, will outline the high level requirements of the framework. By providing an architectural overview of the framework it will show how different parts of the framework will be tied to each other. A structural overview of the abstract security policy language will be provided in this chapter. Finally, readers will be provided with a road-map that the research has taken to achieve its goals

- **Chapter 4: Security Policy Languages**

  In chapter 4, the necessity for comparison of security policy languages in detail will be discussed. A literature review of comparisons of security policy languages will commence, which is leading the research to the requirements for selecting a small number of security policy languages for the framework.

  Applying those requirements to a list of available security policy languages will truncate the list to a set of three policy languages. The chapter will be finalised by an overview of each language in turn.

  Chapter 5: A New Algebra for Security Policy Languages

  This chapter will describe the usefulness of algebra for security policies with a literature review of algebra made for security policy languages. The chapter will continue by choosing an algebra for the proposed framework and a step-by-step evaluation will be provided to challenge the algebra. Finally, it will provide the way that the algebra can be expanded and proved to be complete.

  Chapter 6: Domain-Specific Language

  The main aim of the chapter is to justify the usage of domain-specific languages in the context of this research. That will be achieved by defining the domain specific language, providing advantages, disadvantages and requirements of domain-specific language in detail. Then the chapter will traverse across different stages of implementing a domain-specific language and map that to the research. The chapter will also show how the research narrowed down a number of programming languages to a chosen language.

Chapter 7: Implementation of the Framework

Chapter 7 will show a step-by-step implementation of the framework. This will be gained by choosing an appropriate software development methodology for the framework, defining the framework requirements, providing a high-level design of the framework and the iterating through the low-level designs of the framework. The chapter will be concluded by evaluation of the framework.

Chapter 8: Summary and Future Work

This chapter will summarise the contribution of the thesis and propose some future research works.

Chapter 2

Related Work and Background

In this chapter, the research presents:

- The history of the security policies,
- The relation between security policy models and security policy languages,
- The related work and research,
- The research contributions of the chapter.

2.1 Security Policy

Generally, the term secure domain applies to an environment that is dependable in the face of two different levels of threats, those being *Internal* and *External* threats. Internal threats refer to those activities that are performed within the secure domain. This level of threats is also referred to as *Human Error* or just *Error* in secure domain definitions. External threats, however, refer to those activities that are imposed from outside the boundaries of the secure domain [116].

Taking the above definition into account, performing certain operations or actions is rigidly restricted within a secure domain in order to protect its addressable resources against internal and external threats. Hence, performing those activities

within the boundaries of a secure domain cannot be authorised unless fulfilment of their pre-requisite conditions is met, approved and validated by the system. The document that describes these conditions and the way that they get validated or approved is called the *Security Policy*. As an example, withdrawal of funds from bank accounts is strictly prohibited unless it has been approved by the account holder. The document that describes how funds can be withdrawn from an individual's bank accounts, which perhaps describes that such an action requires the authorisation and approval of the account holder, is a security policy for that specific financial institute.

To formally define security policies and their possible relation to security policy languages, how a typical enterprise application or environment is secured should be examined. Securing enterprise applications or the network of a multidimensional organisation can be visualised as illustrated in Figure 2.1.

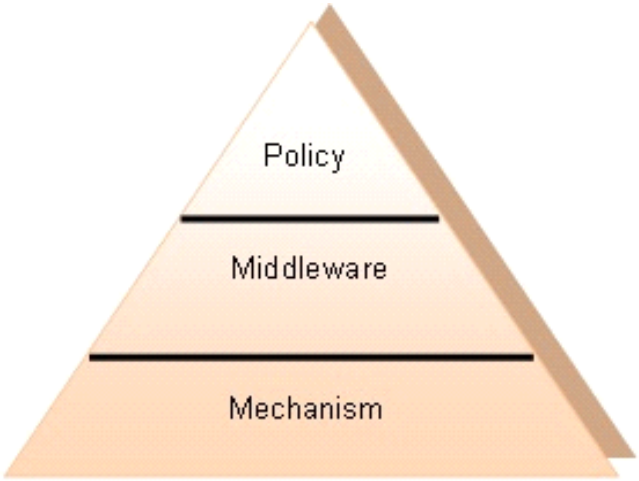


Figure 2.1: Layer of Protection on an Enterprise Application [40]

The security policy, which is located at the highest level of the diagram, usually is a set of high-level documents (i.e. they do not pay attention to details of process) which precisely states what rules should be in place in order to achieve reliable protection. Usually, *Security Officers* are in charge of documenting security poli-

cies, which are driven by their understanding of threats, risks and the sensitivity of the system they are to protect. Security officers are not required to know the exact details of the content that they protect in the environment. For example, a security officer may define how certain confidential documents can be viewed, without knowing the content of those documents.

Mechanism is located at the lowest level. The main responsibility of the mechanism is to deploy security policies across the system. Usually mechanism comprises of computer hardware and software such as, the security management platform, cryptographic primitives, etc.

The security policy and mechanism are connected through the *Middleware*, which composes of business tasks that need to be performed in compliance with the given high-level security policy that has been defined by the security officers. Application developers are typically in charge of middleware [116].

The security policy is a set of high-level documents that states precisely what goals the protection mechanisms are to achieve. It is driven by the understanding of threats and in turn drives the present system design. Typical request access statements in a policy describe which subjects (e.g. users or processes) may access what objects (e.g. files or peripheral devices) under what circumstances. A security policy may be part of a system specification and like the specification its primary function is to communicate [40].

Security policies have a great deal of similarity to the business requirements of a computerised system in describing how the different parts of an application are to tie, interact or communicate with each other. In that regard, like computerised system that are produced or developed using programming languages, security policies also need to be coded in order to carry out their functions. Security policy languages with different levels of functionalities and expressiveness are used to code different security related scenarios.

In accordance with the above definitions, in a very high-level view and in the context of this research, the interactions between security policy language and the

policy server, which are located at the middle and bottom level of the diagram given above can be represented as Figure 2.2. A more detailed view of layers of protection will be presented in the next section.

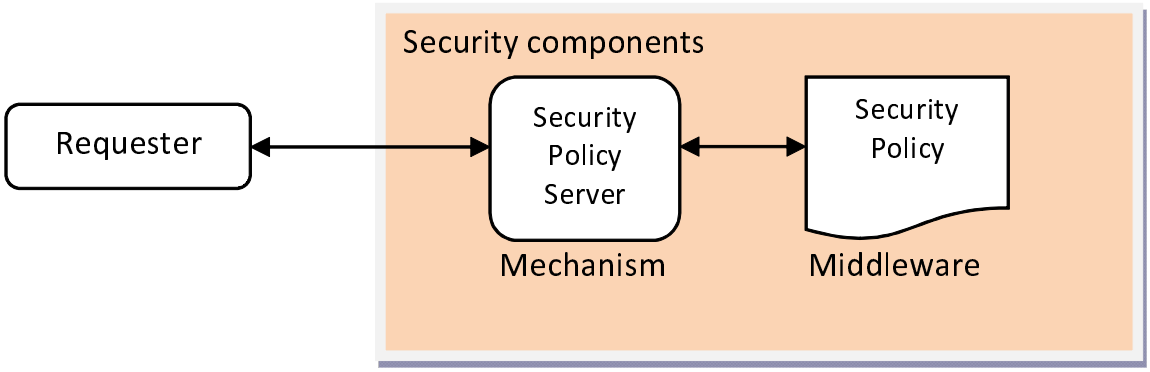


Figure 2.2: High-Level View of Layers of Protection

2.2 Security Policy Models

It would perhaps be beneficial to understand how security policies were introduced. Historically, the first modern network security policy, like many other security aspects, came from a military background. In response to the US Air Force concerns over the confidentiality of data in mainframe systems in 1973 [40], the first modern security policy was introduced. The concern led researchers to devise a simple yet influential model based on restricting information flow between labelled clearance levels such as, *Confidential* and *Top Secret*. This model of security, which was later called the *Bell-LaPadula* [46] was the forerunner for other models like *Biba* [29].

The second wave of security policy models came from well-established business practices such as, accountancy and law firms. In 1987, the *Clark-Wilson* security policy model [60] was introduced, which was an abstraction of the double entry bookkeeping system that is used in accountancy firms and banks. Two years later in 1989, the *Chinese Wall* model [57] was introduced, which targets concerns over

the conflict of interest in business practices (e.g. accountancy and law firm) with different partners serving different customers who are competing in the same fields [116].

Recent attention on the World Wide Web has necessitated not one, but a series of new security policy models to be introduced to address different levels of concern. This can be categorised as the third wave of security policy models addressing the security vulnerability of the associated applications that are used in this medium. In order to respond to market demands, a set of these new models was built from the ground up, enhancing and improving security models that already existed. For instance, *RBAC* was introduced by Ferraiolo and R. Kuhn in 1992 [82]. It is also known as *Access Control List (ACL)* and is widely used nowadays, based on a model that was introduced in 1972 by Graham-Denning [95]. The model was later improved by Harrison, Russo and Ullman in 1976 for operating system protection [97].

2.3 Security Policy Languages

To present a concrete policy especially suited for its automated enforcement, a language representation is needed. As has been mentioned above, security policy languages require an environment in which to function. In addition to security policy languages, there are other components that need to closely interact and co-operate with each other in order to ensure the security of a domain. Often, the combination of all these components that interact and co-operate together is called the *Security Policy Server*. There exists several security policy languages that are closely coupled with the security mechanisms that enforce security policies in a domain.

Perhaps visualisation of these components of security policy servers would be a good starting point. However, as it has already been noted, a variety of security policy languages (and security policy servers) exist, each of which comes with different levels of functionalities, modules, components and indeed their own methods of policy enforcement. As a result, it is almost impossible to provide a unique archi-

tectural model of a security policy server that precisely illustrates the way that their components interact with their environment. Having said that, Figure 2.3 has been presented for illustration purposes in order to have a common vocabulary throughout this document.

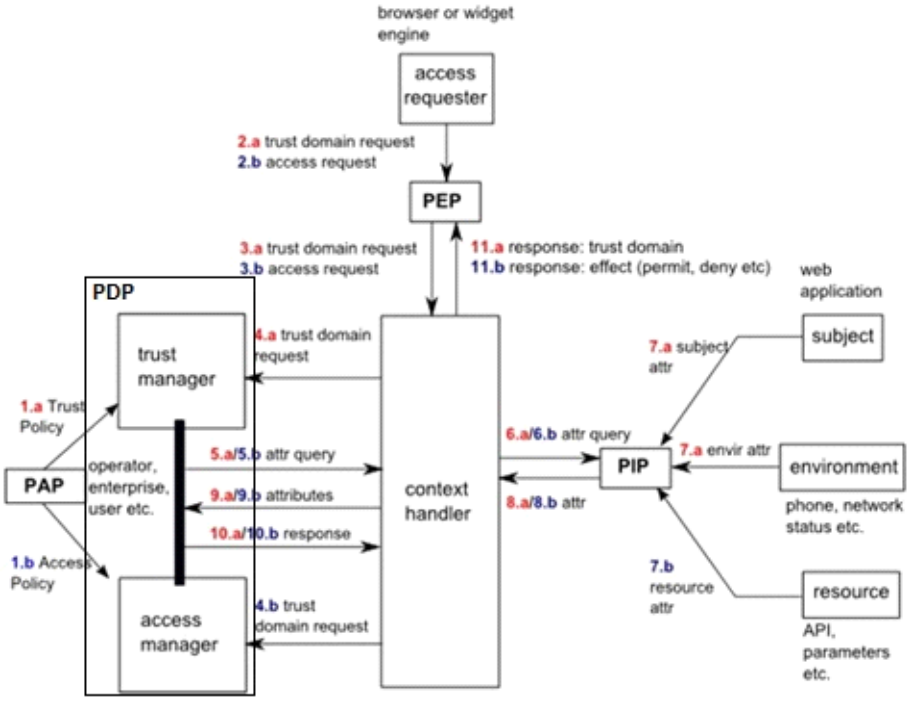


Figure 2.3: An Architectural Model of Security Policy Server [23]

In the picture provided above, the specified functional components are as follows:

- **Policy Decision Point (PDP)** is a system entity that evaluates applicable policy and provides an authorisation decision. This term corresponds to Access Decision Function (ADF) in ISO10181-3 [104]. For trust domain requests, PDP evaluates which trust domain should be assigned to. For access requests, PDP evaluates whether or not a device can be accessed by requester based on the current policy.
- **Policy Enforcement Point (PEP)** is a system entity that performs access

control by making decision requests and enforcing authorisation decisions. This term corresponds to Access Enforcement Function (AEF) in ISO10181-3 [104]. For trust domain requests, PEP assigns a trust domain to a particular web application; for access requests, PEP allows or prevents access to device [23].

- **Policy Information Point (PIP)** is the system entity that acts as a source of attribute values. PIP gathers information that is used by the PDP to evaluate a trust domain or an access control request. For trust domain requests, it collects the subject attributes (for example, how the web application was identified and its associated security attributes), whereas, for access requests, it collects resource attributes (i.e. which device is being requested and using which parameters) and environment attributes (i.e. status of the device) [23].

- **Policy Administration Point (PAP)** is the authority that defines the policy. It could be a network operator, a terminal manufacturer, a web runtime developer, an enterprise or a user at runtime. Policies can be provided by the PAP in different ways, for instance using a pre-loaded file or data structure or a remote management mechanism. The research interoperability framework will be connected to PAP.

- **Subject** is an actor that requires access to resources. Examples of subjects are: websites and widgets. Subject is an entity that may attempt security-relevant actions and corresponds to a single identity.

- **Subject Attribute**, Every subject is associated with a set of attributes. Subject attributes allow the identification of the requester that is attempting access to resources capabilities. The identified requester is then assigned a trust domain according to the appropriate trust policy (for trust domain requests of course). Subject attributes include specific attributes that represent the identity of the requester attempting access to a resource. As an example, validity of a requester can be examined by the URI for the requester e.g. widgets and the URL for websites (in case of web applications) [23].

- **Resource** is the entity(ies) that subjects may request access to. A document, a specific page in a website, a printer, a firewall all these can be presented as resources to the infrastructure.

- **Resource Attribute** Every resource is associated with a set of attributes. Resource attributes include an identifier. Other attributes may be associated with a resource and these can include specific parameters that are specified as a part of the request when attempting access. Resource attributes serve as an input to access control policies [23].

- **Environment** is a set of attributes that are relevant to an authorisation decision and are independent of a particular subject, resource or action.

- **Environment Attributes** are a collection of environment status and/or context attributes that may be relevant to the circumstances of a resource access attempt, but are not directly associated with either the subject or resource. For example, environment attributes can include terminal charging, network connection status, etc. Environment attributes serve as an input of access control policies. Attributes of the environment capture contextual information relating to the device or any other circumstances of the access attempt [23].

Taking these definitions into account, the model operates by the steps as described below:

1. PAP writes policies and policy sets and makes them available to the PDP. These policies or policy sets represent the complete policy for a specified target.

2. The access requester sends a request for access to the PEP.

3. The PEP sends the request for access to the context handler in its native request format optionally including attributes of the subjects, resource, action and environment.

4. The context handler constructs a request context and sends it to the PDP.

5. The PDP requests any additional subject, resource, action and environment attributes from the context handler.

6. The context handler requests the attributes from a PIP.

7. The PIP obtains the requested attributes.

8. The PIP returns the requested attributes to the context handler.

9. Optionally, the context handler includes the resource in the context.

10. The context handler sends the requested attributes and (optionally) the resource to the PDP.

11. The PDP evaluates the policy.

12. The PDP returns the response context (including the authorisation decision) to the context handler.

13. The context handler translates the response context to the native response format of the PEP. The context handler returns the response to the PEP.

14. The PEP fulfils the obligations (if any).

15. If access is permitted, then the PEP permits access (or trusts the peer in case of trust negotiation) to the resource; otherwise, it denies the request [23].

It would be beneficial to look at this research contribution and its impact on the above architecture whilst examining the infrastructure of security policy languages. The proposed framework of this research only focuses on the policy administration point. This is the point at which security administrators write the security policies using different languages. The proposed framework connects to the PAP and acts as an abstract layer that encourages security policy language users to use a standard, abstract language instead.

2.4 Related Works

As mentioned in section 1.2, lack of standard for security policy languages has been noted before. Research such as, SecPal [44], aimed to address this issue accordingly. In this section, however, few research that was conducted to provide security users with a framework for security policy languages will be reviewed.

2.4.1 A Framework for Multi-Policy Environment

Kuhn *et al.* presented a fascinating framework for supporting security policies in a multi-policy distributed environment back in 1995 [113]. In their definition, the framework is literally a policy library that facilitates policy implementation by using mechanisms which have been defined for policy separation, policy persistency and policy reusability. The framework also provides a platform for policy reasoning and conflict.

The paper starts by categorising security policies into three different family models, namely the: *Algebraic Family*, the *Lattice Family* and the *Expert System Family*. In their framework, a policy implementation is a formal representation of that model (family) consisting of a high-level semantic description and model implementation code.

The Algebra family represents those security models which are presentable using an algebraic model. The algebraic model itself defines syntax specification of a policy. In addition, the algebraic family provides modules for specification of abstract data type, type independent specification of abstract data-type like array and list and inheritance of complete security policies.

The Lattice family maps the access-control model, which is widely used within the field of computer security. In this family, all of the entities that are governed by policies are attributed. The attributes are used to map the flow of information security throughout the system. The paper claims that every access-control based security policy that utilises attributes can be rewritten in the form of this family.

The last and final family is the Expert family, which effectively maps those policies that are written in terms of rules and facts. Such a model can be useful for the security policies which require rapid changing and evolution that can be directly be applied onto the rules and facts of the model.

Unlike other papers that have been reviewed as part of this research, Kuhn *et al.*'s paper provides algebra that supports policy comparisons and synthesis. The paper claims that such algebra would assure the co-existence, reuse and implementation of security policies, in addition to defining inter-policy relationships in multi-policy environments.

The paper then provides more information about the algebra, its operators and the way that the algebra presents a policy. It also briefly shows how a security policy can be mapped to a model and how a *custodian paradigm* will be used for implementation [98]. Finally, the paper provides a real world policy example that is using a *Chinese Wall* model and shows how the framework can be used to model and implement such a policy.

As mentioned, Kuhn *et al.*'s paper was written about 20 years ago, and yet it covers the majority of concepts that are utilised in the most up-to-date security policy models and security policy languages. So from that point of view, this is a fantastic piece of work that undoubtedly helped a wide range of researchers to continue their research on security policies into the present day. In fact, this paper was the basis for the use of an algebra in the present research in order to introduce even a more acceptable and presentable framework for security policy languages.

2.4.2 Representing OWL Based Security Policies

Unlike the standard for security policy languages, the lack of an interoperability framework for security policy languages has received limited attention previously. Clemente *et al.* presented a solution for this business requirement in [61]. In order to explore Clemente's proposed solution in more detail, it is necessary to be more familiar with the following concepts:

- **Ontology Web Language (OWL)** is a set of markup languages designed to be used by applications that are required to process the content of information in addition to presenting information to humans. OWL has more facilities for expressing meaning and semantics than XML, thus OWL goes beyond similar markup languages in its ability to represent machine interpretable content on the Web [38].

- **Common Information Model (CIM)** provides a common definition of management information for systems, networks, applications and services and allows for vendor extensions. CIM's common definitions enable vendors to exchange semantically rich management information between systems throughout a network. CIM is a standard proposed by the Distributed Management Task Force [6].

The Clemente *et al.*'s paper for the proposed solution starts by defining the requirement for a framework for security policy languages and then describes the advantages of semantic policy languages. In addition, the paper continues by selecting a limited number of policy languages namely KAoS [148], Rei [107] and SWRL [103] and then proposes the solution by utilising CIM.

The Clemente *et al.* paper is also considered as a well-defined and well-presented piece of work. In fact, this paper was the basis for the use of generic platform in the present research in order to reduce the cost and delivery time of the project.

2.4.3 Identified Areas for Improvements

Despite the fact the two papers discussed above have had their influences in current research, but reviewing these papers with current knowledge of security policy models and languages in mind indicated that the papers can be improved in the following areas:

- **Restricted Approaches**: Both papers provide readers with solutions that satisfy their users to a great extent but none of these approaches can be considered as generic approach to be adopted at any given time.

  For instance, Kuhn and co-workers used two simple models (in addition to one abstract model) to prototype security policies. The provided models seem to be adequate to prototype security policies that existed at the time, but the fact that computer science and particularly computer networking has been revolutionised during the last decade implies that the models provided by the paper more likely will not be sufficient to describe complex security policies that are written and in use currently.

  In addition to that, the Clemente *et al.*'s paper came to a conclusion to choose semantic policy languages over non-semantic languages after comparing these two in the paper. The main reasons were - extensibility at runtime and the adaptability of semantic policy languages. The above mentioned advantages of semantic policy languages are undeniable, but a quick research shows that only a small portion of the available policy languages and the frameworks used in industry and research are based on semantic languages like OWL [24]. Despite the fact that there is no limitation on the use of non-semantic security policy languages with CIM, the proposed solution focuses only on semantic security policy languages to operate over the framework. By imposing such a constraint, a large number of policy languages will not be able to use the proposed solution.

  We will describe in the following chapters that the solution provided by this research is not restricted unlike the related works. The frameworks that is presented as a result of this research will be able to *learn*, *evolve* and *expand* when that is necessary over the time. That makes the framework independent of time, technologies involved or even security policy languages used.

- **Unrepresented Evidences**: Although these two papers have been introduced as an abstract of the research/development conducted, both fail to answer the questions that may have been raised by curious readers.

The Clemente *et al.*'s paper for example, does not pay attention to the formalism of security policy languages. Despite the fact that KAoS and Rei both come with strong underlying formalism, the paper fails to demonstrate the possibility of translation of security policy languages using their formal specifications.

On the other hand, the Kuhn *et al.*'s paper provides readers with formalism to a large extent but does not provide any implementation example. The paper refers to other work that describes the custodian paradigm [98]. However, having reviewed the papers, how the framework breaks down a policy and how the custodian paradigm transits that to an usable piece of code is not readily apparent. Even when the authors show how precisely the framework's algebra is applied to a real world example (the Chinese wall) and divide it down to few sub-models, they do not show how the model gets translated into actual code by the custodian paradigm.

The current research, however, starts with a generic algebra that formally describes security policy languages. Then it presents the readers with a step-by-step development process of the framework. Such an approach **A)** will satisfy curious readers that conversion of security policy languages is an achievable goal (by means of algebra and security policy languages formalism) and **B)** present readers with a detailed guidance that can be used for further development and researches.

- **Security Concerns**: It is undeniable that security is a vulnerable subject. Every security aspects of a system (in the current research the framework) must be known and challenged against possible threats. Having the above fact in mind it seems that these two papers do not pay attention to this subject.

  The solution that is provided by the Clemente *et al.*'s paper uses CIM as the medium between policy languages. It then describes how to map and convert CIM to OWL in order to convert security policies between different languages. Considering that the complexity of CIM to OWL conversion has already been proven in [99] this complexity could lead to instability of frame-

work when it comes to converting complicated and large security policies.

In addition to the above, it must be borne in mind that CIM aims to expand the coverage of information interchange between a wide range of products and vendors through a series of ongoing improvements. Although CIM claims the standard can be used to exchange security information, this definition has not been specifically tailored for this purpose. Hence, a great deal of attention is required when CIM is chosen as an interchange platform for security policy languages in a multi-dimensional organisation. Failure to do so may compromise the security of the system.

- The Kuhn *et al.*'s paper on the other hand shows how the two models defined in the paper can be applied to a single policy (Chinese Wall) that can then be translated by a custodian paradigm (without showing how the implementation works). However, the paper does not show how the two models can and should interact with each other. In fact, the paper claims that when a policy uses more than one model, a decision has to be made as to which model should be used for implementation. Such an approach could have been sufficient for the policies that existed at the time, but as mentioned before, not only might there be more than three defined models in present-day policies, it is also more likely that the implementation must be applied to all of the models as opposed to only one. In addition to that, the interaction between these models must be studied in order to minimise the vulnerability of the system.

  Unlike the presented solutions, this research does not utilise any medium or third party application/product in order to perform the conversion. Instead, the research presents a framework that is implemented from ground up. The research does not rely on the security model conversions that leads to its difficulty as it has been identified in the Kuhn *et al.*'s paper.

- **Real World Example**: The concept of converting security policy written in different languages using a framework is an interesting subject within a multi-domain environment presumably an organisation that is already using different forms and types of security policies. However, none of the documents

above have shown how their framework can be used on and above an existing security infrastructure.

In following chapters, however, the research shows (in a great extent) how a real world example of a security policy written in a generic security policy language can be converted to different security policy languages.

2.5 Summary

2.5.1 Chapter Summary

In this chapter, security policies were examined in more detail, including the history of security policy models, security policy languages and the way that security policy languages have been introduced to the industry. A typical architecture of a security policy language was explored in detail to learn how they react to a request initiated from outside and query access to resources.

In addition to the above, few other related researches conducted in the same area as this research were also reviewed and critically examined from different perspectives. It has also been shown the area that these works can be improved and detailed and how the current research will address these issues in coming chapters.

2.5.2 Research Contributions of the Chapter

Other related research in the same area as this research have been identified and reviewed. Then with the goal of improving these research in mind, they were looked at from different angles. Possible areas in which those papers could be improved were identified. These areas will be addressed throughout this research.

Chapter 3

The Framework Overview and Design Challenges

In this chapter, the research presents :

- The requirements of the framework,
- An architectural overview of the framework,
- An overview of the abstract security policy language used by the framework,
- The current research road-map,
- The research contributions of the chapter.

3.1 The Requirements of the Framework

As a platform for security policy languages, the main purpose of the interoperability framework is to enable security management of a distributed secure domain consisting of few sub-domains and potentially governed by heterogeneous security policy languages from an abstract level. The framework was designed to be a system to meet the complex requirements of real world multi-dimensional organisations. Having said that due to the non-existence of such a framework, its requirements

could not be captured like other software applications by analysing the documents or interviewing the actual users of the system. As a result, in addition to experts refereed to at section 1.2.4 a set of 10 individuals within the industry were selected to participate in a short survey. Their titles and justification for their selection listed as follows:

- **Solution Architects**: Solution architects who design and architect software applications have been chosen to provide their views on how different parts of the framework should be selected and how these parts should communicate with each other.
- **Security Architects**: Eventually the framework will work in a secured and protected environment, therefore, security architects have been selected to improve and enhance the framework's blueprint from a security point of view.
- **Senior Developers**: System developer have been chosen to provide their feedback on low-level designs of the framework. Although logically they should have been invited to participate when the actual development of the system started, however, decision was made to capture their valuable feedback at requirement gathering phase with the aim of having even smoother development phase.
- **Security Administrators**: Security administrators are essentially actual users of the framework. Their expectation of the system helped to capture even more details on system requirements.

In order to obtain the above mentioned set of experts' point of view on the framework, a series of informal interviews was arranged. In addition to that, a small, yet effective questionnaire was distributed amongst them to capture their feedback in more formal way. A copy of this survey has been provided in Appendix D.

Gathered responses in addition to informal interviews filtered and normalised in a way that each of these requirements abstracted to a high-level requirement. For instance if one of the experts asked for the code-assist to be provided as part

of the framework, such requirement categorised under *supportability* requirement. In addition to that each provided sentence within the survey weighted in order to translate the survey to system requirements more accurately. Summarising the interviews and surveys would result the Non-Functional Requirements (NFR) of the system as follows:

- **Simplicity**: This specific requirement is more related to Abstract Security Policy Language (ASPL) of the framework. The ASPL should contain a number of small, orthogonal and well-represented constructs from which complex scenarios can be coded. The simplicity should not compromise the expressiveness of the ASPL. The policies written in ASPL should be simple yet comprehensible.

- **Scalability**: The framework as a platform for security policy language should accept current and future security policy languages within the industry. The framework should not be valid and useful for a trivial period of time nor limited to number of security policy languages.

- **Flexibility**: Within the context of this research, scalability and flexibility are closely coupled. While scalability that ensures future security policy languages would be able to use the framework (i.e. new generators for the framework), flexibility ensures that ASPL is capable of expressing new features that are possibly offered by the future security policy languages.

- **Integrability**: Concerns about the adaptability of the framework with current and future projects. Taking the fact into account that the prime user of the framework are multi-dimensional organisations, in simple words integrability ensures that the framework can be easily coupled to current and future security infrastructures.

- **Supportability**: It is feasible to provide user support via tools for typical model and program management, such as creating, deleting, editing, debugging and transforming policies.

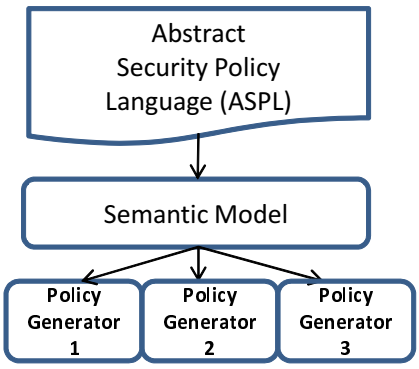


Figure 3.1: Overview of the Framework

Based on the captured information, the blueprint of the framework for security policy languages was drawn. Figure 3.1 represents high level architecture of the framework. In this figure:

- **ASPL**: ASPL is an abstract human readable language that has been designed to allow users to communicate with the framework. The grammar and syntax of the ASPL represents the syntax and grammar of all the security policy languages that the framework supports.
- **Semantic Model**: The semantic model is the core structure of the framework. The structure of the semantic model is independent of security policy languages. It is the perfect abstraction that decouples the input syntax-oriented ASPL from the target security policy languages.
- **Generators**: Valid and parsed scripts will be transformed to language specific scripts using generators. The generator would be responsible to generate language specific scripts based on the populated semantic model. The generated scripts will be sent outside the boundaries of the framework.

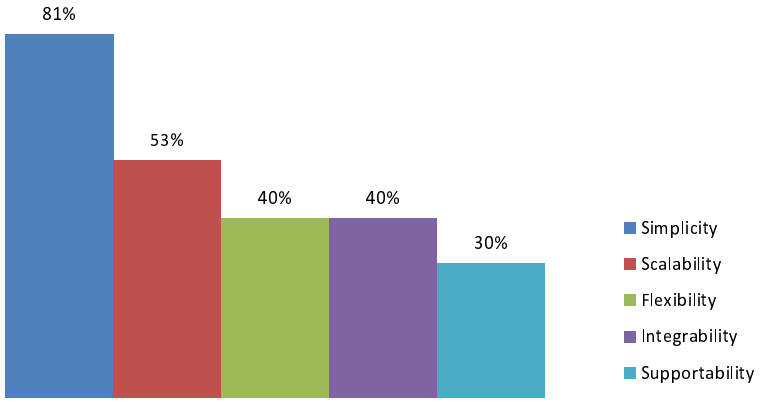


Figure 3.2: Percentage of Requested Requirements (NFR) Asked by Participants

3.2 Architectural Overview of the Framework

Taking the above captured requirement of the framework into account, the architectural overview of the framework is presented in Figure 3.3. The internal components of the framework based on the presented diagram are:

- **Service Requester**: Services provided by the framework have to be invoked by external entity(ies). Service requester represent them in this diagram. Service requester communicates with the framework (possibly) via ASPL script(s). In the first release of the framework, the service invoker could be a system administrator using a Graphical User Interface (GUI) to code the ASPL commands. Enhancement of the framework in the future implies that the service invoker can also be extended to other type of invokers. Remote service invocation via XML (e.g. SOAP) can be presented as an example.

- **Aggregation Layer**: Represents a layer of interfaces which are responsible to receive ASPL scripts in different formats. The very first implementation of the framework would only have one interface. However, as mentioned earlier, other interfaces will be added to this layer whilst the frameworks features is enhanced. These interfaces would have one characteristics in common: they receive ASPL scripts in different formats.

- **Parser / Internal Components**: These represent the core components of the framework. As it has been decided, service invokers are using ASPL scripts to communicate with the framework, therefore, a form of parser is needed to parse the ASPL script based on an agreed and pre-defined grammar and generate a semantic model accordingly. In addition to parser, more other architectural components are needed. Configuration artifacts, ASPL grammar rules and error handling framework are prime examples.

- **Policy Generators**: Policy generators are responsible to generate specific security language policy scripts. These units obtain the semantic model that is populated by the internal components of the framework. Then by traversing the semantic model, the generators produce language specific scripts. These scripts effectively are true representation of the ASPL script received by the framework, in a specific security policy language.

- **Interface Layer**: In order to modularise and decouple the framework's internal components from (policy) generators, interface layer has been utilised. The layer makes the semantic model that represents the ASPL, independent of generators.

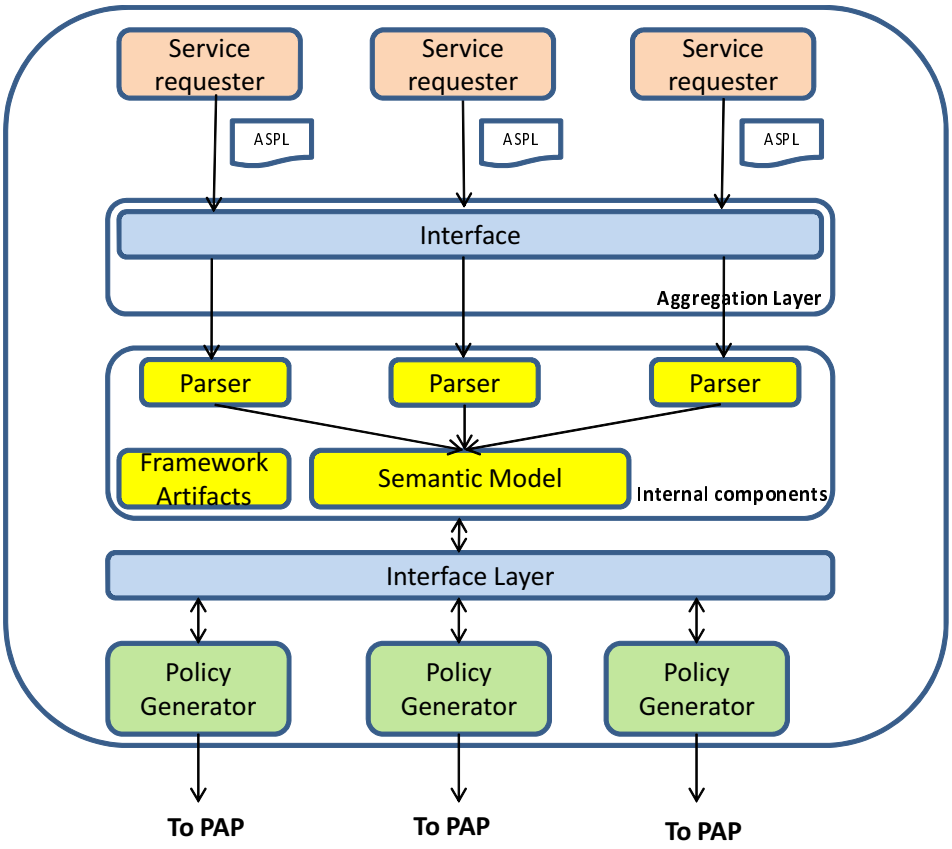


Figure 3.3: Architectural Overview of the Framework

3.3 Overview of the ASPL

As it appears from the name, the ASPL is an abstract language that comes with its own language constants, operators and predicates. Unlike designing a security policy language where the predicates, constants, special keywords are known from the outset, ASPL is introduced with minimum number of features. The reason is unlike other security policy languages, ASPL is a dynamic language. In simple words, it is not a security policy language, it is a collection of all of them. It starts with the minimum features it requires. As the framework grows, it accepts more security policy languages. That in turn results ASPL to be mature and *learns* new features.

Taking the above definition into account, the most simple security policy written in ASPL should support the common scenario of security policy languages using the minimal set of operators and keywords needed to describe that scenario:

```
Authorise [Protect]
       Target "myTarget"
   for executing
       Actions "action1"
   on [from]
       Subjects "mySubject"
```

Listing 3.1: A Policy Written in ASPL

Listing 3.1 presents the simplest yet most common policy (or rule) with positive authorisation policy. Positive authorisation policies define what activities (action) a user (subject) can perform on the set of objects (target) in the domain. In the same sense, negative authorisation policies deny defined activities performed by certain subjects on associated targets.

The above presented listing presents such a simple security policy that does not even contain conditions. In order to slightly enhance the ASPL, it has been decided that *Conditions* be added to the ASPL. In addition to that, most modern policy languages with the exception of few of them (that includes Cassandra [45]), support *Obligation Policies*. Obligation policies specify the actions that must be performed when certain events occur. The assumption here is in a secured domain, obligation policies activities are already authorised to be performed. Therefore, obligations can be merged to authorisation policies as detailed in listing 3.2.

```
Authorise [Protect]
        Target "myTarget"
    for executing
        Actions "action1"
    on [from]
        Subjects "mySubject"
    under following
        Subject Conditions "mySubjectCondition1"
    when
        Obligation Constraints "my Obligation Constraints"
        met
    do
        Obligation Action "myObligationAction"
```

Listing 3.2: Enhanced Policy Written in ASPL

3.4 The Next Steps

Two major steps have to be taken before the current research considered as concluded, namely:

A) **Theoretical Study**

In essence, the policy framework as illustrated above is responsible for providing a platform for policy languages in order to make them interoperable, which would undoubtedly be the main contribution of this research. However, before the system is designed in detail and as a pre-requisite to the present framework's design, it should be proved theoretically that such a transformation is possible. The steps that must be considered on route to this goal can be summarised as:

1. **Policy Language Candidates**

   The framework including its ASPL do not support any security policy language by default. Security policy languages must be introduced to the framework gradually and in a controlled manner. On the other hand, a wide range of security policy languages are available to choose from [24], which makes it almost impossible to cover them all in a single project. Therefore, a handful number of security policy languages must be selected instead. As a result, the ways that security policy languages can be classified into different categories must be identified. A policy language candidate must then be selected from each individual group which represents the characteristics of that specific set. The result of this step will be a set of security policy language candidates that will be input into the next step. These activities will be described and performed in detail in Chapter 4.

2. **Algebra Candidates**

   Logically, the most appropriate way to translate many-to-many languages is to translate them to and from an abstract language. ASPL for instance, acts as an abstract language which makes the security policy languages interoperable. Algebra is chosen as the abstract framework, whilst on the subject to theoretically prove transformation of security policy languages is achievable. Therefore, the research must identify and utilise an appropriate algebra for security policy languages.

   The algebra selected should be capable of expressing different scenarios written in possibly heterogeneous policy languages. Hence, a few scenarios should be evaluated against the chosen policy language candidates (step1). Finally, research should identify any existing weakness of the selected algebra and formulate a solution to address that accordingly. Chapter 5 describes the steps that have been taken to choose, evaluate and enhance an algebra.

B) **Experimental Tests**

The framework has to be tested by real users and their interaction with the system has to be observed. In order to achieve the goal, the following steps have to be taken respectively:

1. **Framework Design** The framework has to be designed in accordance with the software development best practices governed by an appropriate software development methodology. Every decision taken during this step has to be justified and mapped to the framework's requirements or software development best practice. Chapter 6 describes design of the framework in detail.
2. **Framework Implementation and Evaluation** Prototyping of the framework is considered as the next step. Implementation of the prototype should also be governed by an appropriate software development methodology. The result of implementation phase that represents the framework with limited functions and features will be tested by real users. Their feedback will be captured and used to enhanced the framework accordingly.

The road-map of the research is illustrated in Figure 3.4.

3.5 Summary

3.5.1 Chapter Summary

In this chapter, the requirements of the framework surveyed and analysed. By analysing the requirements of the framework the blueprint of the framework was drawn. Then different components of the proposed architecture is explored in details.

In addition to the proposed design, the abstract language of the framework, ASPL, have been introduced. Few simple policies that are supported by ASPL are presented and explained in detail. Finally, road-map of the research, including steps that will be taken at each stage is presented and explained.

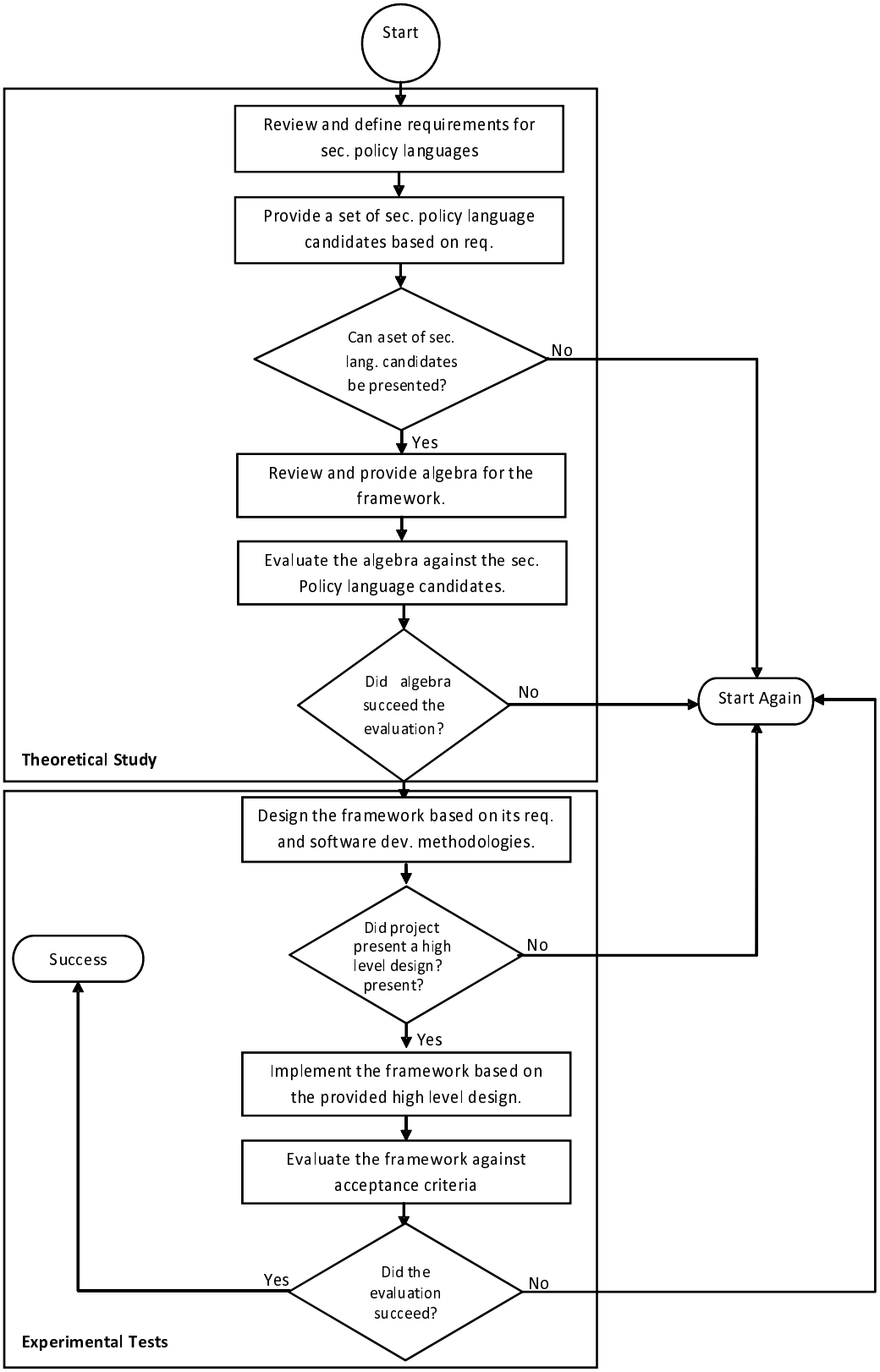


Figure 3.4: The Road-map of this Research

3.5.2 Research Contributions of the Chapter

Due to the non-existence of an interoperability framework for security policy languages, gathering up the requirements for such a framework is considered as a challenging task. In this chapter, it has been shown how in such circumstances these information can be captured by utilising surveys and interviews. Also, it has been detailed how different level of expertise provided by different set of specialists in different areas can be used to obtain the framework's requirements.

In addition, the architectural view of the framework which is considered as the main contribution of the research, proposed and each components of the design is explained in details. One of the main components of the framework is its abstract security policy languages, ASPL, which is explained in this chapter.

As stated before, policy languages come with different levels of formalism, expressiveness and functionality to address different business usage and demands. Variation of security policy languages makes it almost impossible to be able to design and develop a framework that satisfies all security policy languages. Thus, the first step in the present approach to theoretically prove the translation of security policy languages would be to select a subset of available security policy languages. Indeed, the best and most logical way to categorise and select this candidate subset would be to compare these languages in detail. In the next chapter, the way that was considered for comparing security policy languages and how security policy language candidates were selected for this research will be described.

Chapter 4

Security Policy Languages

In this chapter, the research presents:

- The necessity of comparison of security policy languages,
- A literature review of comparisons made of security policy languages,
- Requirements for choosing security policy languages,
- The overview of selected security policy languages for the research,
- The research contributions of the chapter.

4.1 Comparison of Security Policy Languages

Due to the availability of broad range of security policy languages, somehow we need to narrow them down to a specific set and continue our research against those security policy language candidates. No doubt the most logical way to be able to choose an appropriate set of security policy language is to compare them in detail.

Although variation of these security policy languages in terms of formalism, expressiveness, functionality etc., can be considered as their advantages, the boundaries between these languages are neither black and white nor crystal clear. Hence,

security policy languages often overlap with each other. Usually, newly introduced policy languages are intended to address issues that were identified previously, in addition to expanding their comprehensive coverage within the security arena. This makes choosing the right and correct security policy language for a particular project an even more challenging task.

As a result, policy languages have occasionally been compared from time to time by researchers. The main goal of these comparison reports is to typically compare the abilities of different policy languages for the work of other research. These comparison reports of security policy languages often help network architects to customise their security infrastructures and choose the correct security policy language.

What really must be borne in mind is that these comparisons have often focused on a small number of policy languages in order to be able to provide their readers with a precise and clear answer. That, in turn, has produced a number of security policy languages comparisons over the last decade or so. Knowing that the IT industry is continuously evolving at a rapid pace, these reports are usually outdated quickly as time goes by. From the number of surveys regarding security policy languages reviewed in this research, the following reports, which compared these languages from different perspectives have been utilised.

4.1.1 Requirement-Based Comparison

Seamons *et al.* focused on policy languages with trust negotiation capability in [139]. As it appears from the report's title, *Requirement for Policy Languages for Trust Negotiations*, the paper is dedicated to comparison of security policy languages with trust negotiation capability but before they conduct the comparison, Seamons *et al.* clearly defined the requirements of security policy languages in general terms. This clear definition has been used by other researchers over time and has had an impact on other researches.

In a nutshell, Seamons *et al.* defined the requirements for security policy lan-

guages with trust negotiation capability as follows:

- **Well-Defined Semantics**: That requires the language to be simple and compact with mathematically defined semantics such as: logic programs and relational algebra. In their definition, a well-defined semantic policy language should be able to express security policies independent of any particular implementation feature of that language.
- **Monotonicity**: Monotonicity of a security policy language is implied if two parties in a trust negotiation process succeed in achieving trust, then additional disclosure by either party should not have any impact on the trust negotiation decision.
- **Credential Combinations**: A security policy language with trust negotiation capability must allow policy writers to require submission of combinations of credentials using conjunction and disjunction.
- **Constraints on Attribute Values**: A security policy language should allow policy writers to constrain submitted credentials to have a certain type and restrict the values of any other attribute.
- **Credential Chains**: A policy language must provide enough expressive power to describe and constrain chains of credentials to its users.
- **Transitive Closure**: Policy languages with trust negotiation capability should allow users to define trust to be transitive under certain circumstances.
- **External Functions**: Policy languages should utilise well-defined external functions for operation and comparison (e.g. Date).
- **Authentication**: An authentication requirement means that at runtime, the credential submitter will have to demonstrate knowledge of the private key associated with a public key referred to in the credential exchange process [139].

Following the definition of requirements for security policy languages for trust negotiation, the report compared four policy languages with trust management capability, namely: PSPL [54], TPL [100], X-Sec [49] and KeyNote [51]. The report then challenges each of the above mentioned policy languages against the requirements that it defined and presented.

Uniqueness and Areas for Improvements

Seamons *et al.* presented their research more than a decade ago. Knowing the fact that the computer science is evolving at a rapid pace, most of the languages that the report compared and indeed the comparison result itself are now outdated and cannot be referenced anymore. A more up-to-date set of languages should be compared using the requirements provided if trust negotiation comparison of security policy languages is required.

The uniqueness of this report is that it pioneered the comparison of policy languages by defining the requirement for that comparison first. The paper also assisted other researchers by clearly defining the requirement for trust negotiations. Inspired by the Seamons *et al.* paper, this research will also use a set of tailored requirements for comparing security policy languages that should work on the framework.

4.1.2 Scenario-Based Comparison

Knowing the fact that identity theft and uncontrolled exposure of sensitive information are a growing risk for internet users, Duma *et al.* decided to challenge the usefulness of security policy languages against these risks [76].

The ideology of the report is based on scenarios. To be more precise, in order to compare policy languages, the report defines a set of criteria. Following this, for each individual criterion which emerges from real user needs, the report presents a scenario to evaluate the language. If the scenario can be expressed and encoded in a language, then the language fulfils the corresponding criterion.

The report compares six policy languages, namely: Protune [53], Rei [107], Trust-X [33], KeyNote [51], Ponder [67] and APPEL [65]. The scenarios it uses for evaluation are as follows:

- **Minimal Information Disclosure**: Objects that are least sensitive will be selected first for disclosure.
- **Mutual Exclusiveness**: Control the concurrent release of data objects that might be sensitive together.
- **Type of Classification**: Make sensitive objects known and express hierarchies of sensitive objects, semantic equivalence, relationships and more.
- **Granularity of Objects**: Express the granularity of sensitive objects.
- **Access Control**: Control to whom sensitive objects are released.
- **Sensitive Policies**: Control the release of policies that might themselves be sensitive.
- **Push Control**: Address deadlocks due to sensitive policies that cannot be released.
- **Usage Control**: Control how data should be handled by the receiving party.

At the end, the report provides a table that illustrates what each security policy language is capable of with regards to expressing and coding each specific scenarios. The result is presented in Table 3.1.

Uniqueness and Areas for Improvements

Very similar to Seamons *et al.*, Duma *et al.* pioneered comparison of security policies languages by providing a set of scenario as oppose to characteristics of security policy languages. This point of view on the comparison of policy languages makes this report unique in its category. In fact this research also inspired by the Duma *et al.* paper.

Criterion	Protune	Rei	Trust-X	KeyNote	Ponder	APPEL
Type of classification	Ontology	Ontology	Ontology	None	Taxonomy of domain expressions	P3P taxonomy
Granularity of objects	Provided by ontology	Provided by ontology	Provided by ontology	Provided by application	Provided by taxonomy	Provided by taxonomy
Access Control	Yes	Yes	Yes	Yes	Yes	Yes, restricted by P3P vocabulary.
Minimal information disclosure	Yes	No	Yes, partial	No	Yes	No
Mutual exclusiveness	Yes	Yes	Yes, with exceptions	No	Yes, externally	No
Protect sensitive policies	Yes	Yes	Yes, with exceptions	No	Yes	No
Push contol	Yes	Yes	No	Yes	Yes	No
Usage control	No	Yes	No	No	Yes	Yes, restricted by P3P vocabulary.

Table 4.1: Comparison of Security Policy Languages [76]

Having said that it has been noted that one key scenario that this report fails to evaluate is the capability of negotiation with the third parties. Negotiation is provided by almost all modern security policy languages and should have been included in paper's criteria. It will be described in the next chapter that one the areas that this research is going to cover, is negotiation.

4.1.3 Criteria-Based Comparison

De Coi *et al.* claimed that with their detailed and board comparison, users of security policy languages would be able to choose a correct language for their needs [62] . To achieve this goal they rigorously analysed and compared twelve policy languages in details over three years, with the report being concluded in 2008. The first criterion that they considered was that the language selected must be popular and widely used within the industry. Then they described the strongest point of each individual policy languages and the reason why it became popular amongst secu-

rity policy language users. The languages that they reviewed were: Cassandra [45], EPAL [42], KAoS [148], PeerTrust [92], Ponder [67], Protune [53], PSPL [54], Rei [107], RT [115], TPL [100], WSPL [32] and XACML [119]. De Coi *et al.* then evaluated these languages by comparing two sets of criteria, namely: core policy properties and contextual properties.

Basically, the core policy properties that have been inherited from Seamons *et al.* report [139] are used to challenge a security policy language theoretically. Core policy properties are themselves divided into four different sub-properties, the first two of which are exactly borrowed from Seamons *et al.* (as stated above). The full list of these sub-properties can be presented as follows:

- **Well-Defined Semantic**: The same definition as presented in Section 3.1.1, is used here.
- **Monotonicity**:The same definition as presented in Section 3.1.1, is used here.
- **Condition Expressiveness**: A policy language must allow specification of under which conditions the request of the user should be accomplished.
- **Underlying Formalism**: A good security policy language should be based upon a well-known underlying formalism. Knowledge about the formalism of a language would help users to understand some basic features of the language itself.

The second set of properties called contextual properties challenges the practicality of each language in details. Contextual properties are divided into 8 properties defined below:

- **Action Execution**: This criterion evaluates whether a language allows the policy writer to specify actions within a policy.
- **Delegation**: If a language allows policy writers to temporary delegate its rights (mainly access rights) to others, then that security policy language fulfils this criterion. This criterion also challenges the chain of delegation.

- **Type of Evaluation**: Languages that do not support negotiation only support local evaluation, whereas, languages with negotiation capability support distributed evaluation (supposedly each party involved in the negotiation would be equally entitled to break the negotiation). This property determines the type of evaluation each security policy language support.

- **Evidences**: Credentials electronically signed by third parties are referred to as evidences in this report. Determining whether a policy language is capable of providing evidences is the main goal of this property.

- **Negotiation Support**: The name of the property speaks for itself. Whether a security policy language supports negotiation is evaluated by this criterion.

- **Policy Engine Decision**: Although logically the decision in response to a request can be a simple boolean object, some policy languages are capable of providing more information to the requester, in the event that their requests are denied by the policy. The ability to provide such an information to the requester would be determined by this property.

- **Extensibility**: Extensibility is another property that speaks for itself, yet not practically easy to determine. Almost all languages compared provide their users with a level of extensibility.

The uniqueness of this comparison is that it has independently compared a wide range of policy languages with different levels of expressiveness and provides a clear view of the abilities of these languages in one go. The result of this comparison has been provided in Appendix B.

4.1.4 More Comparison of Security Policy Languages

In addition to the above, the following comparison reports have also been studied and reviewed as part of this research. They have been described briefly as follows:

- **A Comparison of Two Privacy Policy Languages: EPAL and XACML** [39]: This report aims to compare two industry standard security policy languages: EPAL and XACML. The report provides an in-depth comparison these two policy languages instead of paying attention to other available security policy languages including those that are widely accepted in the industry.

- **A Survey of Privacy Policy Languages**[112]: This report compared a wide range of security policy languages (or privacy policy languages as it is termed therein). Unlike the previous report, it tries to compare more than a dozen languages but it fails to clearly describe how the authors have arrived at the concluded result. They claimed to have developed a framework for comparison of the languages in question but, the report fails to show any information about the framework.

- **Survey on XML-Based Policy Languages for Open Environments** [34] Compared to the two surveys above, this report can be described as very well presented, organised and provides extensive comparisons. As the name of the report suggests, it only focuses on XML-based policy languages. The report compares X-Author, FASTER, XACL, XACML and SPL, and satisfactorily describes each language. In some cases, it also describes the infrastructure and model of languages in detail, then compares these languages extensively in 22 different categories. Without a doubt, this is a very interesting survey and can be compared with the report from Duma *et al.* Its only shortcoming with respect to the present research was that it only focuses on XML-based security policy languages.

4.2 Requirements for Choosing Security Policy Languages

As noted above, policy languages can be evaluated and classified from different perspectives, for instance, policies with or without negotiation capabilities or poli-

cies with different types of evaluation, i.e. local or distributed. A wide range of comparison reports that look at security policy languages from a different point of view have been examined throughout this present research. Although each of these reports were encouraging, not a single report that can be easily fitted within the context of this research and allow choice of the desired security policy language candidates could be found. As a result, these comparison reports were rigorously reviewed over and over again. Then characteristics of each report were identified in order to be used to generate ideas on how to develop a method of comparison, which has been tailored for this research.

4.2.1 Blending All the Methods Together

Encouraged by the requirement-based comparison i.e. the Seamons *et al.* paper [139], that presented readers with its own requirement for the comparison, it was decided that the present research would have its own requirements for choosing the security policy language candidates. The requirements must be an amalgamation of characteristics of each comparisons reviewed by the research. The requirement that were selected, in addition to the requirements already presented by other reports, are described as follows:

- **Underlying Formalism**: As discussed in chapter 2, the research of Clemente *et al.* [61] did not pay attention to the formalism of the languages they focused on. It is intended to improve on that area with this research, hence, the policies that will be chosen will preferably have strong underlying formalism.
- **Well-Defined Implementation**: As an overview of the design suggested (please refer to chapter 3), policy language generators will be developed in a way to directly communicate with the framework. That implies security policy languages with a proper implementation will have a better chance to smoothly bind with the framework and even make evaluation much easier.
- **Rich Documentation**: Unfortunately, not all security policy languages come with rich and extensive documentation. Understanding the way that policy

languages candidates work is vital to this project, hence, they should provide users with good amount and quality of proper documentation.

- **Programming Language Friendly**: Being more comfortable with Java-based policy languages and taking into account that communication with these languages at low-level will at some point be necessary, then policy languages that are considered Java-friendly security policy languages will have priority.
- **Widely Used and Accepted**:The proposed framework will hopefully be the forerunner for the use of a standard and unique security policy language by security policy language users. If those security policy languages that are widely used by different levels of users are targeted, the chance to promote this framework will be better.

By applying the above defined requirements, a set of policy languages have been shortlisted. In the next step, the requirement definition was improved by merging the report from Decoi *et al.* [62] with that of Duma *et al.* [76] as follows:

A) From Decoi *et al.* a short list of those policy languages that can cover more scenarios was taken, bearing in mind that less featured policy languages can cover only certain (and probably less complicated) scenarios. That in turn implies if languages with more features are chosen and can operate over the framework, then most probably security policy languages with less features would also be able to use the framework.

B) From Duma *et al.*, security policy languages with different properties were selected to mix and match accordingly. For instance, if one policy language with distributed evaluation type was selected, then one policy language with local type of evaluation would also be chosen. This approach would guarantee the selection of wide range of properties to work over the framework.

The top level classification that was presented by Duma *et al.* was adopted. It is believed that the top level classification of policy languages, as described in their report and which is restated here, perfectly fits within the context of this research.

" *Policy languages are classified in three group of Standard-Oriented, Research-Oriented or In-Between of these two mentioned groups. Standard-Oriented policy languages are well defined and widely shared within the industry. However, they come with restricted/minimal set of features. Research-Oriented policy languages are popular amongst academics. They usually provide advance features to their users and go beyond the boundaries that have been put in place by standard-oriented policy languages. There is also another group of policy languages that are neither sufficiently advanced nor fully compatible with standardisation rules to be considered in either of the above mentioned categories. These languages are grouped in the third category called in between.*" [76]

Taking all of the above requirements and properties into account and by extending the top-level classification of Dumas *et al.*, *XACML* from the Standard-Oriented group, *Ponder* from the In-Between group and *Protune* from the Research-Oriented group were selected for further analysis and classification.

4.3 Overview of the Selected Policy Languages

In this section, the selected policy languages will be briefly examined in order to become more familiar with their characteristics.

4.3.1 XACML

XACML stands for *eXtensible Access Control Markup Language*. The standard defines a declarative access control policy language implemented in XML and a processing model describing how to evaluate access requests according to the rules defined in policies.

As a published standard specification, one of the goals of XACML is to promote common terminology and interoperability between access control implementations by multiple vendors. XACML is primarily an Attribute Based Access Control sys-

tem, where attributes associated with a user or action or resource are inputs into the decision of whether a given user may access a given resource in a particular way. Role Base Access Contorl(RBAC) can also be implemented in XACML as a specialisation of Attribute Based Access Control [22].

Structure of Elements

XACML is structured into 3 levels of elements

- PolicySet,
- Policy,
- Rule.

A **PolicySet** can contain any number of *Policy* elements and PolicySet elements. A *Policy*, in turn, can contain any number of Rule elements.

Attributes & Categories

Policies, policy sets, rules and requests all use subjects, resources, environments and actions.

- A **Subject** element is the entity requesting access. A subject has one or more Attributes.
- The **Resource** element is a data, service or system component. A Resource has one or more Attributes.
- An **Action** element defines the type of access requested to the Resource. Actions also have one or more Attributes.
- An **Environment** element can optionally provide additional information [22].

Targets

XACML provides a target, which is basically a set of simplified conditions for the subject, resource and action that must be met for a PolicySet, Policy or Rule to apply to a given request. Once a Policy or PolicySet is found to apply to a given request, its rules are evaluated to determine the access decision and response.

In addition to being a way to check applicability, Target information also provides a way to index policies, which is useful if there is a need to store many policies and then quickly shift through them to find which ones apply. Note that a Target may also specify that it applies to any request. PolicySet, Policy and Rule can all contain Target elements.

Conditions

Conditions only exist in rules. Conditions are essentially an advanced form of a Target that can use a broader range of functions and more importantly, can be used to compare two or more attributes. With conditions, it is possible to implement segregation of duty checks or relationship-based access control.

Obligations

Within XACML, a concept called obligations can be used. An obligation is a directive from the Policy Decision Point (PDP) to the Policy Enforcement Point (PEP) on what must be carried out before or after an access is approved. If the PEP is unable to comply with the directive, the approved access either may or must not be realised. The augmentation of obligations eliminates a gap between formal requirements and policy enforcement.

4.3.2 Ponder

Ponder was a highly successful policy environment used by many in both industry and academia. Yet its design suffered from some of the same disadvantages as existing policy-based frameworks. Their designs were dependent on centralised infrastructure support such as: LDAP directories and CIM repositories. The deployment model was often based on centralised provisioning and decision-making. Therefore, they did not offer the means for policy execution components to interact with each other, collaborate or federate into larger structures. Policy specification was seen as an off-line activity and policy frameworks did not allow them to interact easily with the managed systems. Consequently, such frameworks were difficult to install, run and experiment with. Additionally, they usually did not scale to smaller devices as is needed in pervasive systems. As a result, a new version of the framework came into life: Ponder2.

Ponder2 comprises of self-contained, stand-alone, general-purpose object management system with message passing between the objects. It incorporates an awareness of events and policies and implements a policy execution framework. It has a high-level configuration and control language called *PonderTalk* and user-extensible managed objects are programmed in Java.

The design and implementation of Ponder2 has been aimed to achieve the following goals:

- **Simplicity**: The design of the system should be as simple as possible and incorporate a few built-in elements also if possible.
- **Extensibility**: It should be possible to dynamically extend the policy environment with new functionalities, to interface with new infrastructure services and to manage new resources.
- **Self-Containment**: The policy environment should not rely on the existence of infrastructure services and should contain everything necessary to apply policies to managed-resources.

- **Ease-of-use**: The environment must facilitate the use of policies in new environments and prototyping of new policy systems for different applications.
- **Interactivity**: It must be possible for managers and developers to simply interact with the policy environment and the managed objects, issue commands to the managed objects and create new policies.
- **Scalability**: The policy environment must be executable on constrained resources such as: PDAs and mobile phones as well as for more traditional distributed systems' management.

Ponder2 can interact with other software and hardware components and is being used in environments ranging from single devices, to personal area networks, ad-hoc networks and distributed systems. Ponder2 is configured and controlled using PonderTalk, a high-level, object orientated language.

Ponder2 implements a Self-Managed Cell (SMC). Management services interact with each other through asynchronous events propagated through a content-based event bus. Policies provide local closed-loop adaptation, managed objects generate events, policies respond and perform management activities on the same set of managed objects. Everything in Ponder2 is a Managed Object. The basic Ponder2 system comprises: Event Types, Policies, Domains and External Managed Objects. It is up to the user to create or reuse Managed Objects for other purposes. [79].

4.3.3 Protune

Protune, has a rich set of unique features that are currently not supported by any other standard systems and languages. Some of these features can be individually found in some systems (which are sometimes research prototypes).

An overview of this language can be presented as follows:

- **Flexibility Without Program Coding**: Protune policies can express a variety of static and dynamic requirements without requiring ad-hoc program code. It supports attribute based access control where attribute values may be extracted from different kinds of evidence at different strength and deployment cost. Protune's software interprets the given policies and activates the right procedures for gathering the required kind of evidence.

  Protune's policies may enforce simple obligations by linking some logical pre-conditions (such as - event logged) to actions (implementing event logging, notifications, etc.) that make the corresponding pre-condition true. Policies can declaratively specify when actions are to be executed and which are the peers in charge of their execution. Actions can also be implemented as shell scripts or Java method calls. Changing the policy does not necessarily imply re-implementing actions. Actions provide also an effective integration method for legacy software and data.

- **User Awareness and Documentation**: Proper policy documentation is essential to raise user trust in a system. In fact, that was one of the requirements chosen for selecting the security policy language candidates. Having said that, handwritten documentation is obviously very expensive, especially as policies evolve along over time and the risk of documentation not being aligned with the currently enforced policy becomes higher and higher during a systems life.

  Protune's framework comprises: Protune-X, a unique second-generation explanation facility that presents policies and explains access control decisions in natural language. Since explanations are automatically derived from the executable policy: (i) costs are reduced, (ii) documentation is always up-to-date, (iii) explanations can be contextualised specifically to the current transaction.

  In addition to the above, Protune comes supplied with a rich level of documentation that describes how to write and use Protune policies in details.

- **Policy Confidentiality**: Documentation needs and co-operative enforcement require policies to be accessible. This should be done with care, as poli-

cies themselves may be confidential. Protune allows policy writers to assign sensitivity levels to policy rules and predicates and restricts policy release appropriately.

- **Access Control and Usability**: Policies are application-domain dependent and so are the predicates in access control conditions. In standard frameworks, the context is a black-box and application-specific terms are not stated in a machine-understandable way. This prevents any support to information exchange during authentication phases: users have to be involved because the server cannot specify its credential requests directly to user agents. In Protune, interoperability is enhanced by the means of a lightweight ontology that can specify in a machine-understandable way what it means to be authenticated to a specific system, what are the accepted credit cards, which resources are public and so on. This enables automated support to access control procedures.This approach may significantly improve a user's navigation experience and harmonise usability requirements with strong and articulated access control requirements.

- **Privacy and Usability**: Before using a service for the first time, a user may wish to inspect the service's certifications. Today, this must necessarily be done manually. Protune supports this process by letting user agents ask servers for certificates and other forms of evidence . The semantic infrastructure for interoperability mentioned above is well suited for automating this task, too. Moreover, Protune enhances user privacy by supporting information release policies on the clients. The most common decisions about releasing sensitive pieces of information (be they credentials, unsigned declarations, or whatever) can be specified once and for all as a policy that the user agent can automatically apply, thereby improving the users' navigation experience without sacrificing privacy.

- **Low Deployment and Maintenance Costs**: Protune has been designed to reduce the cost of deployment in new application domains and subsequent maintenance. By minimising program coding and exploiting knowledge-

based techniques to automate a wide range of operations, the costs related to instantiating Protune's framework in a new domain, and the costs related to writing and maintaining policies are significantly reduced.

4.4 Summary

4.4.1 Chapter Summary

Due to the fact that there are a wide range of security policy languages available and taking into account that each individual security policy language would require its users to go through a learning curve, it was essential just to select a sample set of these security policy languages.

In order to have a better understanding of security policy languages and perhaps defining and/or borrowing a set of requirements for the policy language candidates in this research, it was decided to rigorously compare and review these security policy languages. Hence, a number of existing comparison and literature review reports were examined, and that in turn, led to a choice of an appropriate candidate set of security policy languages. Finally, the three policy languages that were chosen were briefly examined in order to be more familiar with the features that these languages provide.

4.4.2 Research Contributions of the Chapter

Despite the fact that a number of review and comparison reports for security policy languages exist, choosing a suitable report that perfectly fits within the context of this research was not a trivial exercise. In this chapter, different comparison of security policy languages were analysed to great details. Then by cross referencing these comparison reports with each other the uniqueness of each individual report was highlighted.

In addition, encouraged by these reports, research, developments and comparisons, a specific requirement set for security policy language candidates to choose from, was produced. How to define the requirements and how to apply them on the set of security policy languages was shown step-by-step. The requirements set that was developed, was specifically tailored for the purpose of this research and lead the project to narrow down the security policy languages to a set of three languages, namely: Protune, Ponder and XACML.

In the next chapter, algebra for security policy languages in particular will be examined. It will be reviewed from different perspectives and the chosen one will be evaluated against the security policy languages selected in this chapter.

Chapter 5

A New Algebra for Security Policy Languages

In this chapter, the research presents:

- The usefulness of algebra to describe security policies with,
- A literature review of algebra made for security policy languages,
- The chosen algebra for the proposed framework and a step-by-step evaluation of it,
- The way that the algebra can be expanded,
- The algebra completeness proof,
- The research contributions of the chapter.

5.1 Algebra for Security Policy Languages

Although access control or to be more specific, role-based access control as it is known in operating systems such as, UNIX, was introduced some time ago, authorisation frameworks and policy languages have been significantly enhanced during

just the last few years. The phenomenon of the internet and the concept of shared resources have forced security architects to allow different policies to be applied on a single resource. The assumption that all policies would be written in the same language and monolithic rule, seemed to be sufficient initially. However, when a combination of heterogeneous policies became vital, people realised an independent non-trivial combination process, or algebra, for security policy languages had to be introduced. As a result, a number of security policy language algebras have been developed during the last decade.

Before describing the algebra for the present research in detail, in the hope of establishing a common vocabulary, the advantage of using algebra for security policy languages will be reviewed. These can be shortlisted as follows:

- The algebra, which can also be called the composition framework, describes policies independent of their implementation. Such a description can be used to examine the completeness and consistency of policies.
- In addition, formal specification, which also can be called algebra, minimises the misunderstanding and ambiguity of policies when different parties refer to the same policy. Such confusion often leads to major security breaches [116].
- Also, the composition framework can be used for the decentralisation of policy descriptions where complicated and sophisticated policies are broken down into smaller manageable and/or heterogeneous policies [151].
- A compositional framework facilitates reuse of policies that are well specified and known to be error-free.
- In the context of this research, the policy algebra, formal specification of policy or composition framework will be used to describe policies on an abstract level.

Considering the advantages of using algebra in conjunction with security policies as stated above, it was almost obvious from the beginning that using algebra

in the present research is inevitable. Accordingly, a number of algebra for security policy language have been reviewed, as listed below.

5.1.1 An Algebra for Composing Access Control Policies

Piero Bonatti is the foremost practitioner of security policy languages. No reputable paper or book in relation of security policy languages can be found without reference to Bonatti's publications.

Bonatti, Vimercati and Samarati came up with the first algebra for modern policy languages in 2002 [52]. Their paper, which is very well organised, begins by describing the characteristics of the composition framework, i.e. the *Algebra*. This definition is used in almost all other papers. It would, therefore, be beneficial to briefly review these characteristics from this paper.

In their definition, the algebra for security policy languages must provide:

- **Support of Heterogeneous Policy Languages**: The algebra as an abstract combination mechanism should be able to express policies defined and enforced by different security policy languages.
- **Support for Unknown Policies**: In their paper, Bonnati *et al.* refer to this property as *Template*. Templates are used to describe policies that are not known at the time a security policy is written. For an example, one can think of a policy that allows access of a user to a certain part of an application based on date and time. These values, i.e. the parts of the application to be accessed, the date and time of the request are not known until run-time. Templates are used to describe such a scenario at an abstract level.
- **Expressiveness**: The algebra should be able to describe and express a wide range of policy combinations independently.
- **Support of Different Abstraction Levels**: The composition language should highlight the different components and their interplay at different levels of

abstraction.

- **Formal Semantics**: The composition language i.e. algebra, should be declarative, implementation independent and based on a solid formal framework.
- **Controlled Interference**: Algebra cannot be simply used to combine policies. The algebra should also be used as a mechanism to detect and prevent conflicts.

Following the definition of algebra characteristics, Bonnati *et al.* provided the preliminary concepts. In their definition, widely accepted and used by following researches, a *Policy* is a set of ground rules (or variable free) triples of *Subject*, *Object* and *Action*. The paper details policy expressions by presenting the algebra's functions and minimal operators that it needs to express policies. These operators, which are described in detail, are: *Addition, Conjunction, Subtraction, Closure, Scoping Restriction, Overriding and Template*. One of the characteristics of this paper, apart from being the first paper that presented algebra for modern security policy languages, is the definition of Template, which presents a partial evaluation of policies. Although, the concept of templates is criticised by subsequent papers, most contemporary algebras introduced afterwards had no choice but to use the concept of Template in their definitions.

The paper then continues by providing a few real word scenarios and tries to express them using the defined algebra. It then proves that the provided algebra as defined and enhanced throughout the paper is complete (with respect to the operations as detailed above). The only criticism applicable to the presented framework is that it explicitly does not support expressing policies with negative authorisation [151][131]. Although they have tried to overcome this limitation by expressing the policy using the subtract operator, this violates the abstraction of policy languages [151].

5.1.2 A Propositional Policy Algebra for Access Control

Shortly after Bonatti presented their algebra for security policy languages, Wijesekera *et al.* published their framework, which was effectively an extension of the former framework [151]. In their definition, policies are relations as opposed to functions and they non-deterministically transform permission sets assigned to the subjects. Permission in their definition is determined by ordered actions allowed on an object (object, $\pm$ actions) and permission sets is a set of permissions. Transforming permission sets to permission sets using policies is the main difference between what they have provided and what Bonatti presented, which transforms permissions to permissions. In their justification, Wijesekera *et al.* believed that using collection of permissions allows authors to model different non-deterministic, incomplete and inconsistent policies.

5.2 An Algebra for Fine-Grained Integration of Security Policies

Having reviewed a few algebras for security policy languages, the conclusion was reached that the algebra introduced by Rao *et al.* [131] could be an appropriate choice for this research by providing the following facts:

- **Enhanced Algebra**: The Algebra that is provided by the Rao *et al.* paper was proposed recently, hence, it addresses issues that have been raised against previous algebra. That also implies the algebra can cover even more security policy languages.
- **Simplicity**: Mathematics and algebra to be specific often are labelled as difficult subjects. Providing simple yet effective algebra will encourage readers and future researchers in this field. Having the above-mentioned fact in mind it has been noted despite extensive operators and semantics that have been provided by the algebra, it is presented in an easily readable manner.

- **Implementation**: Out of all other algebras reviewed by this research, the algebra that is provided by the Rao *et al.* paper is the only one that suggests how the algebra can be implemented by a security policy language thus, the algebra would have a better chance to be extended in order to cover all the security policy languages that were selected in the previous section.

The algebra is first reviewed in detail and then evaluated against the languages that were selected in the previous chapter.

5.2.1 Policy Semantics

Rao *et al.* have presented a simple, yet powerful, algorithm to describe XACML policy languages in [131]. This was later extended by Zhao *et al.* in [155].

In their notation, a that characterises an object, subject or an environment is a finite set of names. In the same sense, a domain defines a set of possible values for a and is denoted by *dom(a)*. Taking these notations into account a *request* is defined as follow:

Definition 1: Let $a_1, a_2, ..., a_k$ be attribute names and let $v_i \in dom(a_i)(1 < i < k)$ then r described as $r \equiv \{(a_1, v_1), (a_2, v_2), , (a_k, v_k)\}$ is an access request.

Using the above definition, an request allowing a PhD student to access digital library without restriction could be described as:

$r \equiv \{$(user, PhD student),(act, access digital library),(time restriction, NO)$\}$.

Assuming system denotes the field where all these entities (e.g. subjects, actions etc.) co-exist in, then the state of system is defined as below:

Definition 2: Let S be the set of subjects, T be the set of targets, E be the set of event triggers and C be the set of conditional constraints. Then, the system state is defined as: $ST = E \times C \times S \times T$. This definition allows a system state to be described as: $st = st(e,c,s,t)$ consisting of an event trigger $e \in E$, the conditional constraint $c \in C$, subject $s \in S$ and target $t \in T$.

In another word, the state of the environment is an entity that captures all the details that require certain action to be executed on a target by a subject. The previous example could be expanded by adding more restrictions to the request as follows:

$r \equiv \{$(user, PhD student),(source, digital library),(time restriction, yes), (access hours, universitys working hours), (download restriction, yes)(download per day, 5 papers)$\}$.

Hence, the state of the environment will be determined not only based on the subject (PhD student) and target (access to digital library) but also by taking conditional constraints (time of the request) and triggers (the total number of downloads).

There are few concepts which have to be introduced before the next definition is presented, namely: *Authorisation Policies* and *Obligation Policies*. Authorisation policies determine whether a specific request is permitted under given conditions and circumstances. Whereas, obligation policies define whether certain action(s) have to be taken place assuming that specific conditions are met and fulfilled. For example, an authorisation policy could determine whether an employee is permitted to use certain part of a company's system, for instance, finance system. However, obligation policies define *IF* an employee is authorised to use the finance system, then certain information has to be captured and stored for further references e.g. time and date that the employee logged-in and perhaps the areas that he/she visited. As it appears from the given example, the obligation policies will be executed when the corresponding authorisation policies are fulfilled.

Both decisions are taken by authorisation policy that is denoted by d_a and decision taken by obligation policies denoted by d_o can only have one distinctive value which will be selected from the set of $\{Y, N, NA\}$. These values present the decision made by a policy that are permitted (Y), denied (N) or not applicable (NA) respectively. Taking these concepts into account, the security policy is defined as:

Definition 3: A security policy is defined as a request evaluation function $P : ST \times A \rightarrow D$, where ST is the set of system states, A represents a finite set

of actions and D denotes the set of decision tuple for authorisation and obligation associated with: $P(D_a, D_o) = \{(Y,Y),(Y,NA),(N,NA),(NA,NA)\}$.

The function P takes a system state $st \in ST$ and an action $a \in A$ as input and returns a decision tuple (d_a, d_o) determining whether a is authorised and obliged to execute in state st. As it is obvious from the decision tuple set provided in the above definition, the authorisation and obligation decisions support three values namely: *Y, N* and *NA*. However, a close examination of decision tuple shows the decision set does not include: (N,Y),(NA,Y). The reason is the obligation policies only satisfy when the corresponding authorisation policies are satisfied in the same state of the system.

5.2.2 Policy Constants

The policy constants that have been defined for algebra can be presented as :

Definition 4: A *Permit Policy* is defined as : $P_+ : ST \times A \rightarrow (Y,NA)$. P_+ permits all requests in at any state of system without considering any obligations.

Definition 5: A *Deny Policy* is defined as $P_- : ST \times A \rightarrow (N,NA)$. P_- denies all requests in any state of system without considering any obligations.

5.2.3 Operators Applied to Policies

In addition, assuming P_1(st, a) and $P_2(st,a)$ are two policies that are going to be combined using algebra operators and assuming P_i denotes the integrated policy. In order to illustrates the effect of integration of different policy operators, a combination matrix will be used. On the combination matrix, the first row and column of the matrix denotes the possible values for each policy with regards to *st* and *a* and the rest of the cells shows the value of integrated policy at different states of the system. Whilst on the subject, it is also assumed that $Y > NA$, $N > NA$, as both Y and N provide more information about a request than NA.

Taking the above definitions and assumptions into account, algebra basic operators can be presented as:

Addition (+): Integrated policy *Pi* would be union of P_1 and P_2. In other words, P_i authorises requests which are permitted by either of policies and denies requests which are denied by both policies. Taking the above definition into account, $P_I(st,a)$ will be denoted as : $P_I(st,a) = P_1(st,a) + P_2(st,a)$. A corresponding combination matrix for the operator has been presented in Table 5.1.

P1 \ P2	{Y,Y}	{Y,NA}	{N,NA}	{NA,NA}
{Y,Y}	{Y,Y}	{Y,Y}	{NA,NA}	{Y,Y}
{Y,NA}	{Y,Y}	{Y,NA}	{NA,NA}	{Y,NA}
{N,NA}	{NA,NA}	{NA,NA}	{N,NA}	{N,NA}
{NA,NA}	{Y,Y}	{Y,NA}	{N,NA}	{NA,NA}

Table 5.1: Policy Combination Matrix for Addition Operator (+)

The above table shows how operator (+) takes authorisation and obligation values and provides with the decision. Each row and column of the table denotes the effect of P_1 and P_2 respectively with respect to the request *r*.

Intersection (&): GivenP_1 and P_2, P_I is defined as the intersection of these two policies, if P_I returns the same decision that is agreed by two policies. More precisely : $P_I(st,a) = P_1(st,a) \& P_2(st,a)$.

P1 \ P2	{Y,Y}	{Y,NA}	{N,NA}	{NA,NA}
{Y,Y}	{Y,Y}	{Y,NA}	{NA,NA}	{NA,NA}
{Y,NA}	{Y,NA}	{Y,NA}	{NA,NA}	{NA,NA}
{N,NA}	{NA,NA}	{NA,NA}	{N,NA}	{NA,NA}
{NA,NA}	{NA,NA}	{NA,NA}	{NA,NA}	{NA,NA}

Table 5.2: Policy Combination Matrix for Intersection Operator (&)

Negation ($\neg a$, $\neg o$): $\neg P_1(st,a)$, returns P_I which effectively denies and/or permits every requests P_1 permits and/or denies. Operator $\neg a$ negates the result of the evaluation of an authorisation request. It does not change the obligation evaluation result. Having said that, negation of $\neg aP_1(st,a) = \{Y,Y\}$ will give the result $P_1(st,a) = \{N,NA\}$ .The reason is that as per *definition 3*, the decision set of $\{N,Y\}$ is not permitted. Equally $\neg o$ negates evaluation result of obligation policies. $\neg o$ does not change authorisation policies in any case. $P_1(st,a) = \neg P_1(st,a)$ where $\neg \in \neg a, \neg o$.

P1(st,a)	{Y,Y}	{Y,NA}	{N,NA}	{NA,NA}
¬aP1(st,a)	{N,NA}	{N,NA}	{Y,NA}	{NA,NA}
¬oP1(st,a)	{Y,NA}	{Y,Y}	{N,NA}	{NA,NA}

Table 5.3: Policy Combination Matrix for Negation Operator ($\neg$)

Subtraction (-): P_I which denotes the result of $P_1(st,a) - P_2(st,a)$ is defined as a policy that allows through all the requests that are authorised and obliged by $P_1(st,a)$ and are not applicable by $P_2(st,a)$. It is not hard to express the subtraction operator only using the operators we have covered so far, namely $\{+, \&, \neg\}$. In other words, $P_I(st,a) = (P_1(st,o) + \neg aP_2(st,a))$ & $(P_1(st,a) + \neg oP_2(st,a))$

P1 \ P2	{Y,Y}	{Y,NA}	{N,NA}	{NA,NA}
{Y,Y}	{NA,NA}	{NA,NA}	{NA,NA}	{Y,Y}
{Y,NA}	{NA,NA}	{NA,NA}	{NA,NA}	{Y,NA}
{N,NA}	{NA,NA}	{NA,NA}	{NA,NA}	{N,NA}
{NA,NA}	{NA,NA}	{NA,NA}	{NA,NA}	{NA,NA}

Table 5.4: Policy Combination Matrix for Subtraction Operator (-)

Projection (Π). Taking into account the fact that the state of an environment is determined by events, constraints, subjects and targets, as described in Definition 2, and assuming c is a computable subset of $(ST \times A)$, the projection operator restricts

the policy P to requests which are satisfied by c. In other words, the projection operator takes parameter, the domain constraints and restricts the policy only to the set of requests identified by domain constraint. If the domain constraint are not satisfied it retains (NA,NA).

$$P_I(st,a) = \Pi^{(d_a,d_o)}_{c(ST\times A)}(P(st,a)) = \begin{cases} \{d_a,d_o\} & \text{if } (st,a) \in c(ST,A) \text{ and } P(st,a) = (d_a,d_o) \\ \{NA,NA\} & \text{o}\textit{therwise} \end{cases}$$

Perhaps reviewing the following scenario will illustrate the effect of the operator in more details. With regards to a computable state of an environment an integrated policy $P_I(st,a)$ can return either of the policy decision mentioned before, to be more precise : $P_I(D_a,D_o) = \{(Y,Y),(Y,NA),(N,NA),(NA,NA)\}$. Now consider a scenario where restriction of the combination of the $P_1(st,a)$ and $P_2(st,a)$ in a way to only return back a specific decision (for example (Y,NA)) in the computable state of the environment is desired. In such circumstances, the projection operator will be used to restrict the policy as described above.

$$P_I(st,a) = \Pi^{(Y,NA)}_{c(ST\times A)}(P_1(st,a)\&(P_2(st,a)) \tag{5.1}$$

Whilst on the subject, it would be useful to note that the projection operator is a classic implementation of *template* that was introduced by Bonatti *et. al* (please refer to section 5.1.1).

5.2.4 Expansion of Algebra

Definitions, operators and policy constants as described above would be able to express the majority of policies combination for a wide range of security policy languages defined within an environment using a set of ground, i.e. variable free, authorisation and obligation policies. However, it is often necessary to go beyond the boundaries of an environment.

One of the security policy language candidates, i.e. Ponder, comes with negotiation capability. Negotiation, which is a characteristic of relatively new security policy languages certainly goes beyond the definition of an environment as each party has no control or visibility of the environment of the other participant in the negotiation. This characteristic of security policy languages cannot easily be fitted into algebra as expressed above. Hence, it is necessary to know more about this distinctive feature and expand the algebra accordingly.

The following example, which is widely shared among security policy languages with negotiation capabilities, goes beyond the concept of unilateral negotiation as it is known in traditional distributed systems [106] [145]. In this scenario, Alice, who is a police officer, would like to apply for a free language course through an online agency. She does not mind providing information as long as it is not categorised as sensitive information.

- **Step 1**: Alice submits a request to the agency for access to the free language course.
- **Step 2**: The agency replies by requesting that Alice shows a police identification number issued by the state police to prove that she is a police officer, and her driving license to prove that she is living in the same province.
- **Step 3**: Alice is willing to disclose her driving license to anyone, so she sends it to the agency. However, she considers her police badge to contain sensitive information. She negotiates with the agency and indicates that in order to provide her police identification number, the agency must prove that it belongs to a certain governing organisation such as, the Better Business Bureau.
- **Step 4**: Fortunately, the agency has a Better Business Bureau membership number. The card contains no sensitive information, so the agency discloses it to Alice.
- **Step 5**: Alice now believes that she is safe to disclose her sensitive information to the agency and so she provides her identification number to the

agency.

- **Step 6**: The agency verifies that the identification number is valid. In addition, it verifies that Alice lives at the same address as stated on her driving license. Accordingly, the agency gives Alice a special discount for this transaction and allows her to sign up free of charge for the course [126].

As is apparent from the above scenario, in simple terms, negotiations can be divided into a series of steps. A message usually gets exchanged in each step whilst the state of the negotiation is partially evaluated. Each evaluation leads the negotiation to the next level. Messages that are exchanged at each step could come from different roots and types. For instance, a message could be a *Query message* like *"Is Alice entitled to a discounted course if she provides her driving license and her identification number?"* Messages that contain credentials are called *Policy Sets*. In the above example, the messages that contain Alices driving license number and her identification number are among these messages. Messages can simply be described as *Decision Messages* that indicate the end of negotiation, possibly with a decision [55].

Expressing Negotiations with Algebra

The algebra for security policy languages as it has been described so far is incapable of expressing the above mentioned scenario, simply because another dimension has been added to the algebra definition, that being *third parties*. In the above example, the agency's relation with the governing organisation (Better Business Bureau, or BBB) is simply beyond the visibility and control of Alice. Thus, the described algebra is not capable of formulating the state of the environment as stated in Definition 2. It is, therefore, necessary to slightly expand the definitions to incorporate the negotiation into it.

Definition 6 (Enhanced version of Definition 2): The state of an environment is defined as a function that accepts ground (variable-free) and non-ground or more appropriately, literal states. Assuming Σ denotes the ground literal states and refers

to the set of literals that are held at the current state of the environment and Ω denotes the set of non-ground literal at the same state, an environment state is defined as: $ST : \Sigma \times \Omega$.

Let S be the set of subjects, T be the set of targets, E be the set of event triggers and C be the set of conditional constraints. Variable free state is defined as: $\Sigma = E \times C \times S \times T$. This definition allows the state function to be described as: $\Sigma = st(e,c,s,t)$ consisting of an event trigger $e \in E$, the conditional constraint $c \in C$, subject $s \in S$ and target $t \in T$.

Ω on the other hand denotes those elements (and/or entities) that are non-deterministic, located outside the boundaries of the environment but that would have a direct impact on the state of environment which, in turn, may have its own impact on the decision made by the policy that is operating at the state.

Taking the above definitions into account definition 3 can now be redefined as follows:

Definition 7 (Enhanced version of Definition 3): A security policy is defined as a request evaluation function $P : \Sigma \times \Omega \times A \rightarrow D$, where Ω denotes the finite set of non-ground literal, Σ is the set of system states, A represents a finite set of actions and D denotes the set of decision tuples for authorisation and obligation associated with: $P\{D_a, D_o\} = \{(Y,Y),(Y,NA),(N,NA),(NA,NA)\}$. The function P takes non-ground literal Ω, an action a and ground state literals Σ (that intuitively specifies which ground literals must be used) as input and returns a decision tuple (d_a, d_o) determining whether the action is authorised and obliged to execute in the state of environment. As is obvious from decision tuple set, it does not include (N,Y),(NA,Y) as the obligation state is satisfied with positive authorisation.

In order to tackle the challenge of providing algebraic expression for policies that involve in a negotiation, Divide and Concur (DandC) algorithm [68] has been used. Divide and Concur is widely used in computer science. DandC works by dividing and breaking down the main problem into two or more sub-problems of the same type, until these become simple enough to be solved directly [64]. Using

DandC, a policy involved in a negotiation will be divided into different stages and instead of dealing with one policy, a series of small and simple policies will be defined for each individual stages of negotiation. Please note each stage of negotiation will be defined and characterised by ground and non-ground literals defined at that stage.

In addition to the above redefinitions, another operator must be introduced that utilises both the sets of ground and non-ground literal states held at any given time. Assuming that Σ denotes the ground literal and refers to the set of literals which are held at the current state of environment (triggers, events, conditional constraints), and Ω denotes the set of non-ground literals (i.e. those literals that are held beyond boundaries of an environment), the *Trace* operator for policy languages with negotiation capability that provides partial evaluation with regards to $\Sigma \times \Omega$ can be defined as follows:

Trace (H) : A Trace for policy P, which is a converging and non-ambiguous process is defined as a set of finite sequence of policies:

$$Pol_0 \overset{\Sigma,\Omega}{\rightarrow} Pol_1 \overset{\Sigma,\Omega}{\rightarrow} \ldots \overset{\Sigma,\Omega}{\rightarrow} Pol_{n-1} \overset{\Sigma,\Omega}{\rightarrow} Pol_n$$

A Trace is complete if for the last element Pol_n in the sequence, there exists no policy Pol_o such that

$$Pol_n \overset{\Sigma,\Omega}{\rightarrow} Pol_o$$

In simple words, the Trace is complete if unchanged criteria of policy P, denoted by $Pol_n, Pol_{n-1}, Pol_{n-2} \ldots$ (with regards to Σ and Ω) results in the policy making the same decision every time.

Theorem 1:

For all policies P

1. In relation to Σ and Ω policy P has no infinite complete traces.
2. All complete traces of policy P (which are defined as finite sequences of policies) with an end policy element of Pol_n with regards to Σ and Ω have the same final element, that is, policy P's decision.

Proof:

In order to prove (1) in Theorem 1 the following must be proven simultaneously:

A) Policy P cannot have a complete trace with infinite end elements and

B) Policy P cannot have infinite complete traces with finite sets of end elements.

and the proof:

A) The term *complete trace* used in the theorem implies that the trace must come to an end that is, $policy_n$ (which denotes the final decision of the Policy P). In contrast, definition 3 clearly introduces a finite set of decision tuples for any Policy P. In other words, Policy P cannot have a set of complete traces with an infinite set of decisions/final elements.

B) Arguably there could be an infinite number of scenarios with a finite number of final elements. However, using the term in relation to Σ and Ω within the theorem narrows down the number of scenarios and distinctly specifies which set of ground and non-ground literals is used.

Taking the above into account, the first part of the theorem proves itself because based on definition 3, policy P cannot have complete traces with infinite end elements $policy_n$ and at the same time utilising the set of Σ and Ω narrows down

the number of scenarios, hence, policy P cannot have infinite complete traces with regards to Σ and Ω.

In order to prove (2) in Theorem 1, again the terms Σ and Ω are used, which implies that the second part of the theorem is referring to a specific scenario. To prove this, one must refer to definition 7, which specifies that the final decision of a policy is determined by three inputs: $\Sigma \times \Omega \times A \rightarrow D$. In other words, the expression can be read as: as long as combination of Σ , Ω and A are met, policy P will make a decision. The way in which the policy collects these inputs (i.e. Σ , Ω and A) has no effect on the decision that is made.

Considering that Trace is a sequence of policies that individually come to a decision and considering the fact that the order of evaluating the sub-policies has no effect on the policy's decision (with regard to Σ and Ω) proves that different complete traces must have the same final element.

To make this part of the proof more tangible, consider the above mentioned scenario in which, Alice asks for a discount on a course. If the ground and non-ground literals of the environment are kept the same, any alteration to the sequence of the events does not change the policys decision. In other words, if Alice asks the agent to disclose their BBB membership number first and then she discloses her police badge number (and perhaps her driving licence), she would still be eligible for a discount.

So taking the newly introduced operator Trace and the new definition of the state of the environment into account the ultimate security policy P which is whether Alice is eligible for a free language course can be broken into a number of security policies, each of which has to be evaluated separately. In effect, steps 2 to 5, can each be described individually as a separate security policy (i.e. operator Trace now is in use). Each security policies can then be evaluated using the state of the environment at that time which brings the non-ground variables into the picture.

5.2.5 Algebra Expressions

In practice, expressing a policy using the proposed algebra requires multiple operators to be used at the same time, so it is necessary to define an algebraic expression as follows. An expression consists of a left associative, an operator and a right associative. Trace has the highest precedence, negation and projection have the same priority, followed by intersection and addition respectively.

Theorem 2: Assuming $P_1 = P_1(st,a)$ and $P_2 = P_2(st,a)$ the algebra expressions can be described as:

- **Community:** $P_1 + P_2 = P_2 + P_1$, $P_1 \& P_2 = P_2 \& P_1$
- **Associatively:** $(P_1 + P_2) + P_3 = P_1 + (P_2 + P_3)$, $(P_1 \& P_2) \& P_3 = P_1 \& (P_2 \& P_3)$
- **Complementary**: $P_+ = \neg a P_-$, $P_- = \neg a P_+$
- **Involution**: $\neg(\neg P1) = P1$
- **Idempotency**: $P_1 + P_1 = P_1$, $P_1 \& P_1 = P_1$
- **Distributivity**: $P_1 \& (P_2 + P_3) = (P_1 \& P_2) + (P_1 \& P_3), P_1 + (P_2 \& P_3) = (P_1 + P_2) \& (P_1 + P_3)$

  $\Pi(P_1 + P_2) = (\Pi P_1) + (\Pi P_2), \Pi(P_1 \& P_2) = (\Pi P_1) \& (\Pi P_2)$
  $\mathrm{H}(P_1 + P_2) = (\mathrm{H} P_1) + (\mathrm{H} P_2), \mathrm{H}(P_1 \& P_2) = (\ \mathrm{H} P_1) \& (\mathrm{H} P_2)$

  $\neg a(P_1 + P_2) = (\neg a P_1) + (\neg a P_2), \neg a(P_1 \& P_2) = (\neg a P_1) \& (\neg a P_2)$

Proof:

Combination matrix is used to prove this theorem. Following proof shows how one of these expressions, combination, was approached. The very same approach can be used to proof other expressions.

Assuming: $P_1 = P_1(st,a)$ and $P_2 = P_2(st,a)$, the effect of $P_1 + P_2$ is seen as follows:

P1 \ P2	{Y,Y}	{Y,NA}	{N,NA}	{NA,NA}
{Y,Y}	{Y,Y}	{Y,Y}	{NA,NA}	{Y,Y}
{Y,NA}	{Y,Y}	{Y,NA}	{NA,NA}	{Y,NA}
{N,NA}	{NA,NA}	{NA,NA}	{N,NA}	{N,NA}
{NA,NA}	{Y,Y}	{Y,NA}	{N,NA}	{NA,NA}

Table 5.5: Policy Combination Matrix for Expression $(P_1 + P_2)$

Now if the operands are changed to read $P_2 + P_1$ the combination matrix will be:

P2 \ P1	{Y,Y}	{Y,NA}	{N,NA}	{NA,NA}
{Y,Y}	{Y,Y}	{Y,Y}	{NA,NA}	{Y,Y}
{Y,NA}	{Y,Y}	{Y,NA}	{NA,NA}	{Y,NA}
{N,NA}	{NA,NA}	{NA,NA}	{N,NA}	{N,NA}
{NA,NA}	{Y,Y}	{Y,NA}	{N,NA}	{NA,NA}

Table 5.6: Policy Combination Matrix for Expression $(P_2 + P_1)$

Comparing the above two combination matrices simply leads to the conclusion that the algebraic expression $P_1 + P_2 \equiv P_2 + P_1$ is true.

5.2.6 Algebra Completeness

The algebra completeness utilises the DandC. Hence, before the algebra completeness discussion starts, it is necessary to describe DandC in more details.

Divide-and-Conquer Algorithm

DandC was briefly explained in previous sections. Now we explain DandC in more details in order to justify its usage in the research. As it has been mentioned in the previous section, the divide-and-conquer algorithm provides a way to solve a problem by:

1. Breaking it into sub-problems that are themselves smaller instances of the same type of problem.
2. Recursively solving these sub-problems.
3. Appropriately combining their answers.

DandC is widely used within Computer Science. A number of these reasons can be describes as follows:

- **Tackling Complexity**: DandC can be considered as a powerful tool that divides a complex problem to manageable sub-problems and solve them accordingly. As it has been shown in previous section, the complexity of security policies can be divided into sub-policies each of which can be dealt with separately. DandC can be efficiently used with the negotiation policies.
- **Implementation Efficiency**: Within the concept of DandC, the key point is to find a way to break the main problem to sub-problems, everything else can be repeated. This approach can be simply implemented using recursion. Recursion is very efficient approach using functional languages like Scala.
- **Parallelism and Memory Access**: Whilst on the DandC implementation subject, it should be added dividing the problem to separate and independent subproblem will allow developers to utilise parallelism in their code. Parallelism is an interesting subject within the functional programming languages like Scala.

The other interesting fact within the implementation subject is the memory access. Smaller problem produced by the sub-problems, would allow application to deal with relatively smaller issues, thus, the problem can be cached and solved before the system moves to the next one. That is the reason DandC implementation using recursion is usually labelled as a non-memory-intensive solution within the context of functional programming languages.

Algebra Completeness by Utilising DandC

Having gone through the definition part, it was shown how the algebra and its operators are used to express policy integration, the proposed algebra should now be challenged for completeness. In other words, the following simple question must be answered: *Is the proposed algebra capable of expressing all possible policy combinations?* Or simply, *Is this algebra complete?* In order to answer this question, start from two-dimensional combinations matrix in which the algebra is used to combine two different policies.

How a policy combination matrix could help to demonstrate the results of combination for certain operators has been shown. The same matrix can be used to show the completeness of the algebra. The matrix consists of 16 cells and each cell has four potentially different values, namely: $\{Y,Y\},\{Y,NA\},\{N,NA\},\{NA,NA\}$. Hence, the total number of combinations that can be presented by policy combination matrix is 4^{16} = 4,294,967,296. Taking that into account, as the next step, one of these possible available matrix combinations is selected and tested to provide the corresponding algebraic formula for it.

Assume a policy matrix as shown in Table 5.7 has been provided and one would like to find the corresponding policy expression.

fi (P1,P2)	{Y,Y}	{Y,NA}	{N,NA}	{NA,NA}
{Y,Y}	{NA,NA}	{NA,NA}	{NA,NA}	{NA,NA}
{Y,NA}	{NA,NA}	{NA,NA}	{NA,NA}	{NA,NA}
{N,NA}	{Y,Y}	{NA,NA}	{Y,Y}	{NA,NA}
{NA,NA}	{NA,NA}	{NA,NA}	{NA,NA}	{Y,Y}

Table 5.7: A Combination Matrix Example

In order to tackle this challenge, again DandC algorithm has been used. Knowing that $P_i + P\{NA,NA\} = Pi$, one would be able to divide the given combination matrix to a number of different sub-matrices. Each individual sub-matrix also would have 16 cells, but only one $\{Y,Y\}$ cell at the most.

Before continuing, in order to have common vocabulary, let us label each individual cells within a cell with a number starting from top left corner to bottom right one, and assume the corresponding expression for each cell described by $e_i, 1 \leq i \leq 16$. Taking that assumption in mind the table is divided to 16 smaller table each as one cell that results $\{Y,Y\}$. Now it is necessary to find the expression for each $\{Y,Y\}$ cell of those individual simple matrices created above by DandC. In other words, three algebraic expressions have to be found which result $\{Y,Y\}$ for e_9, e_{11} and e_{16}.

P1 \ P2	{N,NA}	{N,NA}	{Y,Y}	{NA,NA}
{Y,Y}	e_1	e_2	e_3	e_4
{Y,NA}	e_5	e_6	e_7	e_8
{N,NA}	e_9	e_{10}	e_{11}	e_{12}
{NA,NA}	e_{13}	e_{14}	e_{15}	e_{16}

Table 5.8: Layout of Expressions Which Result {Y,Y}

Let's visualise this approach by considering the following tables. Assume that the following table have been given where e_2 and e_{11} are two algebric expressions whose result is not {NA,NA}.

P1 \ P2	{N,NA}	{N,NA}	{Y,Y}	{NA,NA}
{Y,Y}	{NA,NA}	e_2	{NA,NA}	{NA,NA}
{Y,NA}	{NA,NA}	{NA,NA}	{NA,NA}	{NA,NA}
{N,NA}	{NA,NA}	{NA,NA}	e_{11}	{NA,NA}
{NA,NA}	{NA,NA}	{NA,NA}	{NA,NA}	{NA,NA}

Table 5.9: A Generic Combination Matrix Example

Using DandC, the matrix can now be divided into two simple matrices and then an expression found for each of them:

It is obvious that combining the above two tables using the additional operator (+) would result in the original table. Now if it is known what could possibly result in a {Y,Y} at the e_2 and e_{11} then it would be simple exercise to provide the answer. Using the algebra operators that have been defined, the algebraic expression which results in {Y,Y} at e_2 and e_{11} would be $\Pi^Y(P_1 \& \neg oP_2)$ and $(\neg a \neg oP_1 \& \neg a \neg oP_2)$ respectively. Using these two formulae any other possibility of these two cells can be formulated. Taking that into account and assuming the formula for the Table 5.9 is called $f(P_1, P_2)$, then the combination matrix can now be formulated as follows :

$$\begin{aligned} f(P_1,P_2) &= e_2 + e_{11} \\ &= \Pi_Y(P_1 \& \neg oP_2) + (\neg a \neg oP_1 \& \neg a \neg oP_2) \end{aligned} \tag{5.2}$$

Table 5.11 shows possible algebra formula for each individual cells that results in{Y,Y} in this two dimensional matrix.

P1 \ P2	{N,NA}	{N,NA}	{Y,Y}	{NA,NA}
{Y,Y}	{NA,NA}	e_2	{NA,NA}	{NA,NA}
{Y,NA}	{NA,NA}	{NA,NA}	{NA,NA}	{NA,NA}
{N,NA}	{NA,NA}	{NA,NA}	{NA,NA}	{NA,NA}
{NA,NA}	{NA,NA}	{NA,NA}	{NA,NA}	{NA,NA}

+

P1 \ P2	{N,NA}	{N,NA}	{Y,Y}	{NA,NA}
{Y,Y}	{NA,NA}	{NA,NA}	{NA,NA}	{NA,NA}
{Y,NA}	{NA,NA}	{NA,NA}	{NA,NA}	{NA,NA}
{N,NA}	{NA,NA}	{NA,NA}	e_{11}	{NA,NA}
{NA,NA}	{NA,NA}	{NA,NA}	{NA,NA}	{NA,NA}

=

P1 \ P2	{N,NA}	{N,NA}	{Y,Y}	{NA,NA}
{Y,Y}	{NA,NA}	e_2	{NA,NA}	{NA,NA}
{Y,NA}	{NA,NA}	{NA,NA}	{NA,NA}	{NA,NA}
{N,NA}	{NA,NA}	{NA,NA}	e_{11}	{NA,NA}
{NA,NA}	{NA,NA}	{NA,NA}	{NA,NA}	{NA,NA}

Table 5.10: How to Use D&C to Find Integrated Policy Expressions

P_1 \ P_2	{Y,Y}	{Y,NA}	{N,NA}	{NA,NA}
{Y,Y}	$\Pi^{y}(P_1 \& P_2)$	$\Pi^{y}(P_1 \& \neg_o P_2)$	$\Pi^{y}(P_1 \& \neg_a\neg_o P_2)$	$\Pi^{y}(P_1 - P_2)$
{Y,NA}	$\Pi^{y}(\neg_o P_1 \& P_2)$	$\Pi^{y}(\neg_o P_1 \& \neg_o P_2)$	$\Pi^{y}((\neg_o P_1 \& \neg_a\neg_o P_2)$	$\Pi^{y}(\neg_o P_1 - \neg_a\neg_o P_2)$
{N,NA}	$\Pi^{y}(\neg_a\neg_o P_1 \& P_2)$	$\Pi^{y}(\neg_a\neg_o P_1 \& \neg_o P_2)$	$\neg_a\neg_o P_1 \& \neg_a\neg_o P_2$	$\neg_a\neg_o P_1 - \neg_a\neg_o P_2$
{NA,NA}	$\Pi^{y}(P_2 - P_1)$	$\Pi^{y}(\neg_o P_1 \& \neg_a\neg_o P_2)$	$\neg_a\neg_o P_2 - \neg_a\neg_o P_1$	$P_y - ((P_1 + P_2) + P_1)$

Table 5.11: Possible Expression for Individual Cells Which Results {Y,Y}

It should be noted that a great level of attention have been given to those formula that result $\{Y,Y\}$ in each individual cells. All other values which could possibly held at each individual cells have been ignored. The reason behind this decision is, if one could find an expression that gives the result $e_i = \{Y,Y\}$ for each specific cell, then the other possible values of the very same cell i.e. {Y,NA}, {N,NA} and {NA,NA} can be expressed using $\neg$ operator because $\neg oe_i = \{Y,NA\}, \neg ae_i = \{N,NA\}$ and $\neg a\neg oe_i = \{NA,NA\}$.

Going back to the example at the beginning of this section, the corresponding policy expression can now be presented as:

$$\begin{aligned} f_i(P_1,P_2) &= e_9 + e_{11} + e_{16} \\ &= \Pi^{Y}(\neg a\neg oP_1 \& P_2) + (\neg a\neg oP_1 \& \neg a\neg oP_2) + (P_y((P_1+P_2)+P_1)) \end{aligned} \tag{5.3}$$

Using the above approach, the algebra has been proven to be complete for a two-dimensional matrix, which in turn proves that the following theorem is true:

Theorem 3 : Assuming M presents combination matrix results of P_1 and P_2, there would be at least one policy expression $f(P_1,P_2)$ that describes $M(P_1,P_2)$ using the minimal set of operator that is $f(P_1,P_2) = M(P_1,P_2)$.

In addition to above, the theorem can also be expanded for expressions of the

policies that are involved in the negotiation as follows:

Theorem 4: Let M_i denote a combination matrix in sequence i of a complete trace of a given policy P that requires partial evaluation with regards to Ω and Σ , then an algebraic expression exists which describes policy P.

Proof: The keywords here are *Complete Trace* of policy P which implies that the number of sequences in the trace is finite that is $(1 \leq i < \infty)$. So on one hand there is a finite number of sequences and on the other hand in theorem 3 it has been already proven that the algebra is complete. In other words, for any combination matrix there exists an algebra expression that contains algebra operators representing that table (policy). Hence, combining these two facts proves that that the algebra is capable of representing a policy P that is participating in a negotiation using algebraic expressions.

What must really be taken into consideration is the fact that the minimal set of operators needed to describe and formulate policies as above are $\{P_+$, P_-, + , - , & , $\neg a$, $\neg o$, Π $\}$. The operator H (defined in section 5.2.4) is needed to describe and divide complex policies such as, policies with negotiation capabilities into a finite sequence of simple policies in which the above operators would be sufficient to describe them in detail.

5.3 Summary

5.3.1 Chapter Summary

Algebra as a composition framework has been utilised since security policies have been introduced. Using algebra it would be possible to show how policies are combined whilst they retain their independence. A number of algebras with different characteristics and formalisms have been introduced to help security policy writers to describe their policies more precisely and indeed more efficiently.

In this chapter, some of the algebras that have been introduced during the last

decade have been reviewed. These were examined from different perspectives, leading to the selection of the one that fits in the context of this research.

Furthermore, the selected algebra was evaluated against security policy languages that were chosen in the previous chapter and areas that need improvement were identified and addressed accordingly.

5.3.2 Research Contributions of the Chapter

As mentioned earlier, security policy languages are changing and improving at a rapid pace and as a result, their corresponding algebras (if there are any) are often ignored, forgotten or cannot keep up with the pace. As an example, one of the areas that has attracted attention in recent years is negotiation, but as it has been discovered, there is no algebra existing to generally express this functionality in detail.

In addition, it has been noticed, the algebra for security policy languages did not pay that much attention to partial and conditional evaluation of security policy languages. Thus, this characteristic of relatively new policy languages became a prime focus of this research. In this chapter, a generic algebra was chosen that fits partially in the context of the present research and evaluated against a negotiation scenario. The algebra was enhanced by adding a new operator, *Trace*. Trace comes into use when describing complex security policies that require partial and conditional evaluation that is often unknown at compilation time, is desired. Policies with negotiation capabilities that go beyond the boundaries of an environment is a perfect example of those policies that require partial evaluation. A few theorems that utilise the newly introduced operator, Trace, were introduced and proved in turn.

Knowing that a combination framework, i.e. an algebra, that is capable of expressing complex policies proven to be functional, it can be stated with confidence that the proposed abstract framework, which sits in the same level as the presented algebra, would be able to operate as expected. Thus, the focus now will be on the technical side of the framework and the step by step design of the framework from

the next chapter.

Chapter 6

Domain Specific Language

In this chapter, the research presents:

- The definition of domain specific language,
- The advantages, disadvantages and requirements of domain specific language,
- The domain specific language detailed implementation phases and patterns,
- The research contributions of the chapter.

6.1 How to Start the Design Phase

Previously, it has been mathematically proven that the security policy languages can be mapped onto an abstract combination framework i.e. algebra. The next step is to design and implement a computerised version of the combination framework. Back in Chapter 3, whilst the overall requirements and structure of the framework was presented, it also has been shown that the framework for security policy languages would require the following two distinctive components:

A) **Abstract Language**: In essence, the framework for security policy languages allows security officers and/or administrators to communicate with the security

framework from single point of view, regardless of the underlying infrastructure. In order to achieve such a goal users have to code their security policies and they do need a medium to achieve such a goal: An *Abstract Language*. The grammar, syntax, keywords and operators of the abstract language dictate how users must write their security policies.

B) **Infrastructure of the Framework**: There is an array of off-the-shelf components that need to be architecturally tied to each other using a carefully tailored and designed, well-written code which must be started from the scratch. The combination of all these components, modules, etc. shape the infrastructure of the framework.

Analysing the framework and the abstract language from the users point of view shows that framework (with the help of its abstract language) conceals the complexity of the security policy languages and provides the users with a much simpler language that works regardless of the underlying infrastructure. With that definition in mind, if the search criterion is widened and similar approaches are reviewed, it is evident that similar challenges have already been addressed by researchers in the very similar fields.

A few other abstract languages with great level of similarities are reviewed rigorously as part of this research. As an example, a Database Administrator (DBA) can easily query a database using a specific language called the Structured Query Language (SQL). The DBA does not need to know anything about the underlying database and/or the way it works. Whether the database that the DBA is utilising is a simple open-source database such as, MySQL or whether it is a multi-tiered database running on server farms, the DBA can execute the query.

Hyper-Text Mark-up Language (HTML) also falls in the same category. A web designer can design a web page without being concerned with which web browser will render the page. Despite the tools and technologies that eventually generates and delivers the HTML page to its users, the web designer can focus on the design

and development of the page.

There are lots of other frameworks available that can be referred to ANother Tool for Language Recognition (ANTLR) [129], eXtendable Mark-up language (XML) and many others can be presented as an example. Over and above all of their benefits, each of these tools, products, frameworks, etc. hide the complexity of the underlying infrastructure from the users. In addition, these frameworks provide an abstract language, which has been tailored to the users specific needs and requirements as well as their domains needs. These frameworks provide their users with a domain specific Language or **DSL**.

6.2 Domain Specific Language

Before exploring the DSLs in more detail, the meaning of the programming language must be reviewed. In the context of this research, the best definition of a programming language has been provided by [83].

"*A programming language or computer language is a standardised communication technique for expressing instructions to a computer. It is a set of syntactic and semantic rules used to define computer programs. A language enables a programmer to precisely specify what data a computer will act upon, how these data will be stored and or transmitted, and precisely what actions to take under various circumstances*" [83]

As per the above definition, DSLs are essentially programming languages. Despite this, compared to GPLs, which are designed to cover a broad range of applications from business to scientific computing, DSLs are designed for a special purpose and are usually aimed to address a very specific challenge; therefore, they provide a limited level of expressiveness. Using DSLs facilitates the expression of solutions for domain problems with less effort. Although DSLs are programming languages, due to the usage nature of DSLs they are usually designed in such a way

as to be more easily read by their users. The DSLs provide higher abstraction and compactness and, therefore, better readability, which enables a larger group of people with less programming knowledge to be productive. The DSLs usually have a clearly defined domain focus.

DSLs are not a new concept. These languages have had several aliases over time, such as: *special-purpose languages*, *end-user languages* or as Bentley [47] called them *little languages* before the term domain specific language was coined by Fowler [87]. The history of DSLs dates back to 1957 when a language for numeric controlled machines was developed at the Massachusetts Institute of Technology (MIT), which can be considered as the first modern DSL developed [58]. The DSLs have also been used by researchers and have assisted users for decades.

It seems that the academic definition of DSLs have changed over the past few decades. In the context of this research, Menricks description regarding DSLs can be used as follows:

"*DSLs are languages tailored to a specific application domain. They offer substantial gains in expressiveness and ease of use compared with GPLs in their domain of application*" [125]

However, the definition provided by Fowler [87] would perfectly fit the context of this research:

"*A computer programming language of limited expressiveness focused on a particular domain*" [87]

Although this is a compact and very well-defined expression for DSLs however it comes with four key elements which worth exploring in further detail:

- **Computer Programming Language**: A DSL is used to instruct a computer to perform task or tasks, hence, as per the definition of modern programming language given above, it is a computer programming language which will be executable by a computer.
- **Language Nature**: As DSL is a programming language, it should have a sense of fluency where the expressiveness comes not just from individual expressions but also from the way they can be composed together.
- **Limited Expressiveness**: Compared to a general-purpose programming language which provides lots of capabilities, a DSL supports a bare minimum of features needed to support its domain.
- **Domain Focus**: A limited language is only useful if it has a clear focus on a small domain. The domain focus is what makes a limited language worthwhile.

6.2.1 DSL Stakeholders

There are three typical DSL stakeholders at three different levels:

- **System/Software Engineers**: Who are responsible for choosing or implementing an appropriate DSL. In the context of this research, software engineers who would be responsible for coding the actual framework will fit into this category.
- **Customers**: Who are responsible for providing feedback on descriptions produced using a DSL. Security officers who provide feedback and verify the outcome of the DSL will be categorised into this group.
- **Developers**: Who are responsible for constructing and managing DSL descriptions. Within the context of this research, these would be the security administrators who are responsible for interpreting the actual security poli-

cies to DSL scripts, running them on the framework and providing customers with the results.

The current trend towards end-user programming suggests that in some contexts, the roles of customer and developer may be combined [110].

6.2.2 Boundaries of DSL

When it comes to DSL, an important issue is identifying the boundary of DSL. More specifically: what constitutes a DSL and what does not? Compared with general purpose languages, DSLs show a tendency towards the construction of domain concepts in more details. As a result, a DSL will more accurately represent domain practice and will more accurately support domain analyses.

In addition to the above, typically DSLs utilise external tools and products for different purposes as opposed to building all the components within the code. Error handling and debugging can be presented as examples. Finally, a DSL is often computationally incomplete [110].

Despite all the definition presented above, still the border between DSLs and GPLs cannot be easily distinguished. For example, COBOL was considered a GPL but also a DSL for business applications. Prolog is another example of ambiguity. Although Prolog is a programming language, it also can be classified as a DSL for applications specified by predicate calculus.

6.2.3 Requirement for DSLs

Similar to designing software application, prior to the discussion of the design of the system, the requirements of the system must be defined in detail in order to justify the use of that particular application. The DSLs are not an exception to this rule. In this section, the requirements of DSLs will be reviewed and will be mapped to the current research accordingly.

Generally, some of the requirements for GPLs apply directly to DSLs. The core requirements for DSL are as follows:

- **Conformity**: The language constructs must correspond to important domain concepts.
- **Orthogonality**: Each construct in the language is used to represent exactly one distinct concept in the domain.
- **Integrability**: The language and its tools can be used in concert with other languages and tools with minimal effort.
- **Extensibility**: The DSL (and its tools) can be extended to support additional constructs and concepts.
- **Longevity**: The DSL should be used and useful for a significant period of time.
- **Simplicity**: The language should be as simple as possible in order to express the concepts of interest and to support its users.
- **Quality**: The language should provide general mechanisms for building quality systems.
- **Supportability**: It is feasible to provide DSL support via tools for typical model and program management, such as, creating, deleting, editing, debugging and transforming.

In addition to above, there are optional requirements for DSLs which may not necessarily appear on all DSL implementations. These can be presented as:

- **Scalability**: The language provides constructs to help manage large-scale descriptions. Of course, some DSLs will only be used to build small systems.
- **Usability**: This includes requirements such as, space economy, accessibility, desirable understandability that may be partly covered by the core requirements [110].

Taking the above definition into account it would be easy to justify the use of DSLs in the current research as follows:

- **Conformity**: As discussed in Chapter 1, the framework and its corresponding security policy language will be designed to cover important concepts in security aspects of multidimensional organisations.
- **Orthogonality**: The abstract security policy language that comes with the framework will be designed in a way to precisely describe security policy languages.
- **Integrability**: The framework will be designed in a way to easily be coupled with legacy and new security infrastructure.
- **Expandability**: In addition to the above, the framework originally will be designed to work with only three security policy languages, however, there will be no limitation to expand to other security policy languages.
- **Longevity**: The usage of framework is not restricted to a period of time.
- **Simplicity**: The framework (with the help of its ASPL) will assist security experts to describe their security policies with less effort.
- **Supportability**: Describing security policies using abstract security policy language may not be easily achievable, therefore, the framework will be designed in a way to assist users to achieve their goals.
- **Usability**: The usability of the framework is reviewed in detail in Section 1.3.1.

6.2.4 Advantages of DSL

Implementing a DSL is always an interesting subject and has several advantages. A few of them are discussed below.

- **Involving Domain Experts in the Software Development Process**:If properly designed, DSLs provide an opportunity to involve the domain experts in the architecture of the software product. This does not claim that software architects are no longer needed if DSLs are used, but, claims that DSLs extends the range of people to be able to contribute to the architecture of the software product [41].

- **DSLs are Concise**: Therefore, DSLs are easy to look at, see, think about and show. Roam [134] calls look at, see, think about and show, the four steps to visual thinking. The DSLs due to their limited vocabulary are often designed in a manner similar to human language, reducing the semantic distance between the program and the problem.

- **Reusability**: Taking the conciseness of DLSs into account as well as the domain fitting notation, DSLs are (to a certain degree) self-documenting. This in turn results the embodying of domain knowledge which eases reuse and conservation [75].

- **DSL Improves Development Maintainability**: Perhaps the single most important benefit of using domain specific languages is that the domain specific knowledge is formalised at the right level of abstraction hence, modifications are easier to make and their impact is easier to understand.

  Taking that into account the longevity requirement of a DSL which indicates DSLs should not be useful for a single period of time, DSL-based development tends to produce a higher payoff in the long run of development life cycle [94] [74].

- **Validate at Domain Level**: A general purpose language compiler does not know anything about domain concepts, whereas, a DSL can be checked for domain constraint during the compilation phase [63].

6.2.5 Disadvantages of DSL

Like many other tools and products used within the IT industry, DSL usage comes with unique disadvantages, however, majority of these disadvantages are related to the implementation of a new language.

- **Design a Language is Arduous**: Technically DSL design is a language design and no matter how easy and user friendly the language is, terminology design is a complex and difficult task. That is why instead of designing a completely new language with its complexities, most DSLs are embedded within a higher-level language.
- **Designing DSL Could be Expensive**: Designing a DSL could be expensive as the task must be performed by experienced programmers and involves intense collaboration and communication with domain experts. Design of a DSL must be financially justified first.
- **Expandability of DSLs is Challenging**: The nature of DSL is to focus on a specific problem of a domain. The DSLs are usually evolving iteratively and independently. In an enterprise application, which usually utilises more than one DSLs at a time, often an inevitable task comes to force, which requires combining a few DSLs together. That certainly raises a concern because combining DSLs that are independently and iteratively expanding is not easy [94].
- **Language Cacophony**: As mentioned, DSL is effectively another language, hence, the language must be taught to users. That implies the learning curve of DSL based applications could be marginally higher than expected. Additionally, the learning curve increases the overall cost of developing a DSL-based language [94].
- **Blinkered Abstraction**: The uniqueness and perhaps usefulness of DSL-based application is that it provides its users with an abstraction which allows the expression of domain behaviour with less effort. But the danger here is

this abstraction puts blinkers on users thinking. Blinkered abstraction problem is a general concern which applies to all abstraction but due to the fact that DSLs provides a more comfortable way to manipulate abstraction, DSLs make blinkered abstraction even worse [87].

- **DSL performance**: Often a DSL will suffer from a lower performance than a hand written software, as it is yet another layer of indirection. As long as performance is not critical, then the other DSL benefits will make this a minor problem. In some cases, performance can be equal or faster due to optimisation on high abstraction level but in most cases the potential is limited [94].

6.3 DLS Implementation Phase and Patterns

As per any software development cycle, DSL implementation is divided into major steps: design and development. However, as per Mernik [125], a more fine-grained implementation of DSL can be presented by breaking down the cycle into five stages namely: *Decision, Analysis, Design, Implementation and Deployment*.

Compared to bespoke software development, DSL development is not a simple sequential process. The decision process may be influenced by preliminary analysis. Analysis itself may have to supply answers to questions arising during design and the design is often shaped by implementation considerations [125]. We already covered the fact that DSL implementation is expensive, so in order to minimise the risks, each stages of DSL implementation have to be rigorously followed and mapped to the research.

6.3.1 Decision Phase

It has already been pointed out that due to the nature of DSL, its development is not categorised as a cost-effective approach. The DSL investment must pay for itself in

the long term, therefore, DSL development must be justified beforehand. In some cases, using an existing DSL could be considered as a more appropriate approach. In addition, adopting a new DSL requires less expertise. The drawback here could be using an existing DSL which is not very well publicised could be too risky and even more expensive due to its possible maintenance in the long term.

To be able to make decision on when to use domain specific languages, DSL decision patterns are identified and listed below. Most of these patterns help end users with less programming expertise to perform software development. Majority of these patterns would result in the improvement of software economics.

- **Notation**: The availability of appropriate new or existing domain specific notation is the main characteristic of this pattern. The two common sub-patterns are A) transformation of a visual to a textual notion and B) add user-friendly notation to an existing API or turn an API into a DSL. MSC [17] that is used for telecoms system specification for system architecture design can be presented as an example.

- **AVOPT**: domain specific *Analysis, Verification, Optimisation, Parallelisation and Transformation (AVOPT)* of an existing program written in GPL is considered as an arduous and time-consuming task perhaps due to code complexity and/or lack of documentation. However, use of an appropriate DSL makes these operations possible. This pattern overlaps with most of the other patterns. OWL-Light [138] used for programmable web ontology is an example.

- **Task Automation**: In software development, programmers often spend too much time to generate codes which follows the same pattern. In such a scenario, a code generator driven by an appropriate DSL would ease the operations. RoTL [123] used for traffic control can be presented as an example.

- **Product Line**: Some software products do not exist as a single stand-alone application. They often share common architecture and are developed from a common set of basic elements. In such scenarios, use of an appropriate DSL

could ease automated assembly. An example of this would be ASDL [149] used for Language processing.

- **Structure Representation**: An appropriate DSL is often used to represent structured data in more appropriate, human-readable and maintainable ways. This category can be represented by JSON [66].

- **Data Structure Traversal**: Traversals over complicated data structures can often be expressed better and more reliably in a suitable DSL such as, SQL [124].

- **System Front-End**: Providing users with an appropriate DSL would often facilitate the handling system configuration and adoption such as, Nowra [31], which is used for software configuration.

- **Interaction**: Using well-defined DSL is often needed for complicated or repetitive inputs for menu or text-based interaction with software application for example, Microsoft excel macros.

- **GUI Construction**: Often GUI construction is performed by using an appropriate DSL. For example, XML and HTML represent domain specific languages for GUI construction.

Choosing the Appropriate Decision Pattern

In the context of this research, a few of the above mentioned patterns would come into play simply as they match with what the research intends to implement. They might overlap to some degrees. The ones that are not listed below are excluded due to their incompatibility.

- **Task Automation**: As it has been mentioned previously, majority of scenarios written in various security policy languages can be generalised using an abstract security policy language allowing the task automation pattern above to be applied to facilitate a more robust code generation.

- **Interaction**: In addition, developers (please refer to the stakeholders definition above) of security policies will be using a text-based interaction to communicate with the framework (as described in Chapter 3), therefore, interaction pattern can be chosen.

6.3.2 Analysis Phase

Assuming that the decision is made to develop a new DSL in order to extract a great level of detail about the domain, including objects and operations which are commonly used in that particular domain, the domain *Analyse Phase* must commence. Inputs can be provided from different sources which have implicit and explicit knowledge about the domain that includes but is not limited to *Domain Experts Knowledge, Technical Documentation, Customer Feedback and Reviewing the Code* (most probably the existing GPL code). The output of the phase would be a domain specific terminology and domain specific semantic in an abstract mode.

The following patterns have been identified for DSL analysis which are as follows:

- **Informal Pattern**: The informal pattern, which can be predicated from its name, follow no formal domain analysis process. According to Menrik *et. al.* [125], most DSL developments are done without any formal analysis. That often leads to incomplete requirements which increases the cost of DSL development through maintenance. It is clear that the informal analysis would be a suitable approach for a simple domain with limited requirements.

- **Formal Pattern**: The analysis of the domain can be preformed using well-defined and known methodologies. This approach is known as formal analysis. Unsurprisingly, the disadvantages of an informal pattern are addressed by the formal pattern; using a formal pattern facilitates the prevention of the omission of important parts of the domain and leads to more complete requirements.

There are several methodologies for domain analysis such as: FAST (Family-Oriented Abstractions, Specification and Translation) [150] , FODA (Feature-Oriented Domain Analysis) [154], ODE (Ontology-based Domain Engineering) [81], DSSA (domain specific Software Architectures) [142], DARE (Domain Analysis and Reuse Environment) [89], or ODM (Organization Domain Modeling) [140] . It is useful to know that the majority of methodologies that are used for formal domain analyses come from another research field which is, *Domain Engineering* (DE) [118]. Domain engineering refers to the systematic modelling of a domain [125].

Domain Driven Design (DDD) [80] which has attracted attention over the last few years can also be presented as another methodology of DSL analysis. DDD is a development technique which focuses on understanding the customer's problem and the environment in which the customer works. In this definition, the *Problem Domain* refers to the problem that will be solved by the output of DDD development. In addition to the defined problem domain, based on the customers desires and needs, the developer builds a domain, which can be a representative of all these concepts called *Domain Model*. The model is discussed with the real users and customers and through an iterative process, it is enhanced and refined [80].

While on the subject, it would be beneficial to talk about another concept called *Ubiquitous Language*. On an enterprise software development team, while many different *Actors* with different levels of responsibilities participate in the development process (e.g. developer, product owner, system customer etc.), in order to reduce the miscommunication between the involved parties, a language should be defined in which the key terms of the problem domain are described in a language understandable to both the domain expert and the developer. This language is called *Ubiquitous Language*. Creating a ubiquitous language involves creating a glossary in which the key terms are explained in a way that is understandable to both the domain expert and the developer. This glossary is also updated throughout the project [80].

Choosing the Appropriate Analysis Pattern

While a formal analysis of the surrounding domain of the framework would have been an ideal approach for the project, due the scale of the DSL that will be implemented as part of this research, informal and formal approaches were combined in order to gain the advantages of both patterns. As a result, in a series of informal interviews with the domain experts and having a DDD methodology in mind, an ubiquitous language has been defined. The language that originally used to communicate with the domain experts and other parties involved, later became the base for the ASPL. In addition to above using the ubiquitous, the domain model was defined, modified and enhanced through a series of sessions and presented to the actual users.

6.3.3 Design Phase

Design phase approach is highly dependent on the previous phases and chosen approaches. Having said that, often DSL designers believe that the easiest way to design a DSL is to *host* it on an existing language. There are possible advantages to adopting such an approach, such as, easier implementation due to a reduced learning curve for the development team with that particular language. In addition, some languages such as, Scala or Ruby, provide users with tools and features that can be used to leverage the language [125].

The six possible approaches for DSL design are listed as follows:

1) **Piggyback**: The piggyback structural pattern uses the capabilities of an existing language as the base for the DSL that is to be designed. Often a DSL needs standardised support for common linguistic elements, such as, expression handling, variables, subroutines or compilation. By designing the DSL on top of an existing language, the needed linguistic is provided for free. The piggyback pattern can be used whenever the DSL shares common elements with an existing language [141]. Possible examples of the piggyback design are: YACC [105]

and LEX [114] processor.

2) **Language Extension**: The language extension pattern is used when adding new features to an existing language. Often an existing language can serve new needs by just adding new features to its core features. This pattern challenges the designer to integrate the required features of DSL into existing language [141]. A DSL that follows the extension pattern is SWUL [56] which, supports the development of Java SWING GUIs and is embedded into Java.

3) **Language Specialisation**: Developing a new DSL does not always mean creating something new. A more uncommon pattern is specialisation. In some cases, the full power of an existing language may prevent its adoption for a specialised purpose requiring the language to be reduced to meet the needs of a special domain. One example is, OWL-Lite [69], which is a subset of the Ontology web language [141].

4) **Source to Source Transformation**: There are cases where the DSL cannot be directly designed on top of an existing language using language extension, specialisation or the piggyback pattern. In those scenarios, it is often possible to leverage the facilities provided by existing language tools using a source-to-source transformation technique. Using this technique, the DSL source code is transformed via a suitable translation process into the source code of an existing language [141] [125].

5) **Data Structure Representation**: Assuming that the data structure traversal method has been chosen during the decision phase, then the data structure representation design pattern would be a suitable pattern to use. The data structure representation pattern allows the declarative specification of a complex data. Complicated structures are better expressed using a language rather than their underlying representation such as graphs.

6) **Entirely New Language**: While using an existing language to create a DSL has its own advantages and could be considered a favourite approach for developers, the DSL design and perhaps implementation will always be limited to the

host language boundaries, severely compromising the flexibility of the DSL. If the DSL will be used among wide range of users and will be subject to future enhancement, modification and expansion then using a host language as a vehicle for implementation may not be an appropriate approach. In such a scenario, design and implementation of a completely new language whose design bears no relationship to any existing language would be a suitable approach. In practice, development of this kind of DSL can be extremely difficult and the costs of design and implementation of a new DSL can be considerably high. Considering all these facts, unless adoption of this approach is justified in advance, this pattern will not be developers' first choice [125].

Designing a new DSL is no different than designing a GPL and like many GPLs its design should follow the principals, which have been followed by well-known GPLs design criteria such as, readability, simplicity, orthogonality, etc. The design principles listed by Brooks [48], as well as Tennents design principles [143] retain some validity for DSLs .

Irrespective of the DSL design pattern chosen, DSL design must take into consideration both the characteristic of DSLs as well as the fact that users may not be programmers. Generally, DSL adopt established notations of the domain and the design should suppress a tendency to improve them. As stated in Wile [152], one of the lessons learnt from real DSL experiments is:

"*Lesson T2: You are almost never designing a programming language. Most DSL designers come from language design backgrounds. There the admirable principles of orthogonality and economy of form are not necessarily well-applied to DSL design. Especially in catering to the pre-existing jargon and notations of the domain, one must be careful not to embellish or over-generalize the language. Lesson T2 Corollary: Design only what is necessary. Learn to recognize your tendency to over-design.*" [152]

Choosing the Appropriate Design Pattern

As previously mentioned, the flow of activities in the five stages of implementing a DSL are not always sequential and each stage could directly and/or indirectly influence the decision on previous steps. Taking todays modern languages, such as, Scala or even the equivalent .Net language, would provide a wide range of features, which makes DSL a successful, yet cost-effective task. Taking this into account would narrow-down the design pattern dramatically. Patterns which are not fit for the purpose of this research have been ruled out as below:

- **Entirely New Language**: Due to the difficulty and cost-effectiveness issues related to this approach, its usage must be justified from the outset. In addition, this pattern targets DSL that are overcomplicated and cannot easily fit into other options.
- **Source-to-Source Transformation / Data Structure Representation**: Since this research does not deal with a complex data structure in the DSL and the source for our DSL is not available to reuse, the above mentioned patterns are not effective in this case.
- **Language Extension**: Ruling out the other patterns leaves the project with piggyback pattern and/or its derivatives (i.e. language extension or specialisation). Due to the nature of security policy languages and the scenarios that they cover, simplicity of the to-be-developed DSL is predictable, which in turn rules out the language extension. That is the design methodology for complicated DSLs, which cannot be defined by the other two methods.

The chosen design methodology must be either piggyback or language specialisation; however, at this stage, it is difficult to distinctively decide between either of these two methods, hence, it was necessary to continue on to the next phase and revisit the design phase later.

6.3.4 Implementation Phase

The final step towards the creation of a DSL is implementation. Similar to other steps, there are different patterns that must be considered and the most appropriate one should be chosen. Although this may sound similar to previous steps, in reality the DSL implementation is the most difficult task among all steps, which have already been covered because DSL implementation patterns are solely used for DSL development and attract no other attentions. In addition, compared to other GPL development patterns, they are not very well known due to the lack of documentation. The implementation decision can influence the needed development efforts and should be considered carefully. As with any decision, analysis and design have different possible implementation patterns that have been identified and discussed below:

- **Compiler / Interpreter**: Perhaps the most logical way to approach DSL implementation is the very same method for GPLs; using a *Compiler* and/or *Interpreter*. A wide range of GPLs have been implemented in this way. An interpreter interprets the DSL code in a four stage cycle of recognise, fetch, decode and run. Both the interpreter and compiler would have a great level of similarity but, generally the implementation of the interpreters requires much less effort. In addition, greater simplicity and control over the execution environment as well as easier extension can be presented as advantages of an interpreter over a compiler

  As per the other patterns, the compiler/interpreter have both its advantages and disadvantages. Because the compiler or interpreter must recognise the code first, using this pattern could utilise strong syntax checking and error reporting capabilities. Besides domain specific analysis, verification, optimisation, parallelisation and transformation (AVOPT) is also possible when this pattern is in use. Generally, AVOPT would be an effective tool when the user community is large. Finally, due to the fact that developers are in full control of almost every step of AVOPT, DSL syntax can be close to the notations

used by domain experts.

On the other hand, DSL implementation using this pattern requires a lot of effort due to the fact that a complex language processor must be implemented from the ground up. In addition, language extension is difficult mainly because most language processors are not designed with extension in mind.

- **Preprocessor**: Pre-processing happens before interpretation or compilation commences. In other words, the pre-processor decouples the DSL code from the underlying base language. That gives the developer a great level of flexibility which comes in handy when developing the DSL. However, this implies that syntax checking and any error reporting and error handling should be postponed to compilation or interpretation time. There are different sub-flavours of this pattern available, most of which map to a corresponding design pattern. These sub-patterns are detailed below:

  - **Lexical Pre-processors**: Lexical pre-processors or in simple words *macros*, are a classic example of DSL implementation using pre-processors. Macros work by simple textual search-and-replace at the token rather than at the character level. That usually is executed prior to any parsing performed, according to user-defined rules. A classic use of macros is demonstrated in the computer typesetting system LaTeX and its derivatives [84].
  - **Source to Source**: Source-to-source is another type of pre-processor DSL implementation. Source-to-source pre-processors entirely translate the DSL code to the base language.
  - **Pipeline**: If the DSL code gets broken down into a set of sub-languages and each set is processed by a different processor then the whole chain presents the pipeline architecture for pre-processing.

- **Embedding**: Assuming a GPL allows the use of user-defined abstract data-types and/or operators, then the DSL development becomes much easier by extending the GPL, which will be known as the *host language*. Then describing domain specific notation using new data-type and operators would

be possible. As the DSL code is directly embedded into the host language, it will obtain all the capabilities of the host language for free.

- **Extension**: Developers usually find it easier to make domain specific changes to a GPL. These changes will result in an embedded tailored language without sacrificing the generality and library support of the GPL. Although such an approach would be suitable for DSL with limited expressiveness, the solution may not be appropriate for a more complex approach.

  Although extending the language is presented as a separate pattern, in reality, it is another form of embedding DSL constructs in a host language. While the embedding pattern attempts to construct DSL notation by simply defining new data-types and/or operators, the extending pattern has a tendency to do the same by full-blown DSL features. Having this in mind the best place to start language extending would be compilers; however, compilers get designed without having extension in mind, hence, the extending pattern is a much harder approach compared to other available patterns [125].

- **COTS**: *Commercial Out of The Shelf (COTS)* approach can also be used while implementing DSLs. In this approach, the existing tools and/or notations are applied to a specific domain. Typically, this approach involves applying existing functionality in a restricted way, according to domain rules.

  With the rise of the Java Virtual Machine (JVM) based languages like Scala [127] and Ruby [144], *Model Driven Engineering (MDE)* became an interesting subject for developers and researchers [50]. The MDE approach aims to raise the level of abstraction in program specification and increase automation in program development. The MDE approach encourages users to utilise models at different levels of abstraction for developing systems, thereby raising the level of abstraction in program specification. Tools for MDE are often called *Model Driven Software Factories* [70].

  The beauty of the COTS pattern for developing DSLs is that it allows DSL architects and developers to define DSL syntax, artifacts, editors, etc. and implement the entire platform using a model driven software factory. Mendixs

model-driven enterprise application platform [73] is targeted at the domain of service-oriented business applications [71] and can be presented as an example. The XML-based DSLs with board range libraries and tools, including parsers, such as, SAX [122] or DOM [7], the analyser Xquery [27] and the transformer XSLT [13] also fall in this category.

- **Hybrid**: Knowing that each single pattern has its own advantages and disadvantages makes choosing a single pattern to design and implement a DSL more difficult than it appears. While analysis, verification, optimization, parallelization and transformation (AVOPT) seem necessary in DSL implementation and can be achieved using the compiler approach, embedding the DSL into an host language will significantly reduce the DSL implementation efforts, and while pre-processors pattern with its decoupling feature gets the developer as closely as possible to the domain concepts, usage of COTS pattern might seem a very cost-effective approach for DSL implementation.

  Considering the difficulty of choosing a suitable pattern,in order to maximise the advantages of implementing a DSL , sometimes it makes sense to combine all these patterns into a single project, which leads to a final pattern, *the Hybrid*. There are a number of examples provided that combine the compiler and embedding patterns [78] [125].

Choosing the Appropriate Implementation Pattern

As discussed, choosing the appropriate pattern for DSL implementation is not easy. As a direct consequence, the main characteristics of each pattern were compared against the framework's requirements in order to determine which pattern cannot easily be adapted to the project. The results are detailed below:

- **Compiler/Interpreter**: Compiler and interpreters are used in DSL development projects where AVOPT is a mandatory requirement of the DSL. Analysis and verification are a mandatory part of almost all DSLs; however, optimisation and parallelisation do not play a vital role in the framework DSL.

Compiler and interpreters are more appropriate in DSL projects with a large number of users, whereas, the community user of the framework is restricted due to the nature of security policy languages. Hence, the compiler and interpreter are not a prime implementation choice for this research.

- **Preprocessor**: Pre-processors could have been a good DSL pattern for this project, but due to the fact that error reporting is postponed to the compilation time, this would impose a threat to the efficiency of the DSL. Ideally, the users would be provided with the possible errors that have been made before the compilation time.

- **Source-to-Source Transformation / Pipeline**: Source-to-Source transformation and pipeline processes do not fit in the context of this research, as A) a source code for the DSL is not available and B) due to the nature of the scenarios described by security policy languages, pipeline is not an effective implementation approach; therefore, none of the above patterns can be easily adopted in the implementation phase.

- **Language Extension / COTS**: Language extension has already been ruled out in the previous section. Moreover, the efforts this pattern requires do not fit within the project plan, thus, language extension is not appropriate. The COTS pattern could have been a suitable approach for this project, but considering the scope of the project, COTS and especially the MDE approach could not only over engineer the entire plan, but it is also not justified financially.

By ruling out the patterns that do not fit in the context of this research, the embedding approach remained the only available option. A diagram that illustrates the sequence of steps to choose an implementation pattern has been presented in Figure 6.1.

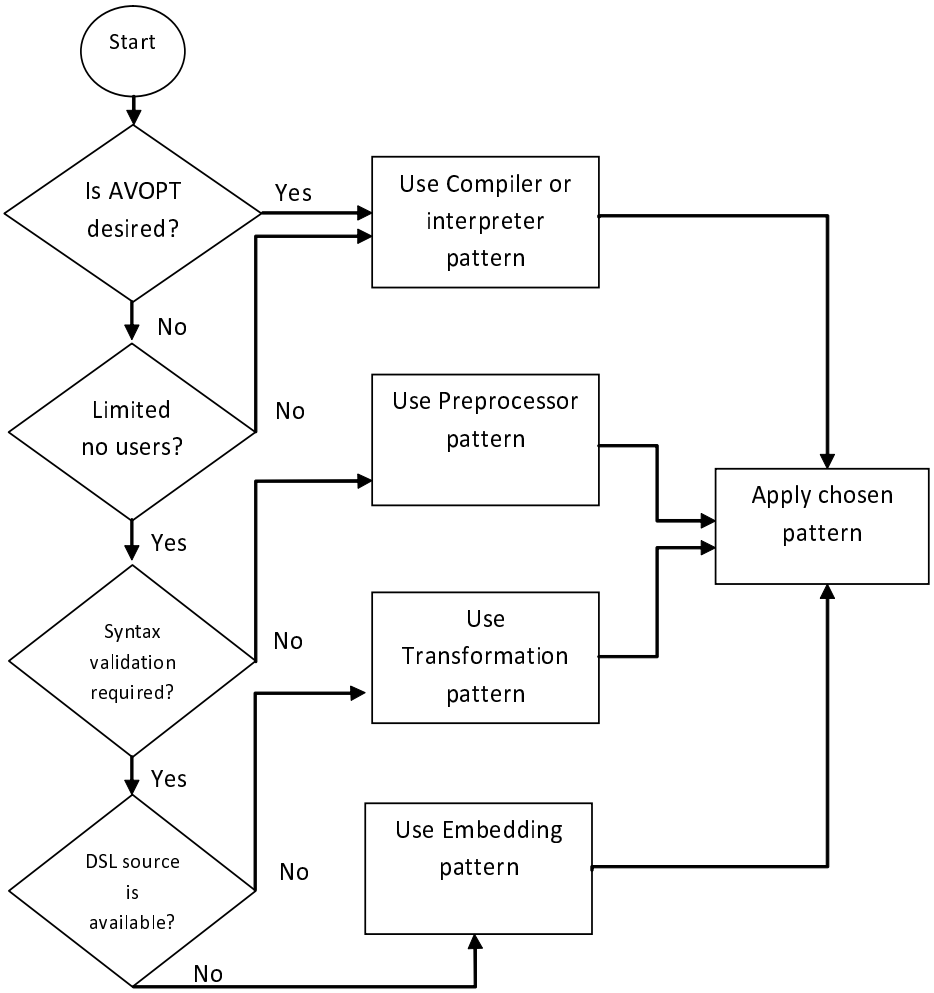


Figure 6.1: How to Choose a DSL Implementation Pattern

6.3.5 Exploring Embedding Pattern

Since the embedding pattern is the desired approach, the thesis will review this pattern in more detail. Technically, there are only two sub-patterns available to implement DSLs using the embedding pattern: *Internal* and *External*. Any other sub-patterns are effectively another flavour of these two patterns [87]. Despite the availability of only two patterns, choosing the right implementation is not a simple task. These two patterns are described below briefly.

- **Internal DSLs** are particular ways of using a host language to lend the host language, the feel of a particular language. This approach has recently been popularised by most JVM functional languages such as, Ruby [144], Groovy [109] or Clojure [96]. A. Internal DSLs are also referred to as *Embedded DSLs* or *Fluent Interfaces*. This pattern is the corresponding implementation of the piggyback design pattern.

- **External DSLs** External DSLs have their own custom syntax and the developer must write a full parser to process them. There is a very strong tradition of doing this in the Unix community. Many XML configurations have ended up as external DSLs, although XMLs syntax is not suited to this purpose. Despite the fact that the development of a external DSL must start on implementation of a full down parser, due to the evolution of JVM-based languages, like Scala or even .Net languages like C#, currently developers are provided with a full parser for free and as a result, external DSLs could also be hosted on top of an existing language like internal DSLs. External DSL development on top of an existing host language corresponds to the language specialisation design pattern.

To summarise the embedding pattern, the advantages are discussed below:

- Host language infrastructure that includes development and debugging environments i.e. editors, debuggers, tracers, profilers, etc. can be reused, which in turn would reduce project effort and costs.

- Project learning curves would be significantly lower compared to other approaches because DSL developers are most likely familiar with the host language.

- It often produces a more powerful language than other methods since many features are available for free.

Embedding pattern comes with its own challenges which include issues such as:

- Expressing DSL syntax using a user-defined abstract data-type written in the host language is not always easy making syntax optimisation hard to achieve.
- Error reporting and error handling is a challenging process in this pattern, as errors and corresponding messages generated for the host language are not easily presentable in a DSL [94].

Internal DSL Implementation Patterns

Internal DSLs patterns themselves can also be classified into different categories. For instance, Debasish divided internal DSLs into two different categories of *Embedded* and *Generative* DSLs [94].

- **Embedded DSL**: In this pattern, the DSL is surrounded entirely by the host language and DSL is embedded deeply inside the host language. Using this pattern, the DSL will be developed in the host language and represented to the DSL user as an entirely new language. There are a number of common approaches which can be used on this specific pattern namely:
  - **Smart API**: Using builders pattern [91], DSL can be developed by coding a series of smart or fluent APIs which can be glued together in a natural sequence. As an example, please look at listing 5.1, which provides an example of a smart API written in Java:

```
BigInt Interest = new Interest.Builder()
                          .forClient("TheClient")
                          .atDate(today())
                          .atInterestRate(0.5) ;
```

Listing 6.1: Smart API Example in Java

Although the code is readable and has a relatively good level of expressiveness, it utilises a great degree of unnecessary parentheses, dots and Java syntax, which may not be useful to the actual DSL user. In addition, using this pattern could lead to the development of lots of small

methods that may not have any use on their own. Besides, smart API that imposes unnecessary parentheses, dots and Java syntax to its users might not be the prime choice for describing complicated security policy scenarios that come with too many conditions and obligations.

  - **Reflective MetaProgramming**: Developing a DSL is not always as easy as the above example. There are scenarios that a DSL must take action based on the information received during runtime. By combination of decorative pattern [91] and writing the code in a GPL this would be achievable through Reflective MetaProgramming. In addition to that, using *Duck Typing* that is provided by languages such as, Groovy, object would be able to decide which method to invoke during runtime without any inheritance forced by languages like Java.
  - **Typed Embedding**: Through Meta-Programming, DSL coders must precisely write the interfaces based on the rules that the domain enforces. However, in more complex DSL, statically typed language such as Scala [127] could help developers. Using statically typed languages DSL coders would be able to define different characteristics of a domain as a*Type* and expose the users in an appropriate manner in order to model the domain.

- **Generative DSL Patterns**: Unlike Embedded DSL pattern, where the DSL is embedded deeply into the host language, through generative DSL pattern utilising a generic code and /or configuration files, a great portion of DSL code gets generated either through the compile time or runtime.

  - **Runtime Meta-Programming**. There are languages available which expose their runtime infrastructure as meta-objects to their users. Ruby and Groovy can be presented as an example of host languages with Runtime Meta-Programming capability.
  - **Compile-Time Meta-Programming**. Very similar to Runtime Meta-Programming by practising Compile-Time Meta-Programming approach,

a user interacts with the compiler during the compilation phase to generate the code. LISP is the pioneer in this category. Clojure is another JVM based language which has been invented in a manner that provides the same level of syntax that LISP provides, while it can easily integrate with Java and JVM based languages.

External DSL Implementation Patterns

Unlike internal DSLs, there are not many patterns available for external DSLs simply because everything must be implemented from scratch. Despite that, in an architectural respect regarding external DSLs, a DSL consists of a script and a black-box which is responsible for parsing the scripts and acting accordingly. The way that the DSL developers structure the black-box can identify the pattern they have used to implement the DSL. The main and most important component of the black box is the parser. This paper will explore the different types of parsers that can be utilised and integrated into a project.

Parsers : The first and the most important component of the external DSL is a parser. In order to develop a parser, two main activities need to be performed beforehand:

A) A context-free grammar for the DSL should be noted. There are two techniques which can be adopted to achieve this: Backus Normal Form [43] or Van Wijngaarden [108]. BNF grammar or any of its derivatives are used in almost all modern parsers. Van Wijngaarden was defined and used in the development of Algol68 [130].

B) Code the parser in accordance to the rules dictated by the grammar.

Unfortunately, writing a parser from scratch for a DSL is a time-consuming process as every single detail of the DSL grammar must be coded deeply into the host language. As a result, using a highly configurable parser like *ANTLR* [129]

and configuring it for a specific projects requirements is recommended by experts. Even though there is a list of parsers available to choose from, each comes with specific characteristics that must be known to the users before utilising them in their projects. A brief description of these characteristics has been listed here:

- **Top-Down Parsers**: These are the parsers that start parsing the tree of objects from the *top* and parse the branches and leaves by first parsing the leftmost derivation of the input stream. While the parser recursively traverses across the tree, it has two choices for processing the tree:

  A) To parse the tokens (leaves) *left to right* and to *Look-ahead* only one token. These parsers are called *LL(1)* parsers, which are usually simple and easy to use. The simplicity of the parsers would imply that they come with their very own issues.

  B) To parse the tokens (leaves) *Left to right* and to *Look-ahead* and fetch *k* tokens. These parsers are called *LL(k)* parsers. These are advanced parsers which can be used in a wide range of scenarios. These parsers can be extended to memorising parsers or predicated parsers. The ANTLR belongs to LL(k) Top-down parser group.

- **Bottom-Up Parsers**: As the name suggests, Bottom-Up Parsers start their processes from the leaves and end it on the root. An LR(k) parser is considered the most efficient bottom-up parser[94].

Utilising a parser, whether it is top-down or bottom-up is not the only task involved in implementing an external DSL, but choosing the right and appropriate parser for an external DSL is the most important decision that must be made. Considering this, it seems there is now enough information available regarding DSL implementation patterns to choose the most suitable one.

6.3.6 Internal or External DSL

When it comes to software design, no universally applicable choice is available. Each different approach can be justified individually, as each has its own advantages and disadvantages. The DSL embedding design and development is not an exception.

In the approach to designing the framework, the external DSL design pattern was chosen. This is justified as follows:

A) **Expressiveness**: Although the scenarios that are covered by security policy languages are limited, each can become extremely complicated. As a result, the DSL should be flexible enough to cope with such requirements. Taking that into account, internal DSLs may not be a good approach simply because exposing a limited level of the functionality of the host language to DSL users while covering a complicated scenario may not be easily achievable using internal DSLs. As mentioned, usually in such scenarios, the expressiveness of DSL is compromised.

B) **Lack of Codebase**: Internal DSLs are often used in scenarios where infrastructure and the knowledge base of a system exist; hence, the DSL can reuse the majority of the existing code and/or infrastructure. The DSLs that are usually written to engage domain experts to test the software in a DDD approach are one example. Although the security policy languages will play an important role in this framework, their codebase is not reused in the framework.

C) **Design Freedom**: In this framework, a new abstract language will be designed. Such a language would be a standard security policy language, which will work on top of the framework; thus, the language will need to be tailored in such a way that it is more appropriate to this project. Such a luxury is not available when internal DSLs are in use.

6.4 External DSL Implementation

In the previous section, DSLs were reviewed from different perspectives in detail, including the DSL concepts, advantages and disadvantage, different phases needed for DSL design, appropriate DSL implementation patterns in accordance to this project and the decision to use an external DSL pattern for the framework. In this section, various external DSL implementation patterns are discussed in detail.

6.4.1 Anatomy of External DSL

External DSL design and implementation lifecycle follows the exact lifecycle of a GPL. Similar to a GPL, an external DSL should receive textual input from the user, parse and tokenise the input and eventually process the tokenised input based on predefined business rules, which are known to the language. Figure 6.2 provides simple illustration of the steps taken by an external DSL.

Assuming that implementation of an external DSL is desired, the parser of DSL, which plays a significant role in an external DSL implementation, can be developed using a pattern-matching technique. In very simple scenarios, parser output can directly be fed into business rules and be processed without further manipulation; however, in a majority of cases and certainly in the context of this research, the parsed input must be modelled in a more formal way. Taking the similarity of GPLs and external DSLs into account, the same technique used by GPLs, which is *Abstract Syntax Tree (AST)*, is used to model the parsed input script.

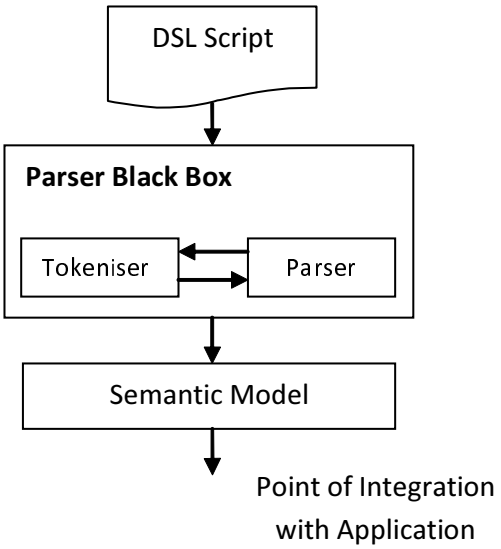


Figure 6.2: Architectural Overview of External DSL

An abstract syntax tree or just syntax tree, is a tree representation of the abstract syntactic structure of a source code written in a programming language. Each node of the tree denotes a construct occurring in the source code. The syntax is *abstract* and does not represent every detail appearing in the real syntax. The AST identifies the structural representation of the language in a form that is independent of the language syntax. Depending on the usage intention of AST, it is possible to use AST expansion, with additional information such as object types, annotations and other contextual notes, in the next stage of processing.

While working on an external DSL, enrichment of AST, which results in a semantic model for the domain, plays a vital role in DSL implementation. In reality, the semantic model is a data structure that is enhanced with domain semantics after the DSL input (i.e. DSL scripts) are processed through the pipeline. Its structure is independent of the DSL syntax and is more aligned to the application model of the system. The semantic model would be responsible for decoupling the input syntax-oriented scripting structure of the DSL from the target actions, as shown in Figure 6.3.

In addition to the above, a well-designed semantic model of DSL would result in a better testable DSL in its life-cycle.

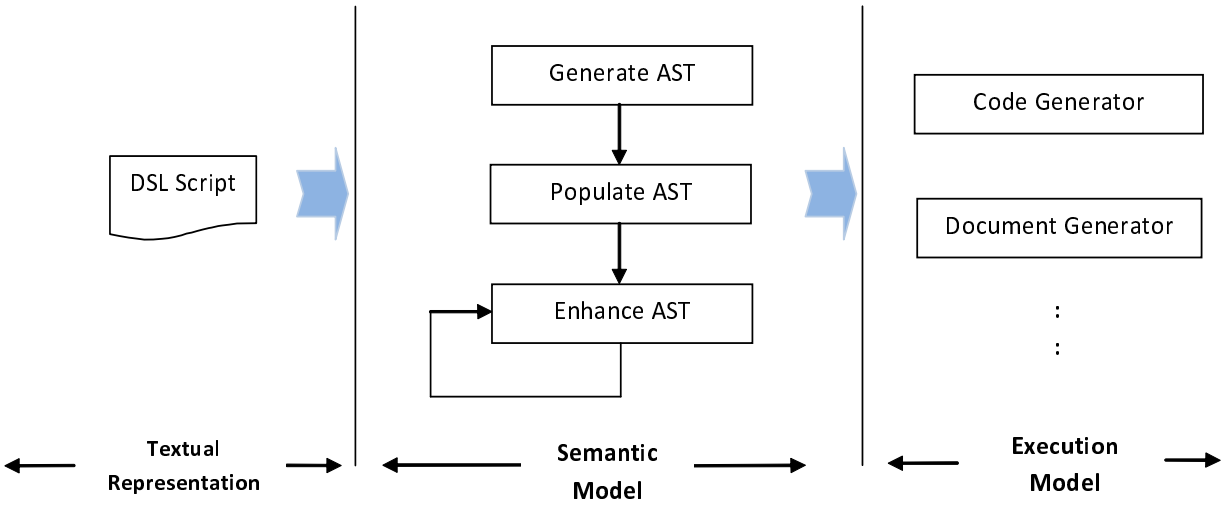


Figure 6.3: Boundaries of Semantic Model in External DSL

Semantic model, as described above, is not exclusive to external DSLs. In fact, both internal and external DSLs use a semantic model. In internal DSLs, the host language parser would be responsible for populating the semantic model, whereas in external DSLs, the developer would be responsible for providing the DSL with a parser to parse and process the DSL script and populate the semantic model [94].

6.4.2 External DSL Implementation Patterns in Details

Depending on the nature of the DSL and the parser that is used to design the DSL, the external DSL implementation could vary. In addition, different people categorise external DSL implementations using different approaches. For instance Fowler [87] categorises external DSL patterns based on the techniques which are used in implementation while Debishesh [94] prefers to categorise them based on their commonality. A brief description of these patterns is presented as follows:

- **Context-Driven String Manipulation**: In this approach, a simple DSL script, which cannot easily fit in any programming language (perhaps in order to use an internal DSL implementation, as opposed to an external DSL), is parsed by the parser and fed into an application depending on the type of the script (delimiter or syntax-directed translation). In the delimiter-directed approach, a delimiter, such as *end-of-line* or *end-of-file*, will be used to parse the script. Whereas when using a syntax-delimited approach, a grammar (hierarchical structure) with multiple levels of context must be defined beforehand, which must be agreed upon and approved by both DSL developers and DSL users.

  Almost always when context-driven approach is used, two distinctive levels of syntactical analysis on the DSL scripts are used, namely:

  A) **Lexer**, which is also called tokeniser or scanner and

  B) **Parser**, While lexer is responsible for splitting the input to the tokens, which represent more reasonable chunks of inputs, parser is responsible for parsing and providing more in-depth and detailed information off the DSL script.

- **DSL Workbench**: To recap, during external DSL implementation, population and enrichment of AST would be a developer's responsibility. Taking that into account, if there is already a system available, which is capable of maintaining the code in form of AST, that makes the development of external DSLs much easier and faster. In addition, using such system would enable easier transformation, manipulation and subsequent code generation of the populated AST. Such a system would be a *DSL Workbench*. Eclipse Xtext [28] and JetBrains [8] can be presented as examples of such systems which offer external DSL (or even GPL) development.

- **Parser Generators Using BNF and EBNF**: *Extended Backus-Naur Form (EBNF)*, which is simply an extended version of Backus-Naur Form (BNF) [43], is used to define the grammar of a programming language (including DSLs which typically come with strong grammars). BNF was invented to de-

scribe the Algol [132] language back in 1960s [87]. BNF grammars has been widely used since that time to describe the syntax of programming languages that adopt context-driven string manipulation.

DSL workbench, which was reviewed above, focuses on maintaining written code in AST form. However, there are tools available, that *generate parsers* in accordance with declaration, configuration and rule specified by the developer. These rules are defined using a syntax notation that is very similar to EBNF. Generally, rules declare the grammar of the language. YACC [105] and ANTLR [129] are two examples.

Unlike the previous two patterns where input scripts are processed after the parsing is finished, in parser generator pattern, certain actions can be defined to be embedded into the final production of the parser code. Such actions will be triggered when certain patterns are matched by the parser.

- **Parser Combinators**. Working with embedded DSL code in the parser and defining the grammar using EBNF type meta-programming cannot be categorised as the most appropriate way to implement external DSL codes.

  Modern languages provide their users with a simple yet powerful tool, which can be used to parse, express and define certain actions which must be taken in accordance with the parsed script. The beauty of such an approach is that while developers are in charge of generating the parser, they can simply perform the task by utilising the host language artifacts like classes, functions, methods etc. Not to mention, they can achieve this using their favourite development kit (as opposed to EBNF), which, in turn, make them more comfortable. Scala and C# both provide their users with parser combinators.

Certainly not every ASPL script that will be written for the framework are simple enough to fit into the Context-Driven String Manipulation pattern, hence this pattern cannot be used for the framework. In the next chapter, implementation starts by combining the Workbench and Parser generators using BNF and EBNF patterns. This will be explained in details.

6.5 Choosing a Programming Language

Irrespective of the DSL implementation approach, the development phase must be started by coding in one programming language. However, choosing the correct programming language was another challenge in this research. Even though most of the available programming languages are sufficiently mature enough to satisfy a developer in the beginning stages, the variety of these languages makes it difficult to choose the right one.

In the beginning, it was clear that an external DSL would be created from the outset. Such a decision had a direct impact on the available and shortlisted programming languages which are capable of providing the tools for external DSL development. Further research in this area revealed that the majority of modern *Java Virtual Machine (JVM)* based languages are currently capable of developing internal and external languages. However, .Net languages did not lose the battle and remained neck and neck with JVM-based languages. For instance, the Irony [133] framework is a parser generator framework for language implementation on the .Net platform that can be easily compared to parser generators, which are provided by other powerful JVM languages like Scala. Admittedly, in addition to the above criteria and facts, the knowledge base of the research team had an impact on the language selected and diverted it to the JVM based languages.

Deciding between JVM-based or .Net languages was the least of problems that the research had to overcome. A simple search shows that the majority of JVM-languages (that was chosen as the base of programming language) are capable of developing internal and external DSLs. Debasish, for instance, compares Clojure, JRuby (or better said Ruby), Java and Scala in [94]. So the research had to explore various aspects, such as, which programming language is faster, comes with less memory access footprint (to help the recursion) etc. As the result a number of non-Java JVM-based languages papers were reviewed [136] [137]. Due to the fast growth of these languages, most of these reports are out-dated. Out of those comparisons, the comparison provided by Wing Hang Li *et al.* [117] fits perfectly

within the context of this research.

They examined four non-Java JVM languages and used exploratory data analysis techniques [147] to investigate differences between these languages (in their dynamic behaviour) to Java. They analysed a variety of programmes and their behaviour to draw distinctions between the different programming languages. The languages they compared are widely chosen in DSL implementations and they are:

- **Clojure**: Clojure [96] is a LISP dialect, with support for advanced concurrency features, including actors and software transactional memory. It is a functional language with dynamic typing.
- **JRuby**: JRuby [18] is a Java-based implementation of the Ruby [144] programming language. It is a dynamic, object-oriented language.
- **Jython**: Jython [19] is an implementation of the Python language for the Java platform. It is a dynamic, object-oriented language.
- **Scala**: Scala [127] is a functional and object-oriented programming language, with a static typing discipline. It has advanced facilities for typing and concurrency.

Figure 6.4 shows the data analysis techniques they applied to gain an understanding of the dynamic behaviour of the various programmes and languages.

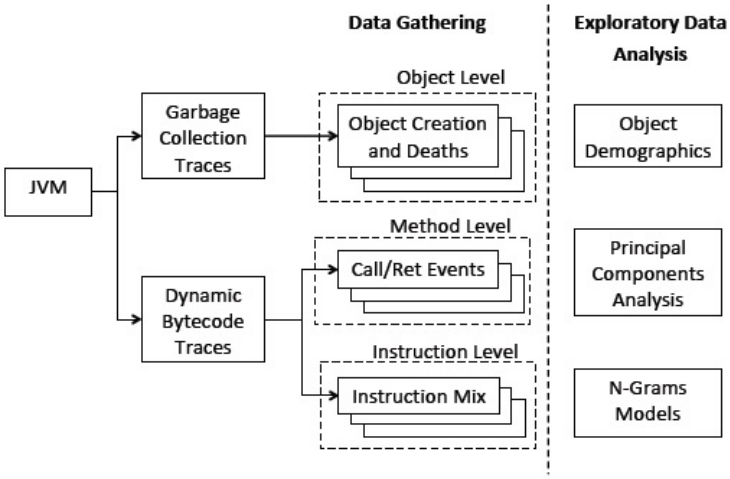


Figure 6.4: Schematic Diagram of Profiling Data Generation [117]

In this figure :

- **N-gram Models**: An N-gram is a sequence of N consecutive JVM byte-code instructions within a single basic block. The authors considered the coverage of observed N-grams in relation to the theoretical maximum.
- **Principal Component Analysis (PCA)**: Principal Component Analysis is a frequently used technique for dimension reduction, to aid visualisation and comprehension. PCA works by choosing and combining dimensions that contain the greatest variance. For each individual benchmark program, in their report, the authors measure the relative frequency of each JVM byte-code instruction to produce a 198-vector of values in the range [0-1] or a 39204-vector of values for 2-grams.

Boxplots, which is a convenient method of graphically depicting groups of numerical data through their quartiles [3], have been used to summarise the distributions of data for measurements on methods and objects. In addition, the authors have used Heat Maps to compare object lifetimes between the JVM languages.

By performing static analysis on the language libraries and dynamic analysis on a set of 75 benchmarks written in those languages and by using the above analysis approach, they compared these languages at three levels, namely: *instruction-level*, *method-level* and *object-level*, in details. They also provided users with their results. All of their results are provided in Appendix C.

Finally, by studying the obtained results and comparing them with the other related works, they concluded their results as follows:

- **Instruction Level**: At the instruction-level, non-Java benchmarks produce N-grams not found within the Java benchmarks, suggesting they do not share precisely the same instruction-level vocabulary as Java.
- **Method Level**: At the method-level, they have found Scala, in particular, has much smaller methods than the other JVM languages. Both Clojure and Scala applications exhibit deeper stack depths than other JVM language benchmarks.
- **Object Level**: At the object-level, lifetimes of non-Java JVM languages are generally shorter compared to Java. Especially on Clojure and Scala, object sizes are smaller compared to Java. Finally, Wing Hang Li observed that non-Java languages use more boxed primitive types than Java.

Although that there is no clear winner in this report, it seems that Clojure and Scala are better performers and more enhanced among JVM-based languages, according to the Wing Hang Li benchmarking. Considering this, these two languages were compared from different angles, demonstrating that Scala provides much richer classes compared to Clojure, which is really limited in many ways. For instance, Scala provides a wide range of arrays and primitives, but what Clojure provides is not even close. Most importantly, the speeds of these two languages almost touch the two extremes. While Scala runs as fast as Java, Cojure users always complain about its speed [4]. There is another fact that also remains, which turns the table in Scala's favour. Scala provides with a parser combinator, but this is not

the case for Clojure. In the context of this research, this is an important fact, as it may be necessary to design the external DSL based on the parser combinators. Due to this, and all the other facts provided above, it was decided to use Scala as the programming language.

6.6 Summary

6.6.1 Chapter Summary

In this chapter, different aspects of DSL design have been rigorously looked at. The chapter started with a definition of DSL and expanded that to DSLs structure, identified DSL stakeholders and its boundaries and continued the discussion with the advantages and disadvantages of DSLs.

In second half of the chapter, the different phases of DSL design and development were identified and reviewed step-by-step. The chapter concluded by carefully exploring different DSL implementations and choosing the one that fits best in the context of this research.

6.6.2 Research Contributions of the Chapter

Although Fowler refers to DSLs as *a new name for an old idea* [87], the number of resources available in this specific arena is really limited and there are no authoritative documents available to demonstrate how the different stages of DSL design and development could and should be applied to a real world scenario.

Throughout this research, as an additional contribution, it has been demonstrated how different patterns of design and development can be adapted for a real world DSL development scenario. In particular, in embedded DSL development, which is currently considered a favourite DSL development approach, the lack of a step-by-step guide to show which of the two flavours of this pattern (internal and

external DSL development) should be adopted, is noticeable. This chapter shows how the requirements of a project could come to the rescue, to answer the most difficult question of embedded DSL development.

In addition, while a wide range of mature, well-documented and well-supported programming languages with a spectrum of functionalities are available, there is no black and white guideline that exists to explain which programming language to choose for a DSL development. In this chapter, an easy method to illustrate how to shortlist and filter the programming languages for DSL development, was discussed.

Chapter 7

Implementation of the Framework

In this chapter, the research presents:

- The chosen methodology for the framework development,
- The framework requirements,
- The high level design of the framework,
- The framework iteration through low level deigns of the framework,
- Testing and evaluation of the framework,
- The research contributions of the chapter.

7.1 Software Methodology

The very first step that must be taken in almost all software development projects is to choose an appropriate software development methodology. There are a number of software methodologies that a project can utilise, the following list can be presented as a subset:

- Agile
- Extreme Programming
- Test driven Development
- Quick and Dirty
- KISS

Each of these methodologies comes with their own characteristics, advantages and disadvantages. Out of the reviewed software methodologies, Agile has been chosen for the development of the framework. The decision made can be justified as follows:

- **User Involvement**: The fundamental of the Agile development is to give the customer the highest priority. As a result, the customer is involved at almost every step of the software development process, which results in a more promising product at the end of the development phase. As the main goal of a DSL in mind, that is providing an abstract language to a client, Agile development would be a perfect match for DSL development [88].
- **Cost**: Cost is always an issue in the software development process. In addition, this imposes even more constraints on an academic project. Involvement of the user in the early stages of the project implies smaller iterative changes at early steps of the project. That prevents fundamental changes at the end of the project, which could lead to unacceptable costs.
- **Defect-Less Development**: Traditional software development provides a software to the client at the end of the software development process that must go through the evaluation process and discover system defects. However, user participation during Agile development suggests that the number of defects found at the end of the development phase would be minimal.
- **Shorter Development Cycle**: Involvement of the end user in the software development process also ensures an even faster software development process.

This becomes more visible when the system becomes ready for an overall test and evaluation by the client.

- **Size of the Project**: Agile development may not be a prime choice for a distributed, multi-site or large-scale development team (although there are techniques that can be applied to adapt even those projects with Agile development). The small size of the present research development team also suggests that Agile development could be considered the best software methodology that can be adopted.

7.2 System Requirements

So far, it has been mathematically proven that the implementation of the framework would work. In addition, an appropriate design and implementation method has been chosen. However, the implementation phase cannot be started before a set of detailed requirements for the system is defined. Such requirements could technically shape the implementation from the outset.

The requirements of the framework as the base for the system architecture should ideally combine the advantages of various DSL implementation approaches, to make it applicable to a large number of DSL scenarios. These properties (a number of them already reviewed in previous chapters) have been identified as follows. These properties will be used to evaluate the approach and the framework towards end of the project.

- **To Reuse Host Language Infrastructure**: The necessity of this property seems obvious and can be considered as a *must have* on a majority if not all embedded DSL implementations. The property includes but is not limited to syntax, semantics and tools available for the host language.

- **To Provide Well-Formed Measurements**: Providing a measurement tool to check the validity of DSL scripts against both syntactical (accordance to

grammar) and contextual (e.g. scoping rules) restrictions would guarantee a well-formed and valid policy as an outcome.

- **Modular Semantics**: Designing the DSL using modular semantics would make it possible to benefit from reusable components that can be composed with other semantics [102].

- **To Implement a High Performance System**: The property also can be considered as an obvious requirement, as this is a must have requirement for almost all computerised applications.

- **To Minimise Development Efforts**: It cannot be emphasised enough that the development of a DSL is an expensive task. Not to mention that it has been decided to develop an external DSL for the policy framework with a relatively more complicated development cycle compared to the internal DSL development. Hence, the development of reusable modules, with the aim of minimising efforts, is considered a vital requirement for the system.

  In addition, it would be cost-effective if a level of automation were integrated into the design. Model Driven Software Factories [70] (also known as Software Workbenches [85]), which are described as a tool for Model Driven Engineering, can be presented as an example. Software Model Driven Engineering, MDE, aims to raise the level of abstraction in program specification and increase automation in program development. The idea promoted by MDE is to use models at different levels of abstraction for developing systems, thereby raising the level of abstraction in program specification. An increase of automation in program development is reached using executable model transformations. Higher-level models are transformed into lower level models until the model can be made executable, using either code generation or model interpretation [72].

- **To Have Expandability**: Expandability is the most desired feature of almost all computerised systems and is usually requested by users. The framework of the current research is not an exception. Knowing the fact that the framework

will be prototyped for only candidate languages implies expandability of the system is almost inevitable. The design should allow other policy languages to be able to use and modify the system. More importantly, any expandability of the system, with regards to adding new security policy languages to the framework, most likely comes with a level of change in the grammar of the DSL. Hence the grammar of the DSL also must be designed with the expandability and flexibility in mind.

Although it may not be directly related to expandability of the system, the design of the framework would preferably need to follow the loose coupling approach. Loose coupling is an approach to interconnecting the components in a computer based system design so that those components, also called elements, depend on each other to the least extent practicable. The goal of a loose coupling architecture is to reduce the risk that a change made within one element will create unanticipated changes within other elements [135].

- **To Present User-Friendly User Interface**: The revelation of *smartphones* and *apps* in recent years has exponentially raised the expectation of ordinary users. Users often expect a clean and simple user interface which executes their desired action(s). Simplicity in the user interface must play a significant role in the design of framework.

- **Security of the System**: The other requirement that could impact the framework design is the fact that the final product must work on a secured domain. In other words, the framework, which sits next to the policy servers inside a controlled and secured domain, should be designed in a way that imposes the minimum security risk on the surrounding domains.

Having gone through the system requirements implies that the other stages of implementation can be started. *High Level Design (HLD)* (Architecture) would be the one to start with and then the discussion can be expanded to *Low Level Design (LLD)*.

7.3 High Level Design

A HLD provides an overview of a solution, platform, system, product, service or process. Such an overview is important in a development project to ensure that each supporting component design will be compatible with its neighbouring components' designs.

The HLD also briefly describes all platforms, systems, products, services and processes that it depends upon and includes any important changes that need to be made. Typically HLD document would also include an *High Level Architecture (HLA)* diagram visualising the components, interfaces and networks that need to be further specified or developed [14].

With the HLD in mind, the HLA of the system can be proposed as detailed in Figure 7.1. In this Figure, there are several components that are interacting with each other, a list of these documents, their responsibilities and brief descriptions are listed in the next section.

7.3.1 High Level Architecture of the Framework

One of the core requirements of the system is to minimise development efforts. As a result, it has been decided to build the framework by utilising MDA. The discussion on the system components continues by briefly reviewing the components that have been used in the proposed architecture.

Eclipse

Initially originated from IBM VisualAge [36] code-base, Eclipse, which is an open source *Integrated Development Environment (IDE)*, has been used during the software development phase of the current research. It contains a base workspace and an extendable plug-in system for customising the environment. Written mostly in Java, Eclipse can be used to develop Java applications. Eclipse may also be used to

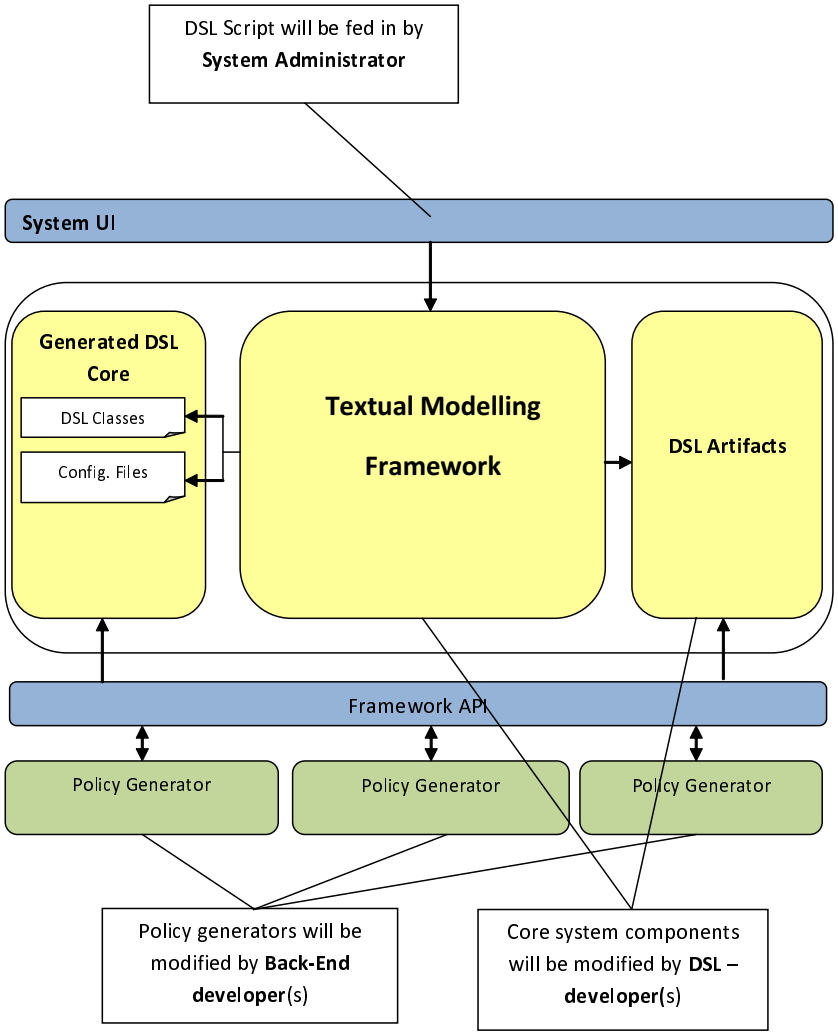


Figure 7.1: The Proposed HLA of the Framework

develop applications in other programming languages by utilising necessary plug-ins.

The Eclipse *Software Development Kit (SDK)*, which includes the Java development tools, is meant to be for Java developers; however, users can extend its abilities by installing plug-ins written for the Eclipse platform, such as development toolkits for other programming languages and can write and contribute their own plug-in modules.

Eclipse employs plug-ins in order to provide all of its functionality on top of (and including) the rich client platform. This plug-in mechanism is a lightweight software framework. The plug-in architecture supports writing any desired extension to the environment. The key to the seamless integration of tools with Eclipse is the plug-in. With the exception of a small run-time kernel, everything in Eclipse is a plug-in [11].

EMF / EMF(Core)

The *Eclipse Modelling Framework (EMF)* project is a modelling framework and code generation facility for building tools and other applications based on a structured data model. From a model specification described in XML, EMF provides tools and runtime support to produce a set of Java classes for the model, along with a set of adaptor classes that enable viewing and command-based editing of the model and a basic editor.

EMF consists of three fundamental pieces:

- **EMF(Core)**: The core EMF framework includes a meta model, which is called ECore and is used for describing models and providing run-time support for the models. That includes change notification, persistence support with default XMI serialisation and a very efficient reflective API for manipulating EMF objects.
- **EMF(Edit)**: The EMF (Edit) framework includes generic reusable classes

for building editors for EMF models.

- **EMF(Codegen)**: The EMF code generation facility is capable of generating everything needed to build a complete editor for an EMF model. It includes a user interface from which generation options can be specified and generators can be invoked. The generation facility takes advantage of the *Java Development Tooling (JDT)* component of Eclipse [10].

Xtext

While external DSL implementation patterns, to be more specific, DSL benchmarking, were reviewed, Xtext was briefly introduced. Due to the ease of use and flexibility of Xtext, it is one of the most used products developed and provided by openArchitectureWare (oAW), which is a leading provider of tools utilising MDA approach. Xtext is heavily dependent on EMFCore.

In summary, Xtext is a framework for the development of programming languages. It covers all aspects of a complete language infrastructure, from parsers, over linker, compiler, or interpreter to fully-blown IDE integration. It comes with good defaults for all these aspects, and at the same time, every single aspect can be tailored to the users' needs. At the core of Xtext, there is a workflow engine allowing the definition of generator/transformation workflows. A number of pre-built workflow components can be used for reading and instantiating models, checking them for constraint violations, transforming them into other models and then finally, for generating code [28].

In conclusion, based on an EBNF-like notation, Xtext generates the following artifacts:

- A parser that can read the textual syntax and returns an EMF-based AST (model).
- A set of AST classes represented as an EMF-based meta-model.
- A number of helper artifacts to embed the parser in an oAW workflow.

- An Eclipse editor that provides syntax highlighting, code completion, code folding, a configurable outline view and static error checking for the given syntax [77].

Xpand

Xpand is another subject that was also briefly reviewed while reviewing code generators in the previous chapter, under the DSL benchmarking section. Summed up briefly, in an MDA approach with the aim of consistency across the domain, at some point, the populated model must be transformed and synced with different project-specific artifacts like source code. This would be the responsibility of the generators. In addition, refactoring, specialisation and annotation also can be done using generators.

Since Xtext populates an AST based meta-model, a generator to produce codes on-the-fly and easily integrate with Xtext was needed. The other criteria that had to be considered was that Scala was chosen as the programming language, therefore, the generator should be capable of generating Scala code on-the-fly.

There was only one code generator option available to choose from: Oitok. Oitok was the only known generator that was capable of generating Scala code by transforming an AST-based model. Unfortunately, integration of Oitok into Xtext was not easy and resulted in many problems. It was then decided to replace it with another generator. Since using a non-Scala based code generator was the only choice available, it was decided to choose a Java-based generator with the aim to integrate the generated Java code to Scala-based framework project. As the result, it has been decided to choose a well-known generator that comes with a good reputation amongst developers; Xpand.

Xpand is a statically-typed template language featuring the following:

- Polymorphic template invocation.
- Aspect oriented programming.

- Functional extensions.
- A flexible type system abstraction.
- Model transformation.
- Model validation.

It includes an editor, which provides features that are handy when it comes to DSL development. A short list of these features are as follows:

- Syntax colouring.
- Error highlighting.
- Navigation.
- Refactoring.
- Code completion.

Similar to Xtext, Xpand was also developed as part of the OpenArchitectureWare project, before it became a component under Eclipse [9].

Dependency Injection

By definition, *Dependency Injection (DI)* is a software design pattern in which one or more dependencies are injected into a dependent object and are made part of the client's state. The pattern separates the creation of a client's dependencies from its own behaviour, which allows program designs to be loosely coupled [86].

The main advantage of DI is loosely coupling the various parts of the application to each other. In the context of this research, there are numbers of places where utilising such a functionality would become useful.

- Dependency injection allows a client to remove all knowledge of a concrete implementation that it needs to use. This specific characteristic of DI helps different parts of the framework to remain isolated and protected from the impact of design changes. This indeed, in turn, meets the modularity and reusability requirements of the system.

- Dependency injection can be used to externalise a system's configuration details into configuration files. Knowing the fact that different code generators would eventually use the system, each of which come with their own configuration, this specific feature of DI would help individual configurations for different security policy language implementation to be written independently.

7.3.2 HLA Components

The HLA components can be described in details as follows. Starting from the top, these components can be seen in the diagram:

A) **System Administrator**: Refers to the person responsible for reading the security policy written by the security officer (please refer to Section 2.1) and translating it to a security policy written in the abstract language known as DSL. The administrator would need to interact with the system. This could be done via an editor, IDE or even a browser.

B) **System UI**: A system UI is a user interface that allows a system administrator to communicate with the system. The UI could utilise a variety of products from a simple text editor to a sophisticated IDE running on an smartphone.

C) **Concrete DSL**: This represents the concrete semantic model of the DSL. Typically these are the DSL objects/classes that textually model the DSL (in terms of programming). Policy generators will use the model during the next stages of DSL transformation.

D) **DSL Properties**: These represent the DSL specific configuration files that shape the DSL, such as DSL grammar and DSL configuration files. These properties are used by the framework to manage the DSL during runtime.

E) **Framework API**: In order to loosely couple the language specific part of the policy framework from the core model of the framework, this component has been used. The API is used to connect the external policy generators/UI to the DSL framework.

F) **Policy Generators**: Policy generators are stand-alone pieces of code that are responsible for generating security policy code (script), written in specific security policy languages. These codes are generated based on the concrete DSL (i.e. the parsed model of the DSL) in accordance of the rules imposed and enforced by DSL meta-models (configurations). Output of these units will be fed directly to security policy servers i.e. PAP (cf. Section 2.3 for more details)

G) **DSL Developer**: This refers to developers who are responsible for maintaining the DSL framework as an independent, abstract code that is capable of parsing the DSL scripts and semantically modelling the DSL in accordance with the enforced rules and grammar.

H) **Back-end Developer**: This represents those developers who are responsible for producing policy generators that are capable of analysing the modelled DSL (based on the DSL concretes) and generating the language specific security policy code. A DSL developer and back-end developer must work in collaboration, as sometimes plugging more security policy languages into the framework implies manipulating the DSL grammar and meta-model accordingly. As an example, assume that a new policy is added to the framework that provides a feature, which is not offered by other policies. If that is the case, then the DSL grammar must be modified in a way that incorporates the new functionality into the system, to be used by the newly added policy language. This would require collaboration between the back-end developer and DSL developer.

I) **TMF**: *Textual Modelling Framework (TMF)* framework represents a black box

that reads the input DSL scripts and provide users with a model. The black box consists of the following internal components:

- **Meta-Model Abstraction Layer**: This represents a meta-model of a parsed DSL script based on the rules dictated by the DSL properties. Such a meta-model will be used by other parts of the TMF (such as Model to text (M2T)) to produce a fine-grained model.
- **Parser Generators**: This is a parser that reads the textual representation of the model and instantiates the corresponding final model.
- **AST**: This stands as the representative of the final model, which is described as the output of the TMF framework.

7.4 Low Level Design

By utilising Xtext, a low level architecture of the framework can be presented as in Figure 7.2.

Starting from the TMF framework, which is in the middle of the framework going outward, the following components can be identified:

1) **TMF Framework**: Xtext has been used as the TMF framework in this architecture. Knowing the fact that it is heavily dependent on Eclipse, the following components have been delivered to the project without any cost, as the result of Xtext usage. This satisfies the requirement of the framework, which necessitate minimal development costs and efforts.

   a) **System UI**: A feature-rich Eclipse editor (based on the Eclipse text editor infrastructure) that is aware of the concrete syntax specified. The editor supports syntax highlighting, code completion, navigation (hyperlinks), hovers, folding, outline, and other features known from Eclipse text editors.

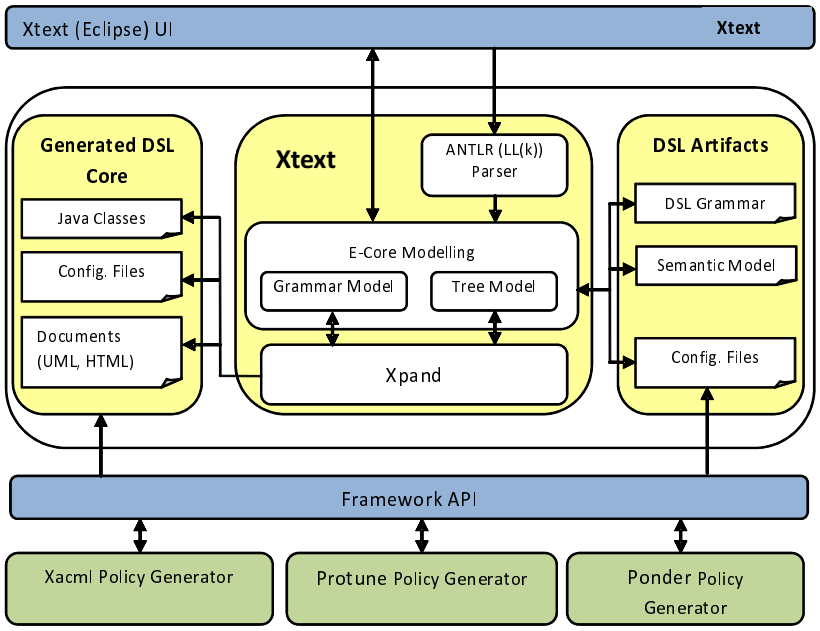


Figure 7.2: The Proposed Low Level Architecture of the Framework

b) **Parser Generator**: Xtext provides users with an ANTLR-based parser generator, which reads the input scripts and parses it based on the DSL properties and grammar and provides users with the corresponding model.

c) **Abstract Syntax Tree**: As previously mentioned, Xtext utilises an ECore-based AST. To recap, ECore is one of the main three components of the Eclipse Modelling Framework.

d) **M2T Transformer**: Initially, the Oitok framework was used as code generator (Transformer) because it produced Scala-based textual representatives of the model. Unfortunately, due to the many issues encountered during incorporating Oitok into Xtext, it was decided to replace it with Xpand. Xpand produces Java classes, helpers etc. by traversing through the ECore model.

e) **Xtext**: Xtext sits in the middle of the architecture and plays a significant role in design. It glues the different components of the architecture together. Briefly put, when using Xtext, the language rules (or in case of this research, the DSL grammar rules) first need to be defined in EBNF syntax. Xtext then uses that and utilises ANTLR to build an ECore semantic AST model out of

the input DSL script.

2) **DSL Properties**: DSL properties in this architecture consist of two parts:

   a) **DSL Grammar**: Xtext uses an EBNF notation style for the representation of DSL grammar.

   b) **DSL Configuration**: There are two different types of configurations available in this architecture, global and localised configurations. Global configurations are properties that are valid across the project, for instance, the output directory of the language specific policies generated by the framework. Localised configurations, however, focus on each policy generators and their specific configurations. The naming conventions used in specific security policy language generators is an example.

   c) **Dependency Injection**: DI plays a significant role in the design of enterprise applications to make different parts of the application independent, yet configurable from a single point of view. In order to deliver a configuration to each individual policy generator code, the Spring *Inversion of Control (IOC)* (equivalent to DI) [25] has been used.

   The Spring framework provides many modules and its core has IOC container. Spring's IOC container is light-weight and it manages the dependency between objects using configurations. Based on the configuration manged by users, Spring IOC links the related objects together, instantiates and supplies them to the code.

7.4.1 Design Review Round 1

During the design and implementation phase, a *User Centric Design (UCD)* approach has been adopted. The user centric design is a process that is mainly focused on the user interface design of the system, which pays extensive attention to user's needs, wants and limitations of the end-users product, service or process at each stage of the design process. The UCD may not be fully restricted to interfaces or technologies used to develop the end user product [121].

In order to obtain end user's feedback, that includes system administrator and DSL developer (please refer to Section 6.3.2.), in a more realistic and enhanced way, PoC approach have been adopted through the implementation phase. A PoC is a realisation of a certain method or idea to demonstrate its feasibility, whose purpose is to verify that some concept has the potential for use. A PoC is usually small and may or may not be complete.

Considering this, a PoC version of the design has been implemented. The product implemented the design, which was presented in a previous section and was made available to system administrators with the aim of obtaining their feedback. Also, the very same PoC has been shared among DSL-developers and other independent developers, in order to capture their points of view as well. The PoC, as described above, is also used to demonstrate that the chosen technologies and products, which are closely tied together, are capable of communicating with each other in order to produce the final result.

7.4.2 Capturing Feedback

System administrators, end users and DSL developers' feedback have been captured through a series of interviews, observation of behaviour and a simple questionnaire. The results of all these activities are detailed below.

A) **End User's Point of View**

Users had three major issues with the design provided, which are listed below:

1. **Security Concerns**: The low level design that utilises textual to model framework was heavily dependent on the Eclipse platform. That implied that two individual components had to be installed on the end users' computers in order to make the framework functional. These two component were the *Java development kit (JDK)* and the Eclipse platform.

   Although installing these two components take a fraction of an hour and seemed a minor change, this raised a major concern to the end user. It

should be noted that the framework's end users are security administrators of secured domains and their computer systems are sitting behind a layer of tight protections. Although none of the above products (i.e. Java and the Eclipse platform) considered harmful software, regardless of type and vendor of a software product, installing the products in a secured domain raises questions and will not be considered as trivial exercise.

2. **Usability**: Dependency of the interoperability framework was under question, as each and every system administrator computers that wanted to use the framework also had to install these two products (i.e., the JDK and the Eclipse platform). Although this could be considered an acceptable approach, because only certain people would be able to access the security policy servers. However, changing system administrator computers, perhaps due to a fault or an upgrade, implies that the above mentioned software needed to be installed again on those computers. Otherwise, system administrators could not manage the security servers through the framework.

3. **Look and Feel**: While concerns that arose above were correct and reasonable, the main reason that the design was rejected by the end user was the look and feel of the framework and to be more precise, the Eclipse IDE. The Eclipse IDE is designed for development and comes with too many features, which are all useful to main user of the IDE (i.e., developers and programmers). However, that is not the case for the end users of the framework, who are non-technical individuals (in terms of programming and using IDE). The look and feel of the Eclipse IDE was not described as user-friendly by the users; therefore, they rejected the design concept. Figure 7.3 shows a snapshot of the Eclipse IDE, which was generated by the Xtext framework.

B) **Developer's Point of View**

The PoC was also shared with developers. Generally, the feedback received from experienced developers was positive, but they also questioned the user-friendliness of the architecture. Xtext uses EBNF style grammar notation, which dictates the overall grammar of the DSL. The developers did not welcome using

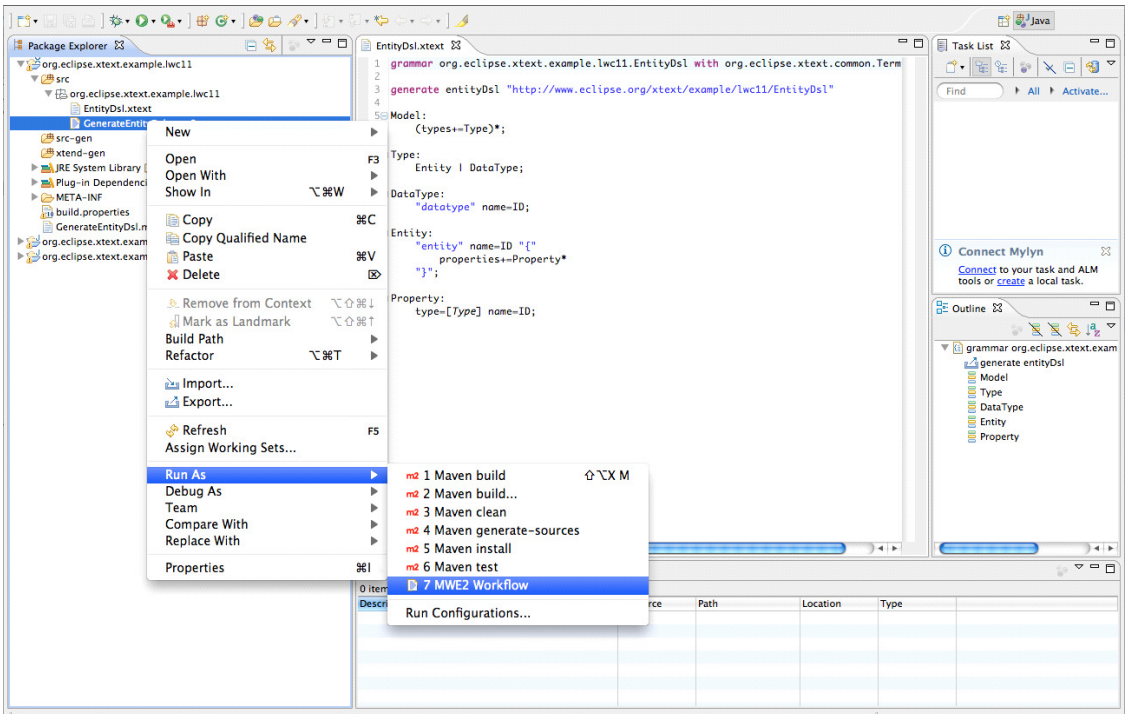


Figure 7.3: A Snapshot of Xtext (Eclipse) Environment

such an abstract language in the design (Language Cacophony). The reason can be presented as:

- The EBNF is effectively another language. The DSL developers should learn and understand how to code EBNF, before they can maintain the DSL through its evolution. The framework should be able to accept more and more security policy languages as time goes by, hence the EBNF rules, which shape the DSL, are subject to change and modification. That implies that the DSL developers must know the EBNF abstract language inside out. The learning curve would definitely have an effect on the timeline of a project. In addition, it increases the costs of a project as the result of extended timelines.
- Generally, developers are always open to understanding and learning new technologies and languages. However, that may not be the case for EBNF, as EBNF is an abstract language, with unique requirements. It is not comparable to a general-purpose programming language. The projects, sce-

narios and occasions that such an abstract language can be used are only a handful, hence, there would be no motivation for developers to learn the language.

C) **Overall Expandability of the System**

In addition to using Xtext, which has been built around EBNF, there are few other limitations:

- Initially Xtext was designed to accommodate the development of simple languages and DSLs. Although the Xtext framework itself has been enhanced over the time, the above characteristic was passed down to the new versions of Xtext. Hence, it only should be considered for development of simple DSLs.
- The EBNF itself is inadequate for defining complex forms of grammar. In other words, it cannot easily be used to define and enforce complex scenarios. That could cause a major impact on the framework presented by this research, when it comes to defining and producing complex security policies [30]. Not to mention, the EBNF is a linear language and that may not be a prime choice for describing security policy languages.

Due to the limitations and feedback received from end users and experienced developers, it was decided to enhance the design by dropping Xtext and using an alternative solution in the way that the following two requirements remain intact:

A) The architecture of the framework should not change fundamentally and

B) The overall development costs and efforts of the framework should not increase significantly.

Figure 7.4 visualises the issues raised by users, based on the framework non-functional requirements defined in Chapter 3. A snapshot of the questioner that have been used to capture participants' opinion on the provided PoC is provided in Appendix E.

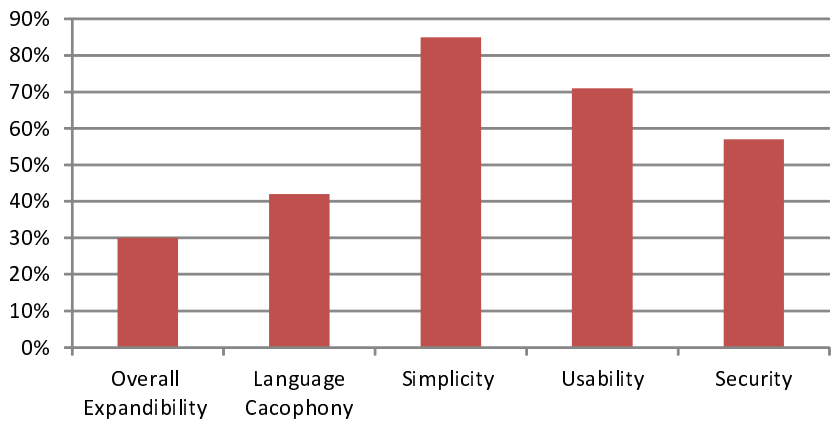


Figure 7.4: Percentage of Complaints/Issues Grouped by Framework Requirements

7.5 Restructure of the Design

Using Xtext in the middle of the design has helped to achieve a few system requirements as explained in the previous section, namely, reducing the cost by applying an MDA design, reducing development efforts and reusing the host language infrastructure. However, using Xtext imposes a level of restriction and discomfort to the end user of the framework. As a result, it was decided to use other alternatives to replace components offered by Xtext.

Preferably, the alternative solution should also provide the two advantages of Xtext, a low cost, low effort solution. Due to the removal of Xtext, the majority of its components had to be rewritten from scratch. Although that could be considered as a disadvantage of the new design, it could also add to the overall flexibility of the framework. Such an approach would put the developers in control of the code to the greatest extent. Rewriting these components implies the code would be more maintainable and expandable, compared to the architecture that utilised Xtext.

7.5.1 Parser Combinators

The first component (which was replaced) was the parser. Scala, as a host language, provides a unique functionality called the *Parser Combinator*, which was explained briefly in the previous chapter. In functional programming, a parser combinator is a higher-order function, which accepts several parsers as input and returns a new parser as output. It follows the *Pipe-Line* design pattern, as described in detail in Section 5.3.3. A parser combinator provides users with a level of comfort that includes easy construction, readable code, a modular approach, a well-structured parser and an easily maintainable project. They have been used extensively in the prototyping of compilers and processors for DSLs [94].

The history of parser combinators goes back to late 1980s, when R. Frost and J. Launchbury demonstrated the use of a parser combinator to construct a natural language interpreter in 1989 [59]. They have been in use since then and most recently, Frost, Hafiz and Callaghan described a set of parser combinators in Haskell to solve the long-standing problem of accommodating left recursion [90]. In Scala, parsers are implemented as monads, hence defining combinators for parsers is just monadic transformations, alternation or any other composition operations [93].

Using parser combinators facilitated meeting a few of the design requirements: there was a reduction in the effort and cost of the project and reuse of the host language infrastructure. Using a parser generator has other advantages that are detailed below.

- Developing the framework's parser using combinators is easy, well documented and almost comparable with Xtext with regards to cost and development efforts.
- There are abstract data types available in Scala to generates AST using parser combinators.
- It comes with a great pattern-matching engine, which enables dealing with complex security policies [93].

- Unlike Xtext, using a combinator does not require any extra languages to be taught to the developers. In fact, they would be able to pick up the concept easily, because the final product of the parser combinator would be in Scala code.

In addition to the Parser Combinator that replaced the ANTLR-based Xtext parser in the architecture, the following components were enhanced, replaced or added to the HLD:

- **Semantic Model**: The other free component that had to be replaced, as the result of abandoning Xtext, was the semantic model which comes with it. Usually populating the AST-based model by Xtext happens beyond user's control. However, in the new system architecture, that luxury vanished and everything had to be rewritten from the ground up.

  As a result, a customised AST tree in Scala was written and joined with the newly written and configured parser combinators, so that the AST could be populated on-the-fly by the parser. The AST has been revolutionised throughout the development process in order to accommodate more branches and present a more realistic view of the parsed tree.

- **User Interface**: Using Xtext implies that the user would be able to use an Eclipse editor free of charge. Although that could be considered a huge benefit for using Xtext, in the case of the current research, the complexity of the IDE increased the end user's discomfort; therefore, in the new design, it was decided to use an alternative solution that should preferably address a few issues raised by the Eclipse editor:

  A) The replacement solution should be simple and fit for purpose.

  B) It should add to the usability of the framework.

  C) The usability should not compromise the performance of the system.

  In response to these requirements, it was decided that a browser would be used in order to interact with the framework. Using http for communication

with the framework would definitely increase the usability of the system and would meet the other requirements, which are listed above. However, using the browser implies that an embedded browser-based editor must be incorporated into the system. A number of browser embedded editors were reviewed, including CodeMirror [5], Ace [15], CodePress [120], EditArea [12] and Ymacs [1]. After reviewing these editors in detail and considering the functionalities provided, it was decided that Ace, a JavaScript editor, would be an editor fit for the purpose.

Ace is a standalone JavaScript code editor that is specifically designed to work as browser-based code editor that matches and extends the features, usability and performance of existing native editors, such as TextMate, Vim or Eclipse. Ace can be easily embedded in any webpage and JavaScript-based applications.

Features and advantages of Ace editor are:

- Syntax highlighting.
- Automatic indent and out-dent.
- An optional command line.
- Can handle huge documents.
- Fully customisable key bindings including VI and Emacs modes.
- Provided with different themes to choose from.

- **Syntax Provider and Validation Engine**: The other two components that have been written from the scratch are the syntax provider and the validation engine. The syntax provider is in charge of defining the DSL syntax. The DSL syntax orchestrates the parser combinator's behaviour. In addition, it was used as an input to the validation engine.

  The validation engine has two distinctive responsibilities. It communicates with syntax provider on both of the following scenarios:

A) The validation engine will be used to validate and approve the parsed DSL script in accordance with the defined grammar rules provided by syntax provider and present users with appropriate message(s) should the validation fail.

B) It will also be invoked by the UI to provide users with syntax errors and or automatic code completion, while users are typing DSL scripts within the user interface.

- **Dependency Injection**: We reviewed the Spring IOC previously. There are contradictory views available on Spring IOC. Some believe that Spring IOC is a fantastic piece of code, which helps readability, maintainability, and expandability of the system. On the other hand, some believe the opposite. However, a majority of these two sets of reviewers believe that Spring IOC is a heavyweight tool. Such a tool with a multitude of features, functionalities and abilities is a *must have* component for enterprise and multi-level applications, but for standalone applications, Spring IOC could be considered a tool that over-engineers the whole design.

  By removing Xtext, the framework became a pure Scala project and after reviewing the Spring IOC, a different DI for Scala applications was chosen, namely, Cake.

  There are a number of ways to perform DI in Scala without adding a framework. The cake pattern is one popular approach. This pattern was first explained in Odersky's paper, Scalable Component Abstractions [128], as the way he and his team structured the Scala compiler. It works based on Scala traits [127] that are very similar to the interfaces in Java, but unlike Java, Scala allows partial implementation of the traits.

Figure 7.4 shows the enhanced architecture of the framework, as detailed above.

There's one last important note before the discussion of the design enhancement is ended. It was decided that a browser-based editor approach would be used on this framework. That implies that the requests coming from the user must be captured

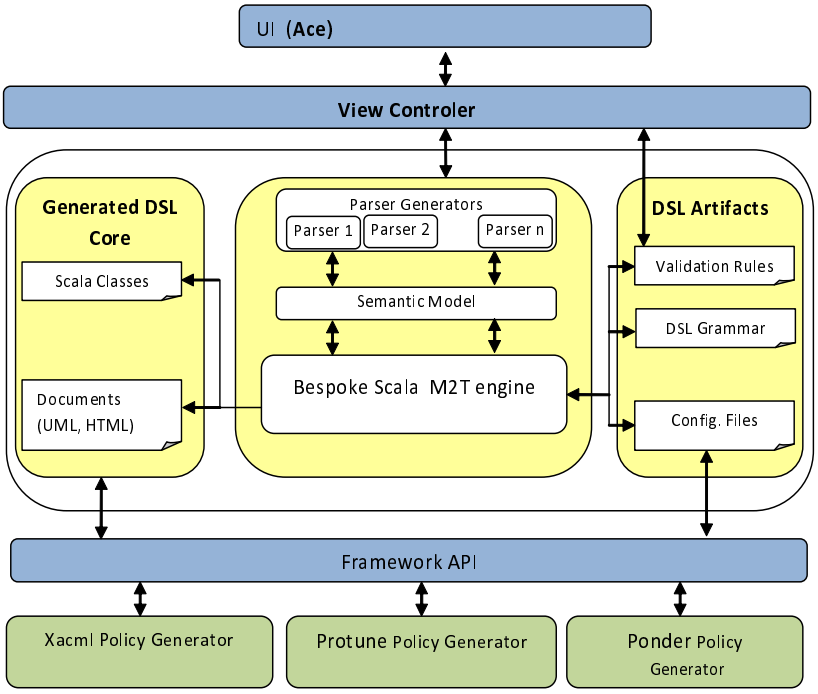


Figure 7.5: The Restructured Architecture of the Framework

and passed to the framework. In web based application architecture, that would be the responsibility of a webserver. As a result, PlayFramework [16], a Scala-based web server was installed and surrounded the framework in order to achieve the goal; however, installing, tuning and maintaining the web server was considered out of the research's scope.

7.6 Detailed Design

So far, a few concepts which have attracted developers in recent years have been reviewed in detail, namely, Domain Driven Design and Domain Specific Language. The other concepts that will be discussed in this chapter are: *Test Driven Development (TDD)* and *Behaviour Driven Development (BDD)*.

In a nutshell, test driven development is a software development process that relies on the repetition of a very short development cycle i.e. first the developer writes an automated test case that defines and tests a desired output of a new function, then

the developer produces the minimum amount of code to pass that test and finally, refactors the new code to the acceptable standards.

BDD, however, in software engineering, is a software development process that emerged from test-driven development. BDD combines the general techniques and principles of TDD with ideas from domain-driven design and object-oriented analysis and design, to provide software development teams with shared tools and a shared process to collaborate on software development [2]. BDD is based on principles of Hoare logic [101] .

BDD developments starts by defining the *behaviour* of a function that has to be written. In principal, this can be achieved by a *story*. The story should clearly and precisely answer the following three questions:

- Who is the main stakeholder of this specific story?
- Which output does the stakeholder want from the story?
- What business value will the stakeholder gain from the story output?

The story also should describes the condition(s) (including event triggers) that expected to be true when the story is commenced. Describing more stories using BDD approach would gradually results a *Ubiquitous Language* which is shared amongst different parties who are participating in software development (Please refer to Section 6.3.2 that describes DDD in details). The listing 7.1 describes one single scenario (story) which is described in BDD.

```
AS A: System Administrator.
I WANT TO: Add time constraint to the system.
In ORDER TO: Restrict user access to the system outside
working hours.

GIVEN: A system that is secured by a security policy,
AND: A user who is allowed to use the system,
WHEN: His/her access to the system should be restricted to
the working hours by the policy,
THEN: Any violation of the policy should be logged into the
security log-file.
```

Listing 7.1: Example of BDD Story

The example provided in listing 7.1 provides stakeholder of the story, desired output of the story and its business value to the environment. Using the ubiquitous language, the above story is modelled and the following DSL script is derived from it.

```
Protect
        Target "mySystem"
    for executing
        Actions "access"
    from
        Subjects "systemUser"
    under following
        Subject Conditions "8:00 < time < 17:30"
    do
        Obligation Action "LogViolation"
```

Listing 7.2: Example of DSL Script

Listing 7.2 shows the DSL script which has been generated from the story that

is described in BDD. In the next step, the parser combinator which is responsible for parsing and populating the semantic model (AST) must be developed. Listing 7.3 and 7.4 show part of the parser combinator which reads and parses the above script and the AST model of the script, respectively.

```
lazy val policyType : Parser[PolicyType]=
"Protect" ^^^ Protect | "Allow" ^^^ Allow

lazy val targets : Parser[Targets] =
  "Targets" ~> rep1sep(target, "AND") ^^ Targets

lazy val target : Parser[Target] = stringLit ^^ Target

lazy val subjects : Parser[Subjects] =
  "from" ~> "Subjects" ~> rep1sep(subject, "AND") ^^ Subjects

lazy val subject : Parser[Subject] = stringLit ^^ Subject

lazy val actions : Parser[Actions] =
  "for" ~> "executing" ~> "Actions" ~> rep1sep(action, "AND")
  ^^ Actions

lazy val action : Parser[Action] = stringLit ^^ Action
```

Listing 7.3: Parser Combinator Written in Scala

By adopting the procedure detailed above and through an iterative agile approach, the framework has been gradually taught different features and functions provided by different security policy languages candidates.

7.7 Enhancing the Framework

There were two features requested by the users, which have been accommodated in the high level design, but their implementation was considered outside the scope of the project; both have been categorised for *further work*. These two features are described below:

```
trait BaseTarget
case class Target(identifier : String) extends BaseTarget
case class Targets (list: Seq[Target])

trait BaseAction
case class Action(identifier : String)extends BaseAction
case class Actions (list: Seq[Action])

case class Condition(lineCondition : String)
case class ConditionList (colList: Seq[Condition])

trait BaseObligation
case class Obligation(identifier : String) extends
BaseObligation
case class Obligations (list: Seq[Obligation])

case class Policy ( policyType : PolicyType,
                    targets : Targets,
                    subjects : Subjects,
                    conList : ConditionList,
                    actions : Actions,
                    obligations : Obligations
                    )
```

Listing 7.4: AST Model Written in Scala

7.7.1 Limitation of Access to the System

In order to improve usability, it was decided that a browser-based approach would be used to interact with the framework. This addresses the usability concern, which was raised by users when they reviewed the framework; however, it creates another level of threat to the framework: *Unauthorised Access*. This can be categorised as an *internal threat* (please refer to Section 2.1). Using the browser-based approach, an unauthorised user inside the secured domain can access the system. Although this level of threat is considered internal and it has a lower risk compared to external threats, yet the issue must be resolved. There are a number of ways that unauthorised access can be prevented. The methods are categorised by their level of impact on the provided solution as follows:

A) Adding a Database to the System: Perhaps adding a database to the system would be the simplest yet most effective way to secure the framework against unauthorised access, without over-engineering the entire architecture. By adding a database, which could be an open source database like MySQL, the users of the framework will be provided with a username and password. User credentials will then be encrypted using an encryption algorithm appropriate to the database. Encryption of the user credentials in the database would protect the framework from internal threats, in case authorised access to the database is compromised. That also would add to the overall security of the system.

   An advantage of such an approach is the simplicity of the solution. A disadvantage, however, is that a new component must be added to the architecture. Although such a component (i.e. the database) can be used for other purposes (covered in the next section), that implies more functionality must be added to the system. As an example, a back-office page must be written in order to manage and maintain the system users (e.g., adding, removing and/or deleting the users). Other functionalities, such as a forgotten password or reset password, must be added to the system in order to maintain the usability of the system.

B) Restricting Users' Access to the System by IP/Mac Address: Knowing the fact

that the framework is located inside a secure domain, access to the server that hosts the framework can be limited using the IP address and/or Media Access Control (MAC) address of the requester. Such restrictions can be accomplished by adding authorised IP or MAC addresses to the surrounding firewalls that protect the system.

This approach would eliminate the necessity of an extra component for the system, but again, a level of administration is needed to protect the system by means of adding and removing addresses that can access the framework. The security of the system would be entirely dependent on the firewalls and the firewall administrators. A combination of A and B would be the most secure approach.

C) Restricting User's Access to the System by Token. As it has been mentioned within the introduction of the research, the main users of the framework will be multi-domain organisations, which generally utilise layers of protection in their organisations; for example, large-scale financial institutes like banks have additional precautions. Knowing that many organisations are currently using one-time tokens to allow their users to access different parts of the network, the framework can be connected into the security infrastructure of the organisation.

By taking this approach, a few pages must be added to the framework in order to challenge users and capture their one-time token password in addition to their credentials. Also, an extra piece of code must be written to send the captured information to the central token validation centre, which is responsible for validating the tokens. The same piece of code would be responsible for receiving the response back from the validation unit and allowing or denying a requester's access to the framework.

This approach does not need any extra components and the security of the system remains within the boundaries; however, the solution assumes the existence of the one-time token password generators and a validation server.

7.7.2 Increasing the Accuracy of the Framework by Reasoning

The second feature which has been requested by the end user was improving the accuracy of the framework. It has been mentioned before that majority of security policies use a simple yet effective pattern: *who can access what under which circumstances*. On the other hand, users are provided with an abstract language to code the policies. Taking these two facts into account shows that even such an abstract language like DSL can be optimised over the time. More usage of the system will reveal the best way to describe a specific security policy using the DSL.

Considering that there might be a possibility of adding a database to the overall architecture of the system, the accuracy of the system can be enhanced by adding a reasoning manager. The reasoning manager would walk through the populated AST of the parsed input DSL script and collect certain keywords that have been used within the scripts. Then, it queries the database for best-practised policies written using the abstract language and feeds the found best practice(s) back to the user via the UI.

Figure 7.5 reveals the enhanced architecture of the framework, as detailed above.

7.8 Testing and Evaluation

Testing and evaluation of the application has been divided into three distinctive tasks as follows:

- Assessing the framework against requirement.
- Assessing the framework against acceptance criteria.
- Assessing by evaluation of the framework.

Combination of the above tasks validated the approach taken to implement the framework.

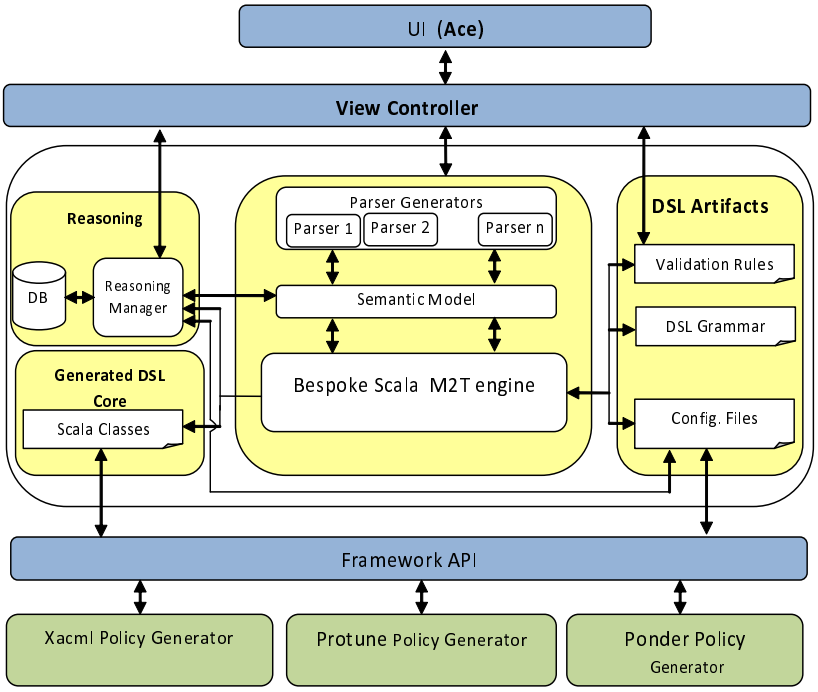


Figure 7.6: The Enhanced Architecture of the Framework

7.8.1 Evaluation the Framework Against Software Requirements

Requirements of the system were discussed back in the Chapter 3 and at the beginning of this chapter. Just to recap, the requirements of the system were a mixture of experts' opinion on how the framework should operate and the advantage of DSL implementation pattern that has been adopted. In this section, the framework's requirements will be assessed against the proposed architecture and its requirements.

- **To Reuse Host Language Infrastructure**: It has been decided to implement the framework by adopting an embedding pattern. That would imply reuse of the host language infrastructure. The second design of the framework, in particular, is purely based on Scala and the features provided by the language.
- **To Provide Well-formed Measurements**: Both the first and second designs of the system provide code-completion and syntax check to the users. Such facilities ensure the validity of written DSL script against both syntactical (accordance to grammar) and contextual (e.g. scoping rules) restrictions.

- **Modular Semantics**: In Section 7.5.1, it has been shown how a feature of the host language can be used as a reusable component that can be put together to make a chain of parsers. Using parser combinators, a pipeline of parser components have been designed and implemented that has been utilised by the framework to parse the DSL scripts.

- **To Implement a High Performance System**: The framework has gone through different stages of testing in order to ensure reliability and accuracy of the system. A detailed list of these tests, all of which passed with a great level of confidence, will be provided in the next section.

- **To Minimise Development Efforts**: Knowing the fact that the development of a DSL is an expensive task, a great level of attention has been paid to minimise the development efforts in both stages of design. MDE, in particular, has been adopted during the first phase to satisfy this requirement of the framework. During the second round, parser combinators have been used to minimise development costs of a performant parser from ground up.

- **To Have Expandability**: The way that the framework was designed and tested using features provided by security policy language candidates, proved that expandability is an achievable target within the scope of the project. Utilising decoupling technique, in addition to designing a DSL language that is independent of underlying security policy languages, ensures expandability of the system in future.

- **To Present User-Friendly UI**: Simplicity is the most desired requirement of the system. In fact, the main reason that the first round of design rejected by users was directly related to this requirement of the system. As the result, a bespoke user interface has been designed for the framework that is simple yet powerful, to satisfy and meet the expectation of the users in many ways.

- **Security of the System**: Security was also another reason that enforced the design review within the life cycle of this project. It was known from outset that this framework will be used within boundaries of a secured environ-

ment. Although a few assumptions were made that potentially compromised the security of the system (e.g. installing Eclispe on system administrators' computer), these issues have been dealt with during system test. In addition to that, a few test scenarios have been executed in order to ensure overall security of the system. Details of these tests will follow in next section.

7.8.2 Evaluation the Framework Against Acceptance Criteria

In addition to evaluation of the framework against its requirements, it should have been rigorously tested against the predefined criteria that were expected to exist when the framework implementation is completed. Those tests usually are automated in software development life-cycle.

Unfortunately, tool support for automated testing of DSLs is not comparable to the like-to-like tools that are provided for GPLs, such as Java and even .Net languages. To be more precise, support for automated testing of a DSL is non-existent [153]. This limits the tester's ability to discover the existence of software errors in a DSL program in the same way that they approach for code written in GPLs.

In the context of this research, there were other challenges, which resulted in using an alternative approach, that are listed here:

- **The Framework Is Not a Complete System**: To be more precise, the output of the framework, which will be policies written in different security policy languages, must be connected to the PAP (please refer to Section 2.3), those policies must be executed on that specific policy server and the results must be examined. In other words, the validity of the output cannot be verified at the point where the code generators produce the policies.

- **Tight Security**: PAPs were not designed to provide online/on-time information. They were designed to capture administrator commands and enforce those to the servers. The PAPs, which would be the point where the frame-

work will be connected to, do not have the capability to send the results to the outside world.

- **Failing Does Not Always Mean Failing**: There are too many parameters that have to be taken to consideration, before a test scenario declared as *failed*. A test scenario could fail as a result of an invalid DSL script, incomplete generator or even incompatible policy server that receives the generated policy. The key point here is that, unlike GPLs where a failure of a test scenario indicates an error in the code, the failure of a test scenario cannot easily identify where an issue is located. A level of human interaction and investigation must be added to the process.

Taking the above into account, a decision was made to abandon the fully automated test strategy and take an alternative approach as described below, which perfectly blends with BDD.

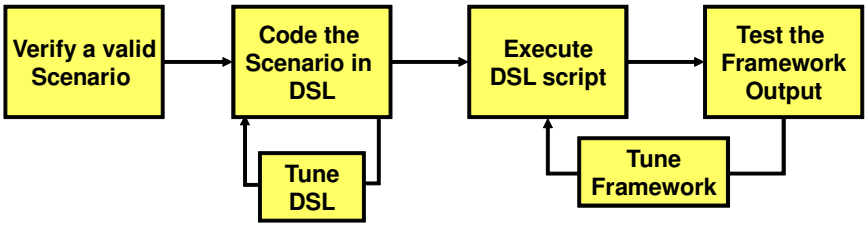


Figure 7.7: The Proposed Testing Procedure for the Framework

In this approach, that was performed by three Capgemini UK testers (one for each language), a transcript has been executed with five different scenarios against each security policy language candidates (i.e. the transcript executed 15 times at each round). The output of each iteration was then sent back to the developers to modify the code/DSL in order to address the issue accordingly. The result of these iterations are presented in this section.

- **Verify a Valid Scenario**: At this stage, a valid scenario with an expected result from all security policy languages must be verified prior to performing the test. It should be stressed that the scenario must be valid, but that does not mean that all the policy languages would be able to code the scenario. If the scenario cannot be coded using a specific security policy language, that must be known before the test cycle commences.

- **Code the Scenario in DSL**: As the next step, the verified scenario must be coded in an abstract policy language (i.e. the DSL). The task must be performed in accordance with the predefined grammar of the DSL.

- **Tune the DSL**: Introducing a new language is an iterative process. The DSL of the current research framework is not an exception. The DSL must be evolved while the framework is constantly tested. During this iterative process, unidentified items will be highlighted and must be added, modified or amended accordingly. Language features, grammars syntax, functionalities which have not been thought of and introduced before can be presented as examples. In these cases, the DSL must be tuned in a way that accommodates/modifies the functionalities and provides them to the framework users. The cycle must be reiterated until all functionalities, features and grammar syntax of a specific scenario have been marked as fit for the purpose of the DSL.

- **Execute the DSL Script**: As the next step, the DSL script must be fed to the framework. Assuming that the corresponding valuation engine, grammar configuration files etc. have been modified accordingly in the previous steps, there should be no issues with the execution of the DSL script by the framework.

- **Test the Framework**: The output of the framework must be checked against the expected result, which was defined at step 1. That must be done separately for every individual security policy language. Specific pattern matching and text comparison codes can be written to fully automate the process; how-

ever, in order to semi-automate the step, *Specs2*, a BDD library for Scala, was used [146]. Using Specs2 , a small acceptance code (Listing 7.5) was written to check the output of the code generators. Irrespective of the level of confidence, the process must be controlled by the developers to ensure the accuracy of the framework and the output of the code generators.

- **Tune the Framework**: There will be cases that result in unexpected behaviour of the framework. In such a scenario, the individual security policy language generator must be modified in a way that produces the expected results. This iterative cycle is continued until either the expected results are generated by the framework or it is determined that this is not achievable unless further changes at other parts of the framework are made. This implies more code changes, which could have their own effect on the framework entirely.

```
class GenereatedPolicySpecs extends Specification {
     def is = s2

 To test the generated policies against expected results

 The generated policy should
   XACML generated policy must match $e1
   Ponder generated policy must match $e2
   Protune generated policy must match $e3
  def e1 = XacmlGeneratedPolicyFile must
          haveSameLinesAs(AccpetdXacmlPolicyFile)
  def e2 = PonderGeneratedPolicyFile
          must haveSameLinesAs(AccpetdPondrerPolicyFile)
  def e3 = ProtuneGeneratedPolicyFile
          must haveSameLinesAs(AccpetdProtunePolicyFile)
}
```

Listing 7.5: Test Script Example Written in SPECS2 (Acceptance Testing)

The cycle described above must be reiterated over and over again, until enough confidence is gained by the DSL and back-end developers.

```
<Policy PolicyId="myPolicy" RuleCombiningAlgId="urn:oasis:
names:tc:xacml:1.0:rule-combining-algorithm:first-applicable"
xmlns="urn:oasis:names:
tc:xacml:1.0:policy" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance">
<Description>This is a Auto generated policy.</Description>
<Target>
<Subjects>
<Subject>
<SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:
function:string-equal">
<AttributeValue DataType="http://www.w3.org/2001/
XMLSchema#string">mySubject</AttributeValue>
<SubjectAttributeDesignator AttributeId="ubdRole"
MustBePresent="false"
DataType="http://www.w3.org/2001/XMLSchema#string"/>
</SubjectMatch>
</Subject>
</Subjects>
<Resources>
<ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function
:string-equal">
<ResourceAttributeDesignator AttributeId="ubd:resource:Type"
DataType="http://www.w3.org/2001/XMLSchema#string"/>
<AttributeValue DataType="http://www.w3.org/2001/XMLSchema#
string">MyTarget</AttributeValue>
</ResourceMatch>
</Resource>
</Resources>
<Actions>
```

```
<Action>
<ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:
string-equal">
<AttributeValue DataType="http://www.w3.org/2001/XMLSchema#
string">Action1</AttributeValue>
<ActionAttributeDesignator AttributeId="ubd:action:type"
DataType="http://www.w3.org/2001/XMLSchema#string"/>
</ActionMatch>
</Action>
</Actions>
</Target>
<Rule RuleId="rul_own_record" Effect="Permit">
<Condition>
<Apply FunctionId=
"urn:oasis:names:tc:xacml:1.0:function:string-equal">
<SubjectAttributeDesignator DataType=
"http://www.w3.org/2001/XMLSchema#string">
mySubjectCondition1
</SubjectAttributeDesignator>
</Apply>
</Condition>
</Rule>
</Policy>
```

Listing 7.6: Example of XACML Policy Generated by Framework

```
execute(denyMyAction(myTarget,MySubject)) :-
   currentRequester(mySubject).

currentRequester(mySubject)->type:provisional.
currentRequester(mySubject)->ontology:{l3s:Dummy_Action}.

denyMyAction(myTarget,MySubject)->type:provisional.
```

```
denyMyAction(myTarget,MySubject)->ontology:{l3s:Dummy_Action
}.
denyMyAction(myTarget,MySubject)->actor:self.
```

Listing 7.7: Example of Protune Policy Generated by Framework

While on the subject of testing the application, it would be worth mentioning that, in addition, the following test strategies have been applied to the system in order to test the integrity of the framework thoroughly. The strategies were:

- **Endurance Test/Stability Test**

  Definition: This tests a given condition over an extended period of time to ensure application availability and sustained performance. Essentially, this test can be described as a load test, where a scenario with a key specification can be run for an extended period of time at a constant load.

  Serious endurance testing can take a week or more, but given the timescales applicable, two hours was the longest practical test duration experienced by the project. Response times and other system metrics, including memory usage (measured by customised log files and print commands) were observed for any indication of system performance degradation over time. The task was mainly executed to determine any possible memory leaks or other problems that may show up after the framework has been live for an extended period. Usually, these tests are most important for high availability 24/7 applications.

  During this test, a specific valid security scenario was fed to the framework and its output has been closely monitored. Although the result was satisfactory, it was noticed that, for extremely large DSL scripts, the session time-out of the webserver that surrounds the framework had to be increased accordingly. Due to the fact that the application should operate on an environment with tight security, the increase of the session time-out could lead to other security breaches (e.g. unauthorised access to the framework, if the administrator forgets to close his/her browser). After a few attempts with different settings, a three-minute session time-out was chosen as an appropriate amount

of time that a session could continue with user inactivity. This provides a good balance between security and performance of the system.

- **Sanity Check**

  Definition: Highly focused tests intended to identify the specific issues of the main functionality of the application. It gives a measure of confidence that the system works as expected, prior to a more exhaustive round of testing.

  A great level of attention was given to the security of the system. As a result, one of the features that was added to the system was the framework to mark any sensitive information provided by the user that must be logged for testing or audit purposes. Such a feature will ensure that the security of the whole domain remain intact, even if human errors occur. A series of scenarios were written to ensure that this feature works under various circumstances. Framework outputs, including log files, were closely monitored as a result. No action was taken, as the framework was designed and coded as to not log/store any sensitive data under any circumstances. Any sensitive information, such as a username that needs to be logged for auditing purposes, was obscured partially to add to the overall security of the system. An example of the output commands can be presented as follows:

```
2014-02-20 10:02:24,267 INFO [STDOUT]
(10.57.115.120-8080-4) About to perform specified Action
2014-02-20 10:02:24,267 INFO [STDOUT]
(10.57.115.120-8080-4) User Amir***ST logged into the system
2014-02-20 10:03:40,932 INFO [STDOUT]
(10.57.115.120-8080-4) Execute the \gls{DSL} was chosen by
user
2014-02-20 10:03:40,932 INFO [STDOUT]
(ajp-10.57.115.120-8080-4) Going to populate the page
```

Listing 7.8: Example of Obscured Logged Information

7.8.3 Evaluation of the Framework by Capturing Experts' Opinion

Last but not the least, the system has been demonstrated to experts and their opinions were captured. During these sessions, the main features of the system have been demonstrated to the experts (mainly the ones who helped the project to start. A list of these individual features are presented in Chapter 1). The following features have been presented to experts:

- The ways that the framework operates .
- The framework's features (please refer to system requirements at Chapter 3).
- The way that the framework can learn new semantics (i.e. framework's extendability).
- Overall security of the framework.
- Architecture of the framework.

A **Question and Answer** session was conducted following the presentation and experts' views on the framework have been captured. All experts who have attended the presentation were satisfied with framework, however, they have suggested a few enhancements to the framework. These have already been discussed in the enhancement section of this chapter.

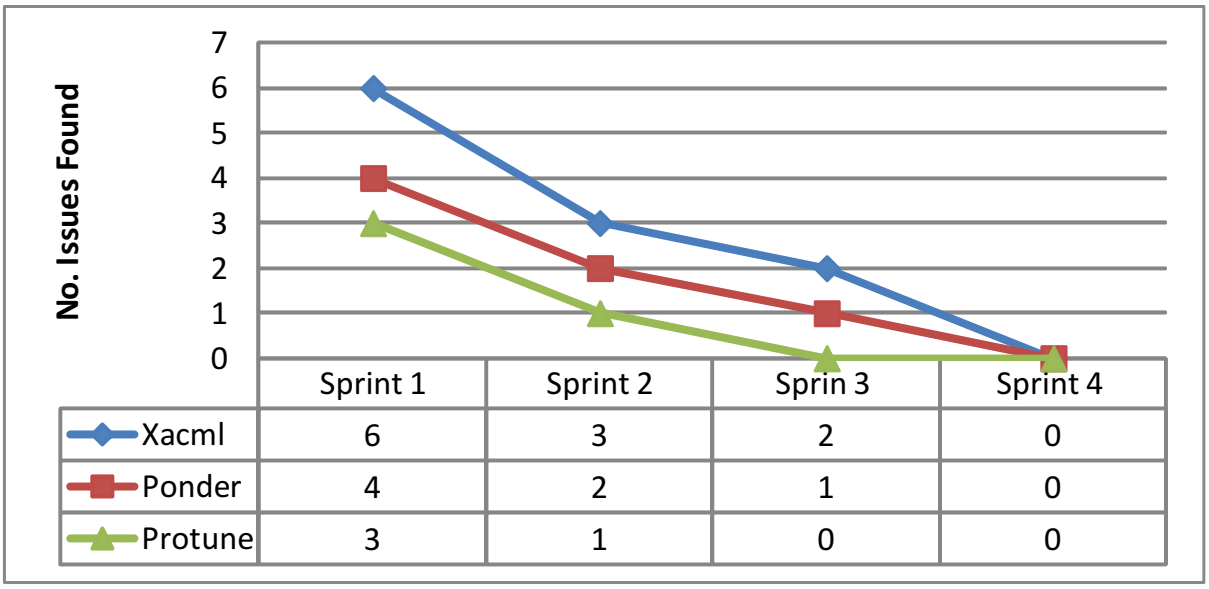
7.9 Analysis of The Framework

Back in chapter 4, it has been justified why evaluation of a limited number of security policy languages against the framework is desired. Hence, adding more security policy languages to the framework is inevitable in future. As a result, this section analyses behaviour of the framework against changes those are required to introduce a new security policy language to the system. This has been analysed as per the languages feature and after a set of features has been added to the framework.

As it has been mentioned before, the software methodology that has been selected for the development of the framework was/is Agile. The process had a short span of the development i.e. two weeks to implement different features for each specific security policy language. At the end of each sprint, the developed features delivered to testers to test it based on the predefined acceptance criteria. Reported issues then were sent back to developers to address them accordingly. The development iterations then continued to the point that it passed all the pre-defined criteria.

The number of issues that have been found by testers per languages was different per features, per sprint. The complexity of the feature and how common the feature was amongst three security policy languages had a direct impact on the number of issues that have been found by the testers. For instance *obligation* that is supported by all languages was more difficult to implement in a way to satisfy all languages, rather than features like *negotiation* that is supported only by one language.

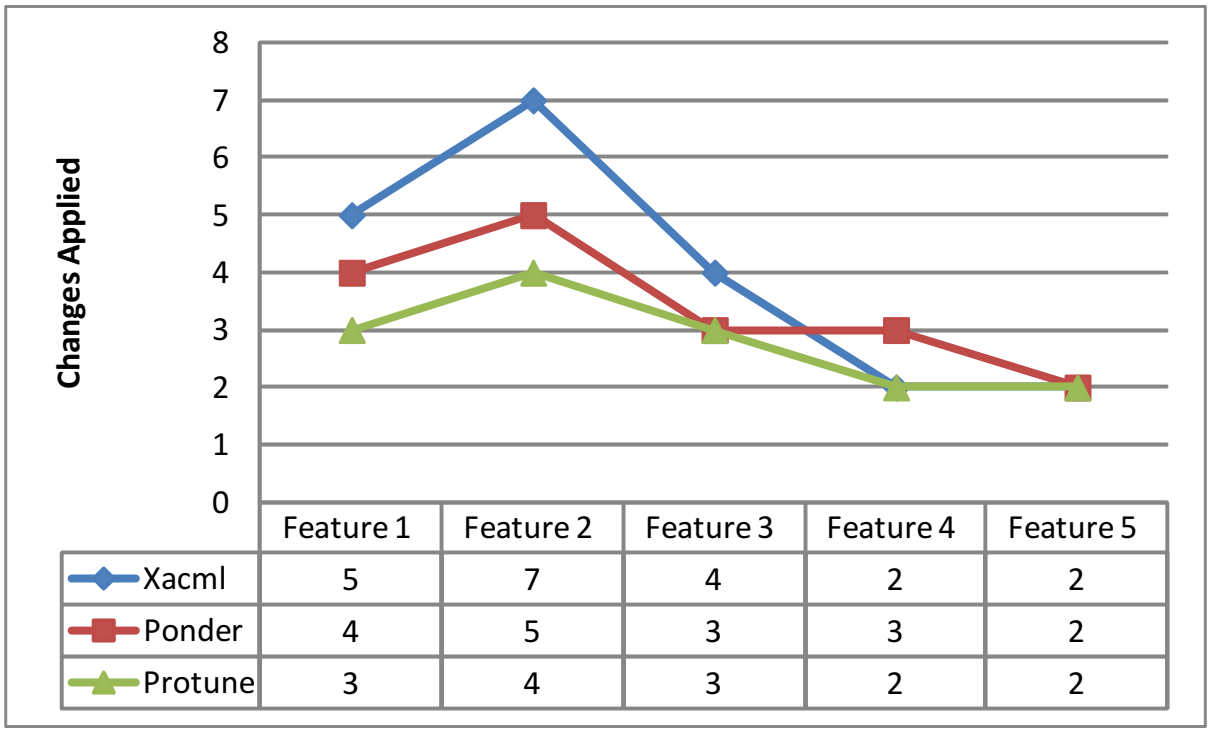
Figure 7.8 presents the number of issues that has been found against one particular feature that was added to the framework. As suggested by the graph, the number of the issues at the end of the first iteration is always more compared to the consecutive sprints. Also it has been noted that the Xacml that is a XML-based security policy language needed more attention than the other two languages. Development of Protune-based security policies created fewer issues (bugs) as it is very close to human language.



	Sprint 1	Sprint 2	Sprin 3	Sprint 4
Xacml	6	3	2	0
Ponder	4	2	1	0
Protune	3	1	0	0

Figure 7.8: Example of Issues Found for a New Feature per Sprint

Figure 7.9 however, presents other aspects of the development. Adding features to framework requires a level of modification to the ASPL in order to allow the system administrators to code the new future via the abstract language. Changing the ASPL in turn forces the semantic model of the framework to be modified to accommodate the new futures that is dictated by the script i.e. ASPL. Figure 7.9 provides the number of changes that were required to add different features to the framework.



	Feature 1	Feature 2	Feature 3	Feature 4	Feature 5
Xacml	5	7	4	2	2
Ponder	4	5	3	3	2
Protune	3	4	3	2	2

Figure 7.9: Required Changes on Semantic Model per Feature

The figure reveals an interesting fact about the framework. The figure proves that irrespective of the required changes enforced by the first feature, the number of changes on the semantic model in the second round of development (i.e. second feature) is always more than the first feature. The reason behind this fact is adding features do have impact on the ASL, however the first feature always can be developed without any constraints enforced by the semantic model. The first feature can be added to the framework freely. But, that is not the case for the second feature onwards, which is added to the framework. The second feature is bound to the constraints already been introduced (that may not be fit for purpose) by the first feature. Hence, the number of changes required to be applied to the semantic model on second feature is always more than the first feature. As more features are added to the framework, ASPL and in turn, the semantic model gradually matures. These findings are in line with the definition of DSL and its properties that were discussed back in Chapter 6 where it has been mentioned that DSL like human languages

evolve and stabilises over a period of time.

Fig 7.9 provides the number of changes that were required to be applied on the semantic model during framework development. As can be discovered from the figure 7.9, Xacml that is a XML-based langue required more changes than human language look alike language e.g. Protune.

7.10 Summary

7.10.1 Chapter Summary

In this chapter, the software methodology for the implementation of the project was chosen and justified. Then, the system requirements that were built upon were detailed, explained and blended with the software methodology.

Then, the chapter presented the high level design of the framework, which was proposed based on the system requirements. The high level architecture was then justified by outlining the advantages of the design. In addition, the detailed design (based on HLA) was presented to the readers. The way that a user's feedback was captured and fed back to the system design was explained in detail. That resulted in the second version of the HLA and its corresponding detail design. Implementation of the system was also discussed to a great extent, to show how different parts of the system tie to each other. In addition, it has been discussed how the framework can be enhanced in future.

Finally, it has been shown how the framework and outline details can be effectively tested. This enabled the system to be more robust and secure in the future.

7.10.2 Research Contributions of the Chapter

To some extent, the entire chapter can be presented as the research's contribution. Ultimate objective of this research was/is to provide a framework for security policy

language, which was presented it in details in this chapter.

Whilst there is no black and white way to decide between different External DSL implementation patterns, it was shown how to determine project specific requirements and how to use that as a filter, to narrow down different possible design approaches.

Different software development methodologies like Agile and user-centric design, that can help to achieve the project's goal with more accuracy, were shown. Additionally, it was demonstrated how test strategies can be utilised to evolve a DSL project in an iterative way.

Chapter 8

Conclusion and Future work

In this chapter, the research presents:

- The conclusion of the project,
- Measuring the defined objectives,
- The future work for the project.

8.1 Conclusion

Novel contribution of this project started with the investigation around security policy languages, which resulted in identifying the necessity of a framework for security policy languages. The advantages that individuals would gain from such a framework, if presented to the industry, were also discussed.

Failure is an acceptable concept in the IT arena. This has been measured by research conducted in the past, including a report compiled by the Standish Group in 1995 that demonstrated that only 16% of the software projects were successful, 53% were challenged and 31% were cancelled. Moreover, the research showed that the average software project runs 222% late, 189% over budget and delivers only 61% of the specified functions. Evidence suggests little has changed since then.

Due to this, it was decided to theoretically prove that the project will be successful when it comes to life. In order to achieve the goal, the following steps have been identified and taken, respectively:

1. The high level requirements of the framework, along with the overview of its abstract security policy language, which will be delivered with the framework, were discussed. That outlined the road-map of the project.

2. The history of security policies, their origins and their models were reviewed in detail. That also included their detailed components and the way they that they work in a secured environment.

3. Then, a literature review of security policy languages was conducted. The review led the research to select a subset of security policy languages to work on the framework as candidates.

4. As the next step, a literature review of security policy language algebras and their characteristics, formalisms and specifications was performed. As a result, one algebra was chosen as a candidate and evaluated against the security policy languages selected in step three.

5. Finally, the areas for improvement in the selected algebra were identified and appropriate solutions to address them were provided by the project.

After it was mathematically proven that the development of the framework is achievable, the design and implementation phases of the project started. Having researched different available best software design and development practises and mapping them to the requirements of this project, it was noted that the design and development of the project could perfectly match the concept of DSLs. As a result, research on the subject was conducted. It was noted that for various reasons, the development of DSLs is an expensive task. In order to control the costs, efforts and the project plan, the design must be performed in a controlled manner; so, it was decided to design and implement the project in stages. Implementation phases have been broken down as follows:

1. **Decision Phase**: The type of DSL was specified at this stage.

2. **Analysis Phase**: Aimed to analyse the domain for which the project is implementing a specific language.

3. **Design Phase**: By analysing different design patterns, it was determined which of the existing design patterns could be fit for the purposes of this research.

4. **Implementation Phase**: Different implementation patterns were reviewed and an appropriate pattern that fits in the context of the research was chosen during this step.

Finally, in an iterative Agile-based approach, the DSL of the project evolved in a way that becomes capable of producing security policies for the security policy language candidates. Lastly, a testing strategy that helped to test the framework from different perspectives was selected and executed at the end of the project.

Now, it is mandatory to go through the research objectives that have been defined at earlier stages, with the aim of evaluating the research accordingly.

Objective One

To define the project specific criteria for shortlisting the security policy languages. The task will be executed following the literature review of security policy languages from different perspectives and categorise them accordingly.

In Chapter 4, different comparison reports on security policy languages that categorise them from different points of view were reviewed in detail. Their uniqueness and characteristics of each individual report were discussed and outlined. Having reviewed these reports in detail, a custom requirement for the research was tailored for the project and applied to the list of available of security policy languages. That resulted in choosing a subset consisting of three security policy languages.

Objective Two

To provide formalism for the framework and evaluate it against security policy language candidates. The task will be executed following the literature review of current algebra for security policy languages.

Steps taken to research algebra for security policy languages were outlined in Chapter 5. The algebras, which have evolved over the last decade and the majority of which were published by well-known authors and presented in reputable conferences and journals, have been reviewed and compared.

Advantages and disadvantages of each individual algebra were also discussed and backed up by acceptable evidence. In addition, an algebra was selected and the area for improvement and enhancement on the selected algebra that fits within the context of this research was identified. The chapter continued by providing the solution and proof to address the identified areas accordingly. Lastly, in the same chapter, completeness of the enhanced algebra was provided.

Objective Three

To design a framework for security policy languages using appropriate software development methodologies.

This objective can be considered the ultimate aim of this research; therefore it was discussed across two chapters, (i.e. Chapter 6 and Chapter 7). In Chapter 6, it was demonstrated that the best software development practice for developing the framework for security policy languages fits perfectly in the context of domain specific languages. In the same chapter, different phases for design and development of a DSL were discussed in detail. As a result of the discussion, the most appropriate design and development pattern for the predefined requirements of the framework was chosen.

In the same chapter, choosing different possible host languages was discussed in detail. Scala chosen as the programming language for the project and its suitability

was outlined.

Objective Four

To implement a Proof of Concept (PoC) using open source components according to software development best practice.

The discussion on implementation of the framework continued in Chapter 7 by providing the high-level design and low-level design of the framework. It was demonstrated how adopting Agile and a user-centric design could rescue a project at its early stages. In addition, it was shown how user feedback and input to the project could affect the HLA and LLD of the project. Also, an outline of how the design can be enhanced and how users can benefit from these was featured by adding more components to the framework.

Objective Five

To design, develop and enhance an abstract security policy language for the framework.

A step-by-step demonstration of the design, development and enhancement of our DSL, which is considered to be an abstract security policy language, using Scalas parser combinators was shown in Chapter 7.

Objective Six

To evaluate the framework in accordance with well-known test strategies.

Finally, the difficulty in fully automated testing strategy for a DSL-based application was discussed. That led the project to overcome the issue by combining a few appropriate testing strategies together and execute them against the project.

8.2 Further Work

Although as part of this research, a novel framework for security policy languages was developed and laboratory studies and a programmatic evaluation process (detailed in previous chapters) satisfied the acceptance criteria of the research, there are still steps that need to be taken, in order to address certain concerns with regards to the framework.

8.2.1 Expansion

In Chapter 4, security policy languages from different perspectives were reviewed and that resulted in tailored requirements for selecting security policy languages. Applying the requirements on the list of well-known and available security policy languages led the project to choose three language candidates. This subset of languages was considered a suitable list of languages, which were then used to evaluate the framework throughout the project.

Although it was proven that the framework, as designed, is capable of transforming the abstract DSL language to a specific security policy language, that does not necessary mean that the framework will be capable of transforming the DSL language to other security policy languages.

There are a number of security policy languages available in the market. It is almost impossible for a single development team to come up with a solution to cover them all; inevitably, the best way to expand the project to other languages, beyond the ones already chosen, would be to make the framework available to other experts who work in the same area and ask for their participation, collaboration and input on the project. Providing the code to the public as open source will also help the framework grow, while others will benefit from its usage.

Assuming such a decision was made, the following steps must be taken before the code is released to the public.

The framework must be properly backed up by a good level of documentation. In other words:

1. The code must be well-documented. That includes inline comments to describe certain parts of the code in detail. Also, the code should come with an extended low-level and high-level design (i.e. extend what has been described in this document) to describe the framework inside and out.

2. A step-by-step document should be provided to show how to develop a plug-in policy generator. The document should also demonstrate that there is possibility that policy code generators could change the top level DSL. In such scenarios, the document also preferably should provide guidance to show how such a change in the top level DSL should be tested to ensure the integrity of the entire framework.

3. A user manual should also be provided. The manual should show how a policy developer (end user) could interact with the framework. It can also provide users with a troubleshooting section to guide them in how to take certain actions, should something go wrong during the interaction with the framework.

8.2.2 Security

Security must have a high level of attention during further development of the system. Assuming the decision was made to incorporate a database into the framework, proper steps must be taken to ensure that sensitive information is excluded from storage in the database. Assuming storing sensitive information in the database is inevitable, then some further steps must be taken to ensure that the sensitive data is properly encrypted before being stored in the database, to reduce the risk of an internal attack.

8.3 The Future of the Framework

Most likely the framework will be used within the application that utilise Micorservices in the future. To justify such a claim the Microservice architecture should be looked at in more detail.

8.3.1 Microservices

Microservices is an approach to develop a single application as a suite of small services, each running in its own process and communicating with each other, often over HTTP resource API. These services are built around business capabilities and independently deployed by continues automated deployment. Microservices are the bare minimum of centralised management of these services, which may be written in different programming languages and use different data storage technologies and located on different domains as well. Microservices usually publish their endpoints that are usually RESTful APIs [20].

REST that is stand for *REpresentational State Transfer*, is an architectural style, and an approach to communications that is often used in the development of Web services. The use of *REST* is often preferred over the more heavyweight *SOAP*. REST is often used in mobile applications, and social networking applications. The REST style emphasises that interactions between clients and services is enhanced by having a limited number of operations. Flexibility is provided by assigning resources their own unique URIs [26].

Taking the fact that Microservices are communicating over HTTP using RESTful APIs, and the fact that micro services can be located in different domains, the implication is that those Microservices can be protected using features that are provided by security policy languages.

Now, in a multi-dimensional organisation where different domains have their own micro services and have their own security policy languages, the security management of the domain becomes very challenging if not impossible. This is where

An Interoperability Framework for Security Policy Languages (i.e. the output of this thesis) can be certainly useful.

Appendices

A List of Open Source Software Used

Software	Version
Java Development Kit	1.6
Scala development Kit	2.10
Play Framework	2.1
Oitok	-
Eclipse	4.0
Xtext	2.2
Xpand	2.0
Ace Editor	1.1.5
Xacml	3.0
Ponder2	2.3
Protune	-
Specs2	2.3
Spring (IoC)	4.0

B Policy Language Comparison

Following is the result of policy languages comparison performed by De Coi *et al.*. [62]

B. Policy Language Comparison

	Cassandra	EPAL	KAoS	PeerTrust	Ponder	Protune	PSPL	Rei	RT	TPL	WSPL	XACML
Well-defined semantics	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	No	No	No
Monotonicity	Yes	–	Yes	Yes	–	Yes	Yes	Yes	Yes	No	–	–
Underlying formalism	Constraint DATALOG	Predicate logic without quantifiers	Description logics	Constraint DATALOG	Object-oriented paradigm	Logic programming	Logic programming	Deontic logic, Logic programming, Description logics	Constraint DATALOG	–	–	–
Action execution	Yes (side-effect free)	Yes	No	Yes (only sending evidences)	Yes (access to system properties)	Yes	Yes (only sending evidences)	No	No	No	Yes	Yes
Delegation	Yes	No	No	Yes	Yes	Yes	No	Yes	Yes (RT^D)	No	No	No
Type of evaluation	Distributed policies, Local evaluation	Local	Local	Distributed	Local	Distributed	Distributed	Distributed policies, Local evaluation	Local	Local	Distributed policies, Local evaluation	Distributed policies, Local evaluation
Evidences	Credentials	No	No	Credentials, Declarations	–	Credentials, Declarations	Credentials, Declarations	–	Credentials	Credentials	No	No
Negotiation	Yes	No	No	Yes	No	Yes	Yes	No	No	Yes	No (policy matching supported)	No
Result format	A/D and a set of constraints	A/D, `scope error`, `policy error`	A/D	A/D	A/D	Explanations	A/D	A/D[a]	A/D	A/D	A/D, `not applicable`, `indeterminate`	A/D, `not applicable`, `indeterminate`
Extensibility	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	No	No	No	Yes

Figure 1: Policy Languages Comparison

C JVM Language Comparison

Following is the result of JVM languages comparison performed by Wing Hang Li *et al.*.

C. JVM Language Comparison

language/benchmark	compiled or interpreted	bytecodes executed	non-Java bytecodes %	methods executed	non-Java methods %	method hotness %	basic block hotness %	objects allocated	non-Java objects %	boxed primitive use %
Java/binarytrees	AOT	170105156	0.00	25176723	0.00	99.81	99.99	2651267	0.00	0.00
Java/fannkuchredux	AOT	141676329	0.00	1003221	0.00	99.96	99.99	2293	0.00	0.26
Java/fasta	AOT	100060247	0.00	2456162	0.00	76.76	99.99	2343	0.00	0.26
Java/knucleotide	AOT	1202353259	0.00	49678282	0.00	50.24	99.99	572950	0.00	0.00
Java/mandelbrot	AOT	262768004	0.00	78893	0.00	99.97	100.00	2878	0.00	0.21
Java/meteor	AOT	330191630	0.00	5431167	0.00	99.48	99.99	262824	0.00	0.00
Java/nbody	AOT	171061794	0.00	152240	0.00	99.47	99.98	4698	0.00	0.17
Java/regexdna	AOT	3947367657	0.00	226454549	0.00	99.68	100.00	312589	0.00	0.09
Java/revcomp	AOT	54864024	0.00	1548046	0.00	6.69	99.97	2868	0.00	9.24
Java/spectralnorm	AOT	38870722	0.00	1079088	0.00	99.76	99.94	4785	0.00	0.13
Java/avrora	AOT	4043803001	0.00	322511916	0.00	99.32	99.96	991319	0.00	0.12
Java/batik	AOT	812845536	0.00	40670592	0.00	74.52	99.75	ET failed to complete trace		
Java/fop	AOT	149718163	0.00	9541593	0.00	90.68	98.88	ET failed to complete trace		
Java/h2	AOT	9653613964	0.00	769997845	0.00	93.64	99.98	25548956	0.00	1.92
Java/javac/sunflow	AOT	16635753858	0.00	1077651364	0.00	88.01	99.50	ET failed to complete trace		
Java/luindex	AOT	123562365	0.00	4524912	0.00	92.91	99.86	112724	0.00	0.01
Java/lusearch	AOT	1140089272	0.00	50417227	0.00	99.17	99.96	1272318	0.00	0.00
Java/pmd	AOT	54097593	0.00	2366460	0.00	91.24	99.15	133466	0.00	0.26
Java/sunflow	AOT	1987247487	0.00	64365743	0.00	99.09	99.98	2422198	0.00	0.03
Java/xalan	AOT	887655171	0.00	48845720	0.00	97.35	99.41	1117739	0.00	0.09
Clojure/binarytrees	AOT	458013959	50.18	46066179	23.11	99.81	99.93	5519929	48.01	47.82
Clojure/fannkuchredux	AOT	949916910	32.01	77452181	1.68	98.21	99.96	258837	0.67	0.79
Clojure/fasta	AOT	430594595	40.53	37624799	9.69	90.69	99.93	2641156	45.50	45.64
Clojure/knucleotide	AOT	1741954505	23.02	131910496	4.43	80.60	99.97	7165252	4.66	85.27
Clojure/mandelbrot	AOT	697889749	62.96	42874651	15.85	99.82	99.94	821435	0.48	70.20
Clojure/meteor	AOT	4097931808	28.21	389333271	3.91	99.02	99.98	10199180	9.40	25.69
Clojure/nbody	AOT	292196152	66.32	17028481	39.02	99.18	99.90	1239830	0.13	80.85
Clojure/regexdna	AOT	4689013910	0.54	226505597	0.02	99.43	99.99	507349	0.43	0.40
Clojure/revcomp	AOT	224945319	44.47	18146640	0.55	55.92	99.86	233852	0.57	1.29
Clojure/spectralnorm	AOT	152301546	38.49	7716311	0.80	99.38	99.78	259541	3.25	3.23
Clojure/incanter	JIT	273986921	4.79	17850648	4.48	96.72	99.03	1635693	1.78	1.33
Clojure/leiningen	AOT	1006923598	2.18	57874646	2.32	97.64	99.49	1278616	1.94	1.68
Clojure/noir/blog	AOT	1323734731	1.37	21635558	5.04	99.81	99.90	1909561	1.09	1.11
JRuby/binarytrees	AOT	3172344964	12.57	354573991	6.27	99.98	99.99	12667277	0.01	0.00
JRuby/fannkuchredux	AOT	4684146884	13.34	537414857	3.14	89.65	100.00	3272573	0.04	0.01
JRuby/fasta	AOT	5564934530	6.21	617823701	2.41	99.97	99.99	14888989	0.01	32.24
JRuby/knucleotide	AOT	11381452118	3.63	1081798483	2.46	99.15	100.00	18740991	0.01	0.00
JRuby/mandelbrot	AOT	13483710728	8.53	1654536903	0.80	99.97	99.99	ET failed to complete trace		
JRuby/meteor	AOT	25990259655	6.26	2455403535	2.72	99.99	100.00	ET failed to complete trace		
JRuby/nbody	AOT	7054115048	8.78	950461750	0.81	99.97	100.00	30638260	0.00	94.66
JRuby/regexdna	AOT	10090042993	0.00	628235221	0.00	99.87	100.00	336937	0.41	0.10
JRuby/revcomp	AOT	277284831	1.52	18636289	1.48	76.62	99.92	333311	0.41	0.11
JRuby/spectralnorm	AOT	2183551569	9.14	272266633	3.32	99.96	99.99	8515852	0.02	96.21
JRuby/jrails	AOT	5197249613	0.47	313574609	1.08	98.24	99.77	ET failed to complete trace		
JRuby/lingo	JIT	12322206926	2.36	1110460420	1.73	98.88	99.88	ET failed to complete trace		
JRuby/warbler	JIT	7943627729	1.33	526990999	1.02	99.22	99.84	ET failed to complete trace		
Jython/binarytrees	int.	6116792936	7.77	481948160	2.21	99.84	99.99	27183665	0.04	19.42
Jython/fannkuchredux	int.	4429512501	4.99	452006371	0.00	94.81	99.98	20308641	0.01	1.72
Jython/fasta	int.	7361689841	6.92	758276792	0.98	99.21	99.98	11091087	0.19	44.00
Jython/knucleotide	int.	10899263124	3.79	1142232229	0.00	95.75	99.99	40776401	0.03	32.48
Jython/mandelbrot	int.	6260601597	5.58	645681679	0.01	94.31	99.99	20477645	0.01	95.81
Jython/nbody	int.	6656716672	7.91	899402775	0.00	91.93	99.99	33011071	0.00	82.28
Jython/regexdna	int.	7144141615	0.03	432717385	0.02	86.18	99.97	136423507	0.02	0.19
Jython/revcomp	int.	1164263231	0.20	78797100	0.04	92.18	99.81	3090414	0.05	3.64
Jython/spectralnorm	int.	2061855998	5.75	225838375	1.37	96.72	99.96	16063707	0.05	57.64
Scala/binarytrees	AOT	116374550	98.60	8209028	98.56	99.67	99.92	2666944	99.37	0.40
Scala/fannkuchredux	AOT	292713859	99.59	27164782	99.67	99.96	99.99	6612	5.05	15.62
Scala/fasta	AOT	247153292	81.63	29521359	66.34	79.92	99.99	3621	4.97	14.50
Scala/knucleotide	AOT	2743126868	92.52	279800678	91.54	96.91	99.99	4329096	32.82	65.83
Scala/mandelbrot	AOT	281828727	99.48	12620894	99.27	99.93	99.98	19299	57.46	5.72
Scala/meteor	AOT	38201494343	97.60	4688797729	95.62	99.99	100.00	2340559	98.48	0.88
Scala/nbody	AOT	307725221	99.67	41360373	99.86	99.60	99.99	5920	2.57	6.05
Scala/regexdna	AOT	3681464412	14.54	246469635	36.78	99.93	100.00	6555923	26.30	0.01
Scala/revcomp	AOT	69607945	98.85	3079180	98.42	14.09	99.97	3181	5.85	7.17
Scala/spectralnorm	AOT	44336176	97.61	1110874	94.41	99.75	99.92	6546	9.21	5.44
Scala/apparat	AOT	5473129320	69.78	782465549	59.90	99.38	99.89	8828083	69.63	0.45
Scala/kiama	AOT	96747090	44.44	8744873	72.05	95.47	99.41	534871	60.45	1.64
Scala/scalac	AOT	841758028	50.77	86270143	77.48	97.11	99.82	4384353	59.82	1.09
Scala/scaladoc	AOT	1009919955	56.19	111072697	76.91	97.16	99.82	4283738	62.97	0.71
Scala/scalap	AOT	50006991	23.40	3387070	53.31	94.23	99.45	166594	43.44	1.52
Scala/scalariform	AOT	665331715	76.76	102056949	88.80	94.47	99.25	5121304	82.18	4.53
Scala/scalatest	AOT	867200843	30.23	61964606	67.16	99.90	99.96	3503330	46.58	0.13
Scala/scalaxb	AOT	371552561	47.38	35445090	72.17	94.64	99.54	1432529	52.82	21.09
Scala/specs	AOT	721279181	20.13	49903727	55.97	99.76	99.90	3872916	41.33	0.90
Scala/tmt	AOT	44096213851	79.66	4757318965	80.30	97.88	100.00	110104114	9.14	74.64

Figure 2: JVM Languages Comparison

D Requirement Gathering Questioner

Following is a snapshot of questioner used to capture functional and non-functional requirements of the framework.

D. Requirement Gathering Questioner

Dear Colleagues,

Many thanks for participating in this survey.

As you have been informed during the interview, an academic project is about to start to provide industry with an *Interoperability framework for Security Policy Languages*. Due to non-existence of such a framework (or any similar product), you have been selected to provide the project with your invaluable views on how such a framework should work. This Survey has been designed to capture these requirements in a formal way.

Please provide the requirement that you would expect the framework to provide in the following form. Please use one form per requirement and please be expressive as much as possible.

I would like to take this opportunity and appreciate your help and support in advance.

Kind regards

Amir Aryanpour

No	Question	Answer	Comments
	Name		
	Job Title		
1	What does this feature need to do?		
2	What is the end result of doing this?		
3	What needs to happen next?/ What must happen before?		
4	How will user use this feature?		
5	How will user use this feature?		
6	How might we think about this feature differently?		
7	Where would the user access (start) this feature?		
8	Where would the results be visible?		
9	When will this feature be used?		
10	When will the feature fail?		
11	Who will use this feature?		
12	Is there any other way to accomplish this?		

Figure 3: Requirement Gathering Questioner

E Survey Questioner

Following is a snapshot of survey used to capture user's opinion after a PoC of the framework presented to them.

E. Survey Questioner

Dear Colleagues,

Many thanks for participating in this exercise.

You will be given a random scenario to code with the Abstract Security Policy Language using the framework provided.
Please complete the following form accurately and legibly when you have finished the exercise.

I would like to take this opportunity and appreciate your help and support in advance.

Kind regards

Amir Aryanpour

No	Question	Answer	Comments
	Name		
	Job Title		
	In scale of 1 to 5, 1 is totally satisfied and 5 is totally unsatisfied		
1	In scale of 1 to 5, how would you rate the simplicity of the Abstract Security Policy Language?	1 ○ 2 ○ 3 ○ 4 ○ 5 ○	
	Please elaborate on the answer provided		
2	In scale of 1 to 5, how would you rate the user friendliness of the environment?	1 ○ 2 ○ 3 ○ 4 ○ 5 ○	
	Please elaborate on the answer provided		
3	In scale of 1 to 5, how would you rate the easiness of the framework installation on your desktop?	1 ○ 2 ○ 3 ○ 4 ○ 5 ○	
	Please elaborate on the answer provided		
4	In scale of 1 to 5, how would you rate the overall usability of the framework? (Developers only)	1 ○ 2 ○ 3 ○ 4 ○ 5 ○	
	Please elaborate on the answer provided		
5	In scale of 1 to 5, how would you rate the simplicity of the EBNF language? (Developers only)	1 ○ 2 ○ 3 ○ 4 ○ 5 ○	
	Please elaborate on the answer provided		
6	In scale of 1 to 5, how would you rate the overall expandability of the framework? (Developers only)	1 ○ 2 ○ 3 ○ 4 ○ 5 ○	
	Please elaborate on the answer provided		
7	In scale of 1 to 5, how would you rate responsiveness/ robustness of the framework?	1 ○ 2 ○ 3 ○ 4 ○ 5 ○	
	Please elaborate on the answer provided		
8	In scale of 1 to 5, how would you rate your overall satisfaction with the framework?	1 ○ 2 ○ 3 ○ 4 ○ 5 ○	
	Please elaborate on the answer provided		
9	Please provide us with any issue(s)/problem(s) you have encountered during this exercise.		
10	Please provide any suggestion that can help us to improve the framework.		

Figure 4: Survey Questioner

References

[1] Ajax based source code editor. Available at http://debasishg.blogspot.co.uk/2008/04/external-dsls-made-easy-with-scala.html, [Online] , Accessed: 2013-09-15.

[2] Behavior-driven development. Available at http://en.wikipedia.org/wiki/Behavior-driven_development, [Online] , Accessed: 2014-09-28.

[3] Boxplot. Available at http://en.wikipedia.org/wiki/Box_plot, [Online] , Accessed: 2013-02-07.

[4] Clojure vs scala, author =Mark Engelberg, howpublished = Available at http://programming-puzzler.blogspot.co.uk/2013/12/clojure-vs-scala.html, [Online] , Accessed: 2014-01-10.

[5] Codemirror a versatile text editor. Available at http://codemirror.net/, [Online] , Accessed: 2013-09-15.

[6] Common information model. Available at http://www.dmtf.org/standards/cim, [Online] , Accessed: 2010-09-30.

[7] Document object model. Available at http://www.w3.org/DOM/, [Online] , Accessed: 2011-02-27.

[8] Dsl development environment. Available at https://www.jetbrains.com/mps/, [Online] , Accessed: 2013-05-10.

[9] Eclipse modelling. Available at http://www.eclipse.org/modeling/m2t/?project=xpand, [Online] , Accessed: 2012-10-19.

[10] Eclipse modelling framework. Available at http://eclipse.org/modeling/emf/, [Online] , Accessed: 2012-10-19.

[11] Eclipse plugins. Available at http://ostatic.com/eclipse, [Online] , Accessed: 2012-10-19.

[12] Editarea, a free javascript editor for source code. Available at http://www.cdolivet.com/editarea/?page=editArea, [Online] , Accessed: 2013-09-15.

[13] The extensible stylesheet language family (xsl). Available at http://www.w3.org/Style/XSL/, [Online] , Accessed: 2011-02-27.

[14] High level design. Available at http://en.wikipedia.org/wiki/High-level_design, [Online] , Accessed: 2013-06-23.

[15] High performance code editor. Available at http://ace.c9.io/#nav=about, [Online] , Accessed: 2013-09-15.

[16] The high velocity web framework for scala and java. Available at https://www.playframework.com/, [Online] , Accessed: 2014-01-28.

[17] Interaction for the new millennium. Available at http://www.sdlforum.org/MSC2000present/index.htm, [Online] , Accessed: 2013-01-11.

[18] Jruby a programming language. Available at http://jruby.org/, [Online] , Accessed: 2013-02-07.

[19] Jython, an implementation of the python programming language on jvm platform, author =Frank Wierzbicki, howpublished = Available at https://irony.codeplex.com/, [Online] , Accessed: 2013-02-07.

[20] Microservices, a definition of this new architectural term. Available at http://martinfowler.com/articles/microservices.html, [Online] , Accessed: 2016-01-28.

[21] An open source identity and access management framework. Available at http://www.gluu.org/gluu-server/overview/, [Online] , Accessed: 2014-01-28.

[22] Overview of xacml. Available at http://en.wikipedia.org/wiki/XACML, [Online] , Accessed: 2012-05-11.

[23] Policy framework for device apis. Available at http://dev.w3.org/2009/dap/policy/Framework.html, [Online] , Accessed: 2013-01-20.

[24] Review of security policy languages and frameworks. Available at http://www.w3.org/Policy/pling/wiki/PolicyLangReview, [Online] , Accessed: 2010-09-30.

[25] Spring inversion of control. Available at http://docs.spring.io/spring-framework/docs/current/spring-framework-reference/html/beans.html, [Online] , Accessed: 2012-11-01.

[26] What is rest. Available at http://searchsoa.techtarget.com/definition/REST, [Online] , Accessed: 2016-01-28.

[27] Xquery 1.0: An xml query language (second edition). Available at http://www.w3.org/TR/xquery/, [Online] , Accessed: 2011-02-27.

[28] Xtext a framework for development of programming languages and domain specific languages. Available at https://eclipse.org/Xtext/, [Online] , Accessed: 2013-05-10.

[29] Integrity considerations for secure computer systems. *MITRE Co., technical report ESD-TR 76-372*, 1977.

[30] Iso/iec 14977:1996 information technology - syntactic metalanguage - extended bnf, 1996.

[31] *Post-design domain-specific language embedding: a case study in the software engineering domain*, 2002.

[32] *An introduction to the Web Services Policy Language*, 2004.

[33] Trust-X: A Peer-to-Peer framework for trust establishment. *IEEE Trans. on Knowl. and Data Eng.*, 16(7):827–842, 2004.

[34] Survey on xml-based policy languages for open environments. *Journal of Information Assurance and security*, 1(1):11–20, Mar. 2006.

[35] C. Abras, D. Maloney-krichmar, and J. Preece. User-centered design. In *In Bainbridge, W. Encyclopedia of Human-Computer Interaction. Thousand Oaks: Sage Publications*. Publications, 2004.

[36] J. Akerley, A. Parlavecchia, and N. Li. *Programming with VisualAge for Java Version 2.0 with Cdrom*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2nd edition, 1999.

[37] P. Almqvist. 10 user interface design fundamentals. Available at `http://blog.teamtreehouse.com/10-user-interface-design-fundamentals`, [Online] , Accessed: 2013-06-23.

[38] G. A. And, G. Antoniou, G. Antoniou, F. Van Harmelen, and F. Van Harmelen. Web ontology language: Owl. In *Handbook on Ontologies in Information Systems*, pages 67–92, 2003.

[39] A. Anderson. A comparison of two privacy policy languages: Epal and xacml. Technical report, Mountain View, CA, USA, 2005.

[40] R. J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley, Hoboken, NJ, USA, 2008.

[41] M. Anlauff, P. Kutter, and A. Pierantonio. Domain specific languages in software engineering.

[42] P. Ashley, S. Hada, G. Karjoth, C. Powers, and M. Schunter. Enterprise privacy authorization language (EPAL 1.2). Technical report, IBM, 2003.

[43] J. W. Backus. The syntax and semantics of the proposed international algebraic language of the zurich acm-gamm conference. In *IFIP Congress*, pages 125–131, 1959.

[44] M. Y. Becker, C. Fournet, and A. D. Gordon. SecPAL: Design and semantics of a decentralized authorization language. *Journal of Computer Security*, 18(4):619–665, Jan. 2010.

[45] M. Y. Becker and P. Sewell. Cassandra: distributed access control policies with tunable expressiveness. pages 159–168, June 2004.

[46] D. E. Bell and L. J. LaPadula. Secure Computer Systems: Mathematical Foundations. Technical report, MITRE CORP BEDFORD MA, Nov. 1973.

[47] J. Bentley. *Programming Pearls (2nd Edition)*. Addison-Wesley Professional, 2 edition, Oct. 1999.

[48] T. J. Bergin, Jr. and R. G. Gibson, Jr., editors. *History of Programming languages—II*. ACM, New York, NY, USA, 1996.

[49] E. Bertino, S. Castano, and E. Ferrari. On specifying security policies for web documents with an XML-based language. In *SACMAT '01: Proceedings of the sixth ACM symposium on Access control models and technologies*, pages 57–65, New York, NY, USA, 2001. ACM.

[50] J. Bézivin. Model Driven Engineering: An Emerging Technical Space. In R. Lämmel, J. a. Saraiva, and J. Visser, editors, *Generative and Transformational Techniques in Software Engineering*, volume 4143 of *Lecture Notes in Computer Science*, chapter 2, pages 36–64. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2006.

[51] M. Blaze, J. Ioannidis, and A. D. Keromytis. Experience with the keynote trust management system: Applications and future directions. In *Proceedings of the 1st International Conference on Trust Management*, iTrust'03, pages 284–300, Berlin, Heidelberg, 2003. Springer-Verlag.

[52] P. Bonatti, S. De Capitani di Vimercati, and P. Samarati. An algebra for composing access control policies. *ACM Trans. Inf. Syst. Secur.*, 5(1):1–35, Feb. 2002.

[53] P. Bonatti and D. Olmedilla. Driving and monitoring provisional trust negotiation with metapolicies. In *In 6th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2005*, pages 14–23. IEEE Computer Society, 2005.

[54] P. Bonatti and P. Samarati. Regulating service access and information release on the web. In *CCS '00: Proceedings of the 7th ACM conference on Computer and communications security*, volume 10, pages 134–143, New York, NY, USA, 2000. ACM.

[55] P. A. Bonatti, G. Antoniou, M. Baldoni, C. Baroglio, C. Duma, N. Fuchs, W. Nejdl, D. Olmedilla, J. Peer, V. Patti, and N. Shamheri. The rewerse view on policies. In *In Proc. of the ISWC Semantic Web Policy Workshop (SWPW), http: //ebiquity.umbc.edu/get/a/publication/215.pdf*, 2005.

[56] M. Bravenboer and E. Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. *SIGPLAN Not.*, 39:365–383, Oct. 2004.

[57] D. F. C. Brewer and M. J. Nash. The Chinese wall security policy. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 206–214, 1989.

[58] S. A. Brown, C. E. Drayton, and B. Mittman. A description of the apt language. *Commun. ACM*, 6(11):649–658, Nov. 1963.

[59] W. H. Burge. *Recursive programming techniques*. The systems programming series. Addison-Wesley, Reading (Mass.), 1975.

[60] D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security policies. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 184–194. IEEE Computer Society Press, 1987.

[61] F. J. G. Clemente, G. M. Pérez, J. A. B. Blaya, and A. G. Skarmeta. Representing security policies in web information systems. *International World Wide Web Conference (WWW 2005)*, 2005.

[62] J. L. Coi and D. Olmedilla. A review of trust management, security and privacy policy languages. In *International Conference on Security and Cryptography (SECRYPT 2008)*, pages 483–490. INSTICC Press, 2008.

[63] C. Consel, H. Hamdi, L. Rveillre, L. Singaravelu, H. Yu, and C. Pu. Spidle: A dsl approach to specifying streaming applications. In F. Pfenning and Y. Smaragdakis, editors, *Generative Programming and Component Engineering*, volume 2830 of *Lecture Notes in Computer Science*, pages 1–17. Springer Berlin Heidelberg, 2003.

[64] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, 2nd edition, Sept. 2001.

[65] L. Cranor, M. Langheinrich, and M. Marchiori. A p3p preference exchange language 1.0 (appel1. 0). *W3C working draft*, 15, 2002.

[66] D. Crockford. JSON: The Fat-Free Alternative to XML, Dec. 2006.

[67] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specification language. In *LECTURE NOTES IN COMPUTER SCIENCE*, pages 18–38. Springer-Verlag, 2001.

[68] S. Dasgupta, C. Papadimitriou, and U. Vazirani. *Algorithms*. McGraw-Hill Science/Engineering/Math, 1 edition, Sept. 2006.

[69] F. v. H. Deborah L. McGuinness. Feature synopsis for owl lite and owl. Available at `http://www.w3.org/TR/2002/WD-owl-features-20020729/`, [Online] , Accessed: 2013-05-17.

[70] J. den Haan. 15 lessons learned during the development of a model driven software factory. Available at `http://www.theenterprisearchitect.eu/blog/2010/09/06/`

15-lessons-learned-during-the-development-of-a-model-driven-software-factory/ [Online] , Accessed: 2013-03-21.

[71] J. den Haan. Domain specific language design based on domain-driven design. Available at http://www.theenterprisearchitect.eu/blog/2009/05/06/dsl-development-7-recommendations-for-domain-specific-language-design-based-o [Online] , Accessed: 2013-03-21.

[72] J. den Haan. Mde model driven engineering reference guide. Available at http://www.theenterprisearchitect.eu/blog/2009/01/15/mde-model-driven-engineering-reference-guide/, [Online] , Accessed: 2013-03-23.

[73] J. den Haan. Mendix platform. Available at http://www.mendix.com/application-platform-as-a-service/, [Online] , Accessed: 2013-03-21.

[74] A. V. Deursen and P. Klint. Little languages: Little maintenance?, 1998.

[75] D. Duggan. A mixin-based, semantics-based approach to reusing domain-specific programming languages. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, ECOOP '00, pages 179–200, London, UK, UK, 2000. Springer-Verlag.

[76] C. Duma, A. Herzog, and N. Shahmehri. Privacy in the semantic web: What policy languages have to offer. In *Policies for Distributed Systems and Networks, 2007. POLICY '07. Eighth IEEE International Workshop on*, pages 109–118, 2007.

[77] S. Efftinge and M. Völter. oAW xText: A framework for textual DSLs. In *Eclipsecon Summit Europe 2006*, Nov. 2006.

[78] C. Elliott, S. Finne, and O. de Moor. Compiling Embedded Languages. In *SAIG*, pages 9–27, 2000.

[79] M. Engelberg. Ponder overview. Available at `http://ponder2.net/cgi-bin/moin.cgi/Ponder2Overview`, [Online] , Accessed: 2013-09-10.

[80] E. Evans. *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[81] R. d. A. Falbo, G. Guizzardi, and K. C. Duarte. An ontological approach to domain engineering. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, SEKE '02, pages 351–358, New York, NY, USA, 2002. ACM.

[82] D. Ferraiolo and R. Kuhn. Role-based access controls. In *15th NIST-NCSC National Computer Security Conference*, pages 554–563, Baltimore, MD, Oct. 1992.

[83] R. Finkel. *Advanced Programming Language Design*. Addison-Wesley, 1996.

[84] M. Flatt. Composable and compilable macros: you want it when? In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, volume 37, pages 72–83, New York, NY, USA, Sept. 2002. ACM.

[85] M. Fowler. Language workbenches: The killer-app for domain specific languages? Available at `http://www.martinfowler.com/articles/languageWorkbench.html`, [Online] , Accessed: 2013-03-21.

[86] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 1 edition, Nov. 2002.

[87] M. Fowler. *Domain-Specific Languages (Addison-Wesley Signature Series (Fowler))*. Addison-Wesley Professional, 1 edition, Oct. 2010.

[88] M. Fowler and J. Highsmith. The agile manifesto. *Software Development*, 9(8):28–35, 2001.

[89] W. Frakes, R. Prieto-Diaz, and C. Fox. Dare: Domain analysis and reuse environment. *Ann. Softw. Eng.*, 5:125–141, Jan. 1998.

[90] R. Frost. Monadic memoization towards correctness-preserving reduction of search. In *Proceedings of the 16th Canadian Society for Computational Studies of Intelligence Conference on Advances in Artificial Intelligence*, AI'03, pages 66–80, Berlin, Heidelberg, 2003. Springer-Verlag.

[91] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[92] R. Gavriloaie, W. Nejdl, D. Olmedilla, K. E. Seamons, and M. Winslett. No registration needed: How to use declarative policies and negotiation to access sensitive resources on the semantic web. In *The Semantic Web: Research and Applications*, pages 342–356. 2004.

[93] D. Ghosh. External dsls made easy with scala parser combinators. Available at `http://debasishg.blogspot.co.uk/2008/04/external-dsls-made-easy-with-scala.html`, [Online] , Accessed: 2013-09-07.

[94] D. Ghosh. *DSLs in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2010.

[95] G. S. Graham and P. J. Denning. Protection: principles and practice. In *Proceedings of the May 16-18, 1972, spring joint computer conference*, AFIPS '72 (Spring), pages 417–429, New York, NY, USA, 1972. ACM.

[96] S. Halloway and A. Bedra. *Programming Clojure*. Pragmatic Bookshelf, 2nd edition, 2012.

[97] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in operating systems. *Commun. ACM*, 19(8):461–471, Aug. 1976.

[98] H. Hartig, O. Kowalski, and W. Kuhnhauser. The birlix security architecture. *Journal of Computer Security*, 2:5–21, 1993.

[99] D. Heimbigner. Dmtf-cim to owl: A case study in ontology conversion. In *16th International Confernece of Software Engineering and Knowledge Engineering (SEKE) (2004)*.

[100] A. Herzberg, Y. Mass, J. Mihaeli, D. Naor, and Y. Ravid. Access control meets public key infrastructure, or: assigning roles to strangers. pages 2–14, Aug. 2002.

[101] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, Oct. 1969.

[102] C. Hofer, K. Ostermann, T. Rendel, and A. Moors. Polymorphic embedding of dsls. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*, GPCE '08, pages 137–148, New York, NY, USA, 2008. ACM.

[103] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, M. Dean, et al. Swrl: A semantic web rule language combining owl and ruleml. *W3C Member submission*, 21:79, 2004.

[104] Iso. ISO/IEC 10181-3:1996 - information technology – open systems interconnection – security frameworks for open systems: Access control framework.

[105] S. Johnson. YACC: Yet another compiler-compiler, 1979.

[106] L. Kagal, T. Finin, and A. Joshi. A policy based approach to security for the semantic web. In *The SemanticWeb - ISWC 2003*, pages 402–418. 2003.

[107] L. Kagal, T. Finin, and A. Joshi. A policy language for a pervasive computing environment. In *Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks*, POLICY '03, pages 63–, Washington, DC, USA, 2003. IEEE Computer Society.

[108] D. Knuth. The genesis of attribute grammars. In *Attribute Grammars and their Applications*, pages 1–12. 1990.

[109] D. Koenig, A. Glover, P. King, G. Laforge, and J. Skeet. *Groovy in Action*. Manning Publications Co., Greenwich, CT, USA, 2007.

[110] D. S. Kolovos, R. F. Paige, T. Kelly, and F. A. C. Polack. Requirements for Domain-Specific Languages. In *Proc. 1st ECOOP Workshop on Domain-Specific Program Development (DSPD 2006)*, Nantes, France, July 2006.

[111] J. Kulandai. Dependency injection (di) with spring. Available at `http://javapapers.com/spring/dependency-injection-di-with-spring#springdi`, [Online] , Accessed: 2012-11-01.

[112] P. Kumaraguru, J. Lobo, L. F. Cranor, and S. B. Calo. S.: A survey of privacy policy languages. In *In: Workshop on Usable IT Security Management (USM 07): Proceedings of the 3rd Symposium on Usable Privacy and Security, ACM*, 2007.

[113] W. E. Khnhauser and M. von Kopp Ostrowski. A framework to support multiple security policies. In *In Proceedings of the 7th Canadian Computer Security Symposium*, 1995.

[114] M. E. Lesk and E. Schmidt. Unix vol. ii. chapter Lex&Mdash;a Lexical Analyzer Generator, pages 375–387. W. B. Saunders Company, Philadelphia, PA, USA, 1990.

[115] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a role-based trust management framework. In *Proc. IEEE Symposium on Security and Privacy, Oakland*, May 2002.

[116] N. Li and Q. Wang. Beyond separation of duty: An algebra for specifying high-level security policies. *J. ACM*, 55(3), 2008.

[117] W. H. Li, D. R. White, and J. Singer. Jvm-hosted languages: they talk the talk, but do they walk the walk? In M. Plmicke and W. Binder, editors, *PPPJ*, pages 101–112. ACM, 2013.

[118] F. v. d. Linden, editor. *Proceedings of the Second International ESPRIT ARES Workshop on Development and Evolution of Software Architectures for Product Families*, London, UK, UK, 1998. Springer-Verlag.

[119] M. Lorch, S. Proctor, R. Lepro, D. Kafura, and S. Shah. First experiences using XACML for access control in distributed systems. In *XMLSEC '03: Proceedings of the 2003 ACM workshop on XML security*, pages 25–37, New York, NY, USA, 2003. ACM.

[120] F. M.A.d.S. Codepress, online real time syntax highlighting editor. Available at `http://codepress.sourceforge.net/`, [Online] , Accessed: 2013-09-15.

[121] B. A. N. L. Maedche, Alexander. *Software for People*. Springer, 2012.

[122] B. Marchal. Sax, the power api. Available at `http://www.ibm.com/developerworks/xml/library/x-saxapi/`, [Online] , Accessed: 2011-02-27.

[123] S. Mauw, W. T. Wiersma, and T. A. C. Willemse. Language-driven system design. *International Journal of Software Engineering and Knowledge Engineering*, 14(6):625–663, 2004.

[124] J. Melton and A. R. Simon. *SQL: 1999 - Understanding Relational Language Components*. Morgan Kaufmann, May 2001.

[125] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, Dec. 2005.

[126] W. Nejdl, D. Olmedilla, and M. Winslett. PeerTrust: Automated trust negotiation for peers on the semantic web. In *Secure Data Management*, pages 118–132. 2004.

[127] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Inc, 1st edition edition, Nov. 2008.

[128] M. Odersky and M. Zenger. Scalable component abstractions. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 41–57, New York, NY, USA, 2005. ACM.

[129] T. Parr. Another tool for language recognition. Available at `http://www.antlr.org`, [Online] , Accessed: 2011-05-09.

[130] J. E. L. Peck, editor. *ALGOL 68 Implementation: Proceedings of the IFIP Working Conference on ALGOL 68 Implementation, Munich, Germany, July 20-24, 1970*. North-Holland, 1971.

[131] P. Rao, D. Lin, E. Bertino, N. Li, and J. Lobo. An algebra for fine-grained integration of xacml policies. In *Proceedings of the 14th ACM Symposium on Access Control Models and Technologies*, SACMAT '09, pages 63–72, New York, NY, USA, 2009. ACM.

[132] J. C. Reynolds. The essence of ALGOL. In P. W. O'Hearn and R. D. Tennent, editors, *ALGOL-like Languages, Volume 1*, chapter The essence of ALGOL, pages 67–88. Birkhauser Boston Inc., Cambridge, MA, USA, 1997.

[133] Rivantsov. A development kit for implementing languages on .net platform. Available at `https://irony.codeplex.com/`, [Online] , Accessed: 2013-01-30.

[134] D. Roam. *The back of the napkin: Solving problems and selling ideas with pictures*. Portfolio Hardcover, Mar. 2008.

[135] M. Rouse. loose coupling. Available at `http://searchnetworking.techtarget.com/definition/loose-coupling`, [Online] , Accessed: 2013-03-11.

[136] A. Sarimbekov, A. Podzimek, L. Bulej, Y. Zheng, N. Ricci, and W. Binder. Characteristics of dynamic jvm languages. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*, VMIL '13, pages 11–20, New York, NY, USA, 2013. ACM.

[137] A. Sarimbekov, A. Sewe, S. Kell, Y. Zheng, W. Binder, L. Bulej, and D. Ansaloni. A comprehensive toolchain for workload characterization across jvm languages. In *Proceedings of the 11th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '13, pages 9–16, New York, NY, USA, 2013. ACM.

[138] G. Schreiber and M. Dean. OWL web ontology language reference. Available at `http://www.w3.org/TR/2004/REC-owl-ref-20040210/`, February 2004.

[139] K. Seamons, M. Winslett, T. Yu, B. Smith, E. Child, J. Jacobson, H. Mills, and L. Yu. Requirements for policy languages for trust negotiation. *Policies for Distributed Systems and Networks, IEEE International Workshop on*, 0:0068+, 2002.

[140] M. Simos and J. Anthony. Weaving the model web: A multi-modeling approach to concepts and features in domain engineering. In *Proceedings of the 5th International Conference on Software Reuse*, ICSR '98, pages 94–, Washington, DC, USA, 1998. IEEE Computer Society.

[141] D. Spinellis. Notable design patterns for domain specific languages. *Journal of Systems and Software*, 56(1):91–99, Feb. 2001.

[142] R. N. Taylor, W. Tracz, and L. Coglianese. Software development using domain-specific software architectures: Cdrl a011—a curriculum module in the sei style. *SIGSOFT Softw. Eng. Notes*, 20(5):27–38, Dec. 1995.

[143] R. Tennent. Language design methods based on semantic principles. *Acta Informatica*, 8(2):97–112, 1977.

[144] D. Thomas and A. Hunt. *Programming Ruby: A Pragmatic Programmer's Guide*. Addison-Wesley Professional, 1st edition, Dec. 2000.

[145] G. Tonti, J. Bradshaw, R. Jeffers, R. Montanari, N. Suri, and A. Uszok. Semantic web languages for policy representation and reasoning: A comparison of KAoS, rei, and ponder. In D. Fensel, K. Sycara, and J. Mylopoulos,

editors, *The SemanticWeb - ISWC 2003*, volume 2870 of *Lecture Notes in Computer Science*, chapter 27, pages 419–437. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2003.

[146] E. Torreborre. A bdd library for scala. Available at `http://etorreborre.github.io/specs2/`, [Online] , Accessed: 2014-02-14.

[147] J. W. Tukey. *Exploratory data analysis*. Addison-Wesley series in behavioral science : quantitative methods. Addison-Wesley, Reading (Mass.), 1977. On spine: EDA.

[148] A. Uszok, J. Bradshaw, R. Jeffers, N. Suri, P. Hayes, M. Breedy, L. Bunch, M. Johnson, S. Kulkarni, and J. Lott. Kaos policy and domain services: Toward a description-logic approach to policy representation, deconfliction, and enforcement. In *Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks*, POLICY '03, pages 93–, Washington, DC, USA, 2003. IEEE Computer Society.

[149] D. C. Wang, A. W. Appel, J. L. Korn, and C. S. Serra. The zephyr abstract syntax description language. In *DSL*, pages 213–228. USENIX, 1997.

[150] D. M. Weiss and C. T. R. Lai. *Software Product-line Engineering: A Family-based Software Development Process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[151] D. Wijesekera and S. Jajodia. A propositional policy algebra for access control. *ACM Trans. Inf. Syst. Secur.*, 6(2):286–325, May 2003.

[152] D. Wile. Lessons learned from real dsl experiments. *Sci. Comput. Program.*, 51(3):265–290, June 2004.

[153] H. Wu, J. Gray, and M. Mernik. Unit testing for Domain-Specific languages. In W. Taha, editor, *Domain-Specific Languages*, volume 5658 of *Lecture Notes in Computer Science*, chapter 7, pages 125–147. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2009.

[154] N. S. Zalman. Making the method fit: An industrial experience in adopting feature-oriented domain analysis (foda). In *Proceedings of the 4th International Conference on Software Reuse*, ICSR '96, pages 233–, Washington, DC, USA, 1996. IEEE Computer Society.

[155] H. Zhao, J. Lobo, and S. M. Bellovin. An algebra for integration and analysis of ponder2 policies. In *Proceedings of the 2008 IEEE Workshop on Policies for Distributed Systems and Networks*, POLICY '08, pages 74–77, Washington, DC, USA, 2008. IEEE Computer Society.