An Interoperability Framework for Security Policy Languages

Amir Aryanpour, Edmond Parkash, University of Bedfordshire, UK
Andrew Harmel-Law, Scott Davies, Technology Service, Capgemini UK

Abstract— In today’s economy, whilst corporations are looking to control costs and still driving productivity, the cost of acquiring and maintaining a company’s software is under the spotlight. IT departments are under pressure to deliver more services, in shorter amounts of time, and with ever decreasing budgets; hence, IT departments are willing to invest in and choose technologies that provide more business value at a lower cost. Taking these facts into account, and bearing in mind that a number of security policy languages available and the majority of scenarios covered by them are similar, this paper proposes a framework that understands security domains and provides users with an abstract security policy language, which can be translated into the desired policy language as per framework’s configuration.

Using such a framework would allow multi-dimensional organizations to use an abstract policy language to orchestrate all security scenarios from a single point, which could then be propagated across the environment. In addition, using such a framework would help security administrators to learn and to use a single, common abstract language to describe and model their environment(s). That in turn would help IT departments to control their security related costs.

Keywords: Security Policy Language, Domain specific Language, Management of Secure Domains, Scala, Software System Design.

I. INTRODUCTION

The notion of protecting networked resources came to life at the very same moment computer networking was introduced. Similar to programming languages which facilitate programmers to orchestrate a series of actions to achieve a goal, many access control model and security policy languages have been proposed in order to address the abovementioned concern. These security policy languages, which have undergone a revolution during the last decade, usually come with different specifications, advantages and disadvantages aim to tackle different business requirements.

As an information technology expert, how many times have you heard or tried to answer a simple question, such as ‘which programming languages is most efficient, e.g. Java or C#?’ As with programming languages, it is not easy to distinguish the advantages of one specific security policy language over another but, unlike programming languages that are used to code an unlimited amount of scenarios, the majority of scenarios covered by security policy languages can usually be modeled and described at an abstract level: who can access what under which circumstances?

Taking the above fact into account through our research, we have focused on security policy languages and proposed a framework. The framework which can be controlled by security administrators using an Abstract Language, will facilitate the representation of security policy languages.

Perhaps the very first question comes to mind would challenge the usability of such a framework. In other words do policy language users want a standard? Or why not use one of the current security policy languages as the abstract language instead of introducing one? These are valid points and questions we have reviewed over and over. No doubt generic and academic languages could possibly be considered as an abstract security policy language however we need to bear in mind that the framework has a very specific goal to achieve and that is to provide current and future security policy language users with an abstract and/or standard security policy language. Adopting an existing policy language more generally would probably work for new users however that more likely would not be an acceptable approach for existing security policy language users. We believe our framework can be easily coupled with an existing security infrastructure or can be used as an independent tailored approach for a new project. We have provided even more benefits of the framework in [1].

There are number of similarities between Structured Query Language (SQL) and the Abstract Language of our framework. Similar to SQL, which hides the complexity of the underlying database infrastructure (whether it is Oracle, MySQL or another) and provides Database Administrators (DBAs) with a common and abstract language i.e. SQL to communicate with databases, our framework provides security architectures with an abstract security policy language that understands the security domain and hides the complexity thereof. Using this framework eliminates the need to learn how to code security policies in Protune and/or XACML and then code relatively common scenarios like ‘who can access what, under which circumstances?’ The proposed framework, which understands the security domains, provides users with a much simpler language that maintains the orthogonality of the security system. In other words, we provide a Domain Specific Language (DSL) for security policy languages picture (see Figure 1).

![Figure 1. Overview of proposed framework.](image-url)
In [1], we looked at the framework, the main contribution of our research, from different angles. We first described benefits of the framework in details and then re-examined works related to our research and aimed at improving them accordingly. We then theoretically proved that the translation of security policy languages, irrespective of their formalism and specification, is possible and we provided algebra to this effect. In the same paper, we selected and justified our selection of three different policy languages that work in our framework.

In this paper, we aimed to design this framework and to justify our approaches at each individual step. The rest of this paper is organized into three parts. The first part details domain-specific languages, their requirement, patterns and so on. The second part justifies the pattern we have chosen and the third part describes proposed architecture in detail.

II. SECURITY POLICY LANGUAGES

We classified security policy languages according to different categories and nominated one candidate from each category in [1], namely XACML, Ponder and Protune. We will now briefly describe each of these policy languages.

A. XACML

The eXtensible Access Control Markup Language (XACML) [2] is an OASIS standard that describes both a policy language and an access control decision request/response language (both in XML).

The policy language allows the specification of access control conditions that must be fulfilled by a requester. There are three kinds of top-level elements:

- **Rule**: It is a boolean expression that is not intended to be evaluated in isolation but which can be reused by several policies.
- **Policy**: It is a set of rules and obligations that apply to a request. It contains a set of rules and an algorithm describing how to combine the results of the evaluation.
- **PolicySet**: It contains a set of policy and policy set elements, together with an algorithm describing how to combine the results of the evaluation.

The request/response language allows the sending of queries in order to check whether a specific request should be allowed. There are four different valid values for the answer in the response: Permit, Deny, Indeterminate (a decision could not be made) or Not Applicable (the request cannot be answered by this service).

This provides the basis for the separation of the so-called Policy Enforcement Point (PEP), which is the entity in charge of protecting a resource, and the Policy Decision Point (PDP), which is responsible for checking whether a request is conformant with a given policy. In order to include the execution of actions within the standard, the authors define the **Obligation** element.

An obligation is “an action that must be performed in conjunction with the enforcement of an authorization decision”.

XACML is standard, so it includes many features, among which we highlight the following:

- The language allows the use of attributes in order to perform authorization decisions without relying exclusively on the identity of the requester.
- Different arithmetic formulae, sets and boolean operators, and built-in functions are provided, as well as a method for extending the language with non-standard functions.
- The language includes a <Target> element in each rule, policy or policy set in order to allow for indexing and to increase performance.
- Different combinations of algorithms are provided for rule and policy composition: deny-overrides, ordered-deny-overrides, permit-overrides, ordered-permit-overrides, first-applicable and only-one-applicable.
- An XACML context is defined in order to provide a canonical form for representing requests and responses. As it is encoded in XML, it is possible to extract information from the context using XPath 2.0.

Following is a security policy example written in XACML.

```
<Policy PolicyId="pol_own_records" RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:permit-overrides">
  <Description> My Policy </Description>
  <Rule RuleId="rul_own_record" Effect="Permit">
    <Condition>
      <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
        <SubjectAttributeDesignator Data-Type="http://www.w3.org/2001/XMLSchema#string">
          urn:oasis:names:tc:xacml:1.0:subject:id
        </SubjectAttributeDesignator>
        <ResourceAttributeDesignator Data-Type="http://www.w3.org/2001/XMLSchema#string">
          urn:emc:cdn:samples:xacml:resource:resource-owner
        </ResourceAttributeDesignator>
      </Apply>
    </Condition>
    <Apply>
      <Effect>Permit</Effect>
    </Apply>
  </Rule>
  </Condition>
</Policy>
```

B. Ponder

Ponder is a declarative, object-oriented language for specifying security policies with role-based access control, as well as general-purpose management policies for specifying what actions are carried out when specific events occur within the system or what resources to allocate under specific conditions [3]. Unlike many other policy specification notations, Ponder supports typed policy specifications. Policies can be written as parameterised...
types, and the types instantiated multiple times with different parameters in order to create new policies. Furthermore, new policy types can be derived from existing policy types, supporting policy extension through inheritance.

Ponder has four basic policy types: authorisations, obligations, refrains and delegations and three composite policy types: roles, relationships and management structures that are used to compose policies [4]. Ponder also has a number of supporting abstractions that are used to define policies: domains for hierarchically grouping managed objects, events for triggering obligation policies, and constraints for controlling the enforcement of policies at runtime. Shown below is an access control policy written in Ponder:

```
{ inst auth+ switchPolicyOps { 
  subject /NetworkAdmin; 
  target <PolicyT> /Nregion/switches; 
  action load(), remove(), enable(), disable() ; } } }
```

C. Protune

The PRovisional TrUst NEgotiation framework PROTUNE [5] aims at combining distributed trust management policies with provisional-style business rules and access-control related actions.

PROTUNE’s rule language extends two previous languages: PAPL, which was one of the most complete policy languages for trust negotiation until 2002, and PEERTRUST, which supports distributed credentials and a more flexible policy protection mechanism.

In addition, the framework features a powerful declarative metalanguage for driving some critical negotiation decisions, and integrity constraints for monitoring negotiations and credential disclosure. PROTUNE provides a framework that has:

- Different arithmetic formulae, sets and boolean operators, while built-in functions are also provided as a means of extending the language with non-standard functions.
- Trust management, language-supporting general provisional-style actions (possibly user-defined).
- An extendible declarative metalanguage for driving decisions regarding request formulation, information disclosure and distributed credential collection.
- A parameterised negotiation procedure, which provides semantics for the metalanguage and provably satisfies some desirable properties for all possible metapolicies.
- Integrity constraints for negotiation monitoring and disclosure control.
- General, ontology-based techniques for importing and exporting.

In Protune, a policy is a set of rules. The vocabulary of predicates occurring in the rules is partitioned into the following categories:

- **logical** predicates: usual Prolog predicates
- **provisional** predicates: predicates that are meant to represent actions

- **decision** predicates: predicates used to signal a policy's entry point

The following can be presented as an example of a policy written in Protune:

```
execute(access(resource)):-
  declaration(Uid,Pwd),
  password(Uid,Pwd),
  password(uid1,pwd1),
  password(uid2,pwd2),
  access(_)->type:provisional.
access(_)->actor:self.
declaration(_)->type:provisional.
declaration(_)->type:provisional.
>ontology:<www.L3S.de/policyFramework#Declaration>.
declaration(_)->actor:peer.
password(_)->type:logical.
password(Uid,Pwd)->sensitivity:public :-
ground(Uid).
ground(Pwd).
```

III. DOMAIN SPECIFIC LANGUAGES

Previously we have mentioned that we will utilize DSL in our framework. So it would be useful to examine the concept of DSL in more detail.

DSL are not new, having been around for decades. Fowler, in his domain specific language book, called DSL a new name for an old idea [6]. DSL, as implied by the name, is a computer programming language that has been explicitly tailored, designed and developed for a specific usage. It provides limited expressiveness and should have a clearly defined domain focus. A domain focus makes a limited language worthwhile [6].

A. DSL requirement

Before we dive into the design of the framework, let us have a quick look at the requirements, advantages and disadvantages of DSL in order to justify the use of DSL in our framework.

Generally speaking, some of the requirements for general-purpose programming languages apply directly to DSLs. The core requirements for a 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.
- **Integratability**: 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 [7].

B. Advantages of DSLs

Undoubtedly, the main advantage of using a domain-specific language in the context of our research is that DSL provides a common vocabulary for the domain experts. Using this limited vocabulary results in:

More accurate communication with Domain experts: DSLs provide a higher level of abstraction which strips out the low-level details of a programming language, hence allowing programmers to more easily engage with domain experts.

Efficient system productivity and maintainability: Domain users are using a limited programming language to communicate with each other. Abstract languages like DSLs are easy to read and understand by all a system’s stakeholders and that implies system vulnerabilities will be easier to identify. That in turn would result in better maintainability and productivity of the system. Besides developing a DSL-based application could be considered as cost efficient in long term.

In addition to above, in a large development team where a mixture of experienced and a junior programmer exists, experienced programmers can focus on design and development of the DSL – in collaboration with domain experts - leaving junior members of the team to focus on everything else. This is an efficient use of team resources.

Validation at domain Level: A general purpose language compiler does not know anything about domain concepts whereas a DSL can be configured in a way to check validity of domain constraints during the language compilation phase and avoids confusion [6][8].

C. Disadvantages of DSLs

DSL is effectively another programming language thus:

Designing a language is hard: no matter how easy and user friendly is the language, terminology design is a complex and hard task.

Expandability of DSLs is challenging: The nature of DSLs is to focus on a specific problem of a domain. DSLs usually evolve iteratively and independently.

Designing a DSL could be expensive: Although we have posited the positive aspects of productivity as a benefit of developing a DSL, designing a DSL could be expensive as the task has to be performed by experienced programmers and also involves lots of collaboration and communication with domain experts. The design of a DSL has to be financially justified first. In addition, as DSLs are not multi-purpose languages hence their development could be expensive [8].

D. DSL Patterns

Generally speaking, there are only two patterns available for implementing DLSs: Internal and External DSLs. Despite this, choosing the right implementation is not as easy as it seems.

At a very high level of definition, internal DSLs are developed on top of an existing host language like Java, Ruby or Scala. Therefore, although you are interacting with your DSL, your DSL’s commands are parsed and converted to the host language’s lines of code behind the scenes. The lines that are generated are then compiled and executed, the outcomes of which will eventually result in your desired action(s). Smart-API, Reflective Meta-Programming, Typed Embedding and many other techniques can be listed as sub-categories of Internal DSL development patterns.

With regard to external DLS development however, the DSL developer is responsible for receiving and parsing the DSL command/script and then generate and execute the code accordingly. In External DSL development, the DSL evolves as it grows. Unlike Internal DSLs, there are not too many patterns available for External DSLs, simply because the coder has to implement everything from scratch.

To put it simply, with an internal language, the developer starts with many of the host language’s features and then strips-out/hides those functionalities that are not appropriate to their users. With external DSLs, the developer starts from almost nothing and gradually adds the functionality that is needed for the DSL [6][8].

E. 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 advantages and disadvantages.

In our approach to designing this framework, we decided to use the External DSL design pattern. We justify this approach as follows:

Expressivity: Security policies can become extremely complicated; thus, the DSL should be flexible enough to cope with such requirements. As a result, Internal DSLs may not be a good choice, simply because exposing a limited level of the functionality of the host language to DSL users may not be easily achievable with internal DSLs if the correct level of DSL expressiveness is desired.

Lack of a 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. DSLs that are usually written to help and engage system domains to test the software can be presented as an example. Although the security policy languages will play an important role in our framework, we do not use/reuse their codebase in our framework.

Design freedom: In our framework, we are literally going to design a new abstract language. Thus, we need to be able to tailor the language in such a way that it is more
appropriate for the user. Parsing the DSL, error handling and DSL grammar can all be presented as examples. These, and many more features, cannot be easily achieved using Internal DSLs.

IV. PUTTING THEM ALL TOGETHER

A. Programming Languages

Choosing the correct programming language was another challenge that we faced. Although most of the available programming languages are sufficiently mature to satisfy a developer in the beginning stages, the variety of these languages makes it difficult to choose the right one. Having said that, we knew that we were going to develop an external DSL from the outset, and that helped us to narrow the options, as seen in Table 1.

Each individual column in the table would have an impact on development of DSLs and specifically on External DSLs. For instance parser combinator are a great feature (library) provided in recent programming languages like Scala [9]. By using this library developers would be able to build a structure which gets populated by the defined rules whilst DSL gets parsed by host language’s parser.

**Table 1. Programming Language to Code DSLs**

<table>
<thead>
<tr>
<th>LANGUAGE</th>
<th>STatically TYPED</th>
<th>PARSEr COMBINATOR</th>
<th>CONTEXT-DRIVEN STRING MANIPULATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scala</td>
<td>Yes</td>
<td>Yes</td>
<td>Medium</td>
</tr>
<tr>
<td>Groovy</td>
<td>No</td>
<td>No</td>
<td>Easy</td>
</tr>
<tr>
<td>C# (.Net)</td>
<td>Yes</td>
<td>Yes</td>
<td>Medium</td>
</tr>
</tbody>
</table>

As it was expected, Table 1 also shows that there is no distinctive winner amongst programming languages. Having said that it seemed Scala provided noticeably more features (libraries) compare to others languages. Admittedly, the knowledge base of our team also had an impact on our selection and resulted to choose Scala as our programming language.

B. Proposed Architecture

As mentioned previously in terms of External DSL development, the developer has to build everything from the ground up. Generally speaking, External DSL scripts are parsed by a parser and the system then creates a semantic model from the parsed tokens. The semantic model is then usually integrated into the application.

The development of an External DSL is not very different from designing and developing a completely new programming language. Fortunately, there are existing tools that make language development easier. These tools usually provide a framework that provides the developer with off-the-shelf functionalities, which can be configured to achieve the goal. Xtext [10] is an example of one of these tools.

Xtext is a framework for development of programming languages. It covers all aspects of a complete language infrastructure, from parsers, over linker, compiler or interpreter to fully-blown Eclipse IDE integration [11]. It comes with good defaults for all these aspects and at the same time every single aspect can be tailored to users’ needs.

Briefly put, when using Xtext, the language rules - or in our case, the DSL grammar rules - first need to be defined in Extended Backus Naur Form (EBNF) syntax. Xtext then uses that and utilizes ANTLR to build an e-Core semantic Abstract Syntax Tree (AST) model out of DSL script. Existing code-generators then walk through the semantic model and generate code from it.

Using Xtext, we came up with the following architecture for our framework (see Figure 2). In this architecture, the grey area is provided by the Xtext framework. Having said that, we were able to choose our code generator and, as we had already chosen Scala as our programming language, we decided to use the Oitok framework [12]. Unfortunately, we had so many problems incorporating Oitok into Xtext that we decided to replace it with Xpand[13]. Xpand is a statically-typed template language featuring

- polymorphic template invocation,
- aspect oriented programming,
- functional extensions,
- a flexible type system abstraction,
- model transformation,
- model validation and much more

However, Xpand generates Java (as opposed to Scala) code from the produced AST which effectively forces us towards a diversion from our initial design.

![Architecture of proposed framework.](image)

In a nutshell, the components used in the above structure are:

**Xtext:** as language development framework uses Eclipse Modelling Framework (EMF); hence, Xtext UI is an Eclipse IDE. **ANTLR** was chosen and **eCore** was used to
model the DSL, but both are internal to Xtext and cannot be easily changed. However, Xpand has been chosen as the Java code generator.

**Concrete DSL:** represent concrete semantic model of DSL. Typically these are the DSL Objects/classes which are used by policy generators later on. 

**DSL Artifacts:** represents DSL specific configuration like DSL grammar and configuration files.

**Framework API:** is used to connect the external policy generators to DSL framework.

**Policy Generators:** are peace of code which are responsible to generate policy specific code based on the generated concrete DSL and imposed rules enforced by DSL artifacts.

**C. Design Review and Enhancement**

Following a user-centric design approach [14], we implemented a proof of concept (PoC) of the proposed architecture and analyzed the code from different angles. The outcome can be summarized as follows.

1) **Users’ Point of View**

The main issue of the design was its dependency on the Eclipse framework. The end user is required to have both Eclipse and the Java Development Kit (JDK) installed on their machines. The look and feel of Eclipse IDE itself to a non-technical user was not very friendly and was the major issue of the design. It imposed another level of unnecessary learning curve to the end user. Not to mention installation of a new piece of software i.e. Eclipse / JDK on a secure domain environment can be pragmatically impossible as well.

2) **Developer Team’s Point of View**

The above design came with a few advantages. As an example Xtext does most of the messy and hard jobs for developers. Providing intellisense/code-assist to developers and even end-user also can be presented as another advantage. However developers also need to learn another language, i.e. EBNF syntax. Imposing such a specific language which can be used in very limited occasions is not always welcomed. Beside this we realized that the EBNF rules could get really complicated and messy when the DSL evolves.

Taking the above issues into account we improved our External DSL implementations and we came up with following design for the framework. In this new architecture we decided to walk away from the Eclipse dependency hence changed the architecture as shown in Figure3.

**User Interface:** Instead of Eclipse IDE, we used ACE [15], an iFrame-editable code editor in JavaScript. The main advantage of this component is that the end users can communicate with the system over http via their browsers.

**Parser:** As opposed to ANTLR, we used the Scala combinator parser, which is sophisticated, yet easy to use.

![Figure3. Revised architecture of proposed framework](image)

**Code Generator:** We used Scala and developed a code generator from the ground up.

**Validation Rules**, the new architecture again gave us freedom of mix and match various components into the framework. Validation rules was one of the new components we used in our new design. Using the new component security administrator provides the luxury of checking policy rules at compile time instead of runtime.

**Configuration**, Another component which was added to the system was a configuration module. In order to make different parts of system configurable yet decoupled, we have used Scala Cake dependency injection [16].

**AST:** Finally, we used AST for modeling the parsed DSL script, instead of eCore, which is used by Xtext.

Unlike enhancement of the architecture, the abstract language of our framework is going through ongoing enhancement. We have started from a very basic language and enhanced the grammar of the language through a repetitive process. As it has been mentioned before, at the moment the framework only is capable of translating the abstract language to three security policy languages that we choose in [1]. More likely expanding the framework to accepts more security policy languages would have direct impact on the abstract language.

In Appendix we have provided an example of a policy written in the Abstract language and its corresponding policy written in actual security policy language as it has been generated by the framework.

**V. CONCLUSION**

Further to our on-going research on security policy languages, we have proposed and outlined a framework for them in this paper. We have shown how security domain administrators would be able to take advantage of such a framework. In addition, we have shown that such a framework fulfills the requirements of DSL implementation. We then briefly reviewed DSL implementation patterns,
chose an appropriate one and nominated a programming language to implement the proposed architecture. Finally, we showed why and how the architecture had to be evolved in order to be considered fit for the purpose.

VI. REFERENCES


VII. APPENDIX

The following is an example of a policy written in our proposed abstract security policy language. As it appears from the example we have tried to design our abstract language a human readable language with the hope to provide a common vocabulary to all different parties involved defining a security policy.

If we configure the framework to produce Protune or Ponder policies instead, it will generate policies written on those specific security policy languages in accordance to the abstract security policy provided. The generated Protune and Ponder policies for above sample abstract security policy are removed for space conservation.