Vrije Universiteit Brussel

Faculty of Science and Bio-Engineering Sciences
Department of Computer Science
and Applied Computer Science

Expressing and checking application-specific, user-specified security policies

Graduation thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science in Applied Sciences and Engineering: Computer Science

Valentijn Spruyt

Promotor:	Prof. Dr. Coen De Roover
Advisors:	Jens Nicolay
	Quentin Stievenart

JUNE 2016

Vrije Universiteit Brussel

Faculteit Wetenschappen en Bio-
Ingenieurswetenschappen
Departement Computerwetenschappen
en Toegepaste Informatica

Expressing and checking application-specific, user-specified security policies

Proefschrift ingediend met het oog op het behalen van de graad van
Master of Science in Applied Sciences and Engineering: Computer Science

Valentijn Spruyt

Promotor:	Prof. Dr. Coen De Roover
Begeleiders:	Jens Nicolay
	Quentin Stievenart

VRIJE UNIVERSITEIT BRUSSEL • SCIENTIA VINCERE TENEBRAS •

JUNI 2016

Abstract

Acknowledgements

Contents

Chapter 1

Introduction

1.1 Context

Hallo citation [**latexcompanion**] lel.

1.2 Motivation

1.3 Objective

1.4 Overview

Chapter 2

Background

2.1 Program representations and querying mechanisms

Programs can be represented in several ways. These representations can then be queried to detect all kinds of information, such as control- and data-flow properties. In this section, three program representation approaches are presented, together some means to query those representations.

2.1.1 Exploring and Enforcing Security Guarantees via Program Dependence Graphs

2.1.2 GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code

2.1.3 Parametric regular path queries

2.2 Expressing policies using a domain-specific language

This section describes three internal domain-specific languages (*DSL*s). We present one DSL written in Java, a statically typed language, and two DSLs written in dynamically typed languages, namely Ruby and JavaScript.

2.2.1 Fluent Interfaces to a Java-Based Internal Domain-Specific Languages for Graph Generation and Analysis

Many complex systems problems manifest themselves as networks. Reasoning about these networks can be hard to do manually and asks for complex algorithms to perform sometimes even simple calculations. Hawick[5] pleads for the use of some sort of abstraction to perform graph generation and analysis, more specifically the use of an internal domain-specific language. He presents a DSL built using fluent interface techniques and the statically typed Java programming language. Common data structures and repetitive computations often offer an opportunity to abstract over them, as is the case for models based on networks and graphs.

The goal of this graph DSL is to be able to compare individual network sets to detect chatacteristic signature properties. The approach is powerful because a major set of data structures and operations on graphs can be abstracted into a library framework, which can then be used by domain experts.

The first step in setting up the DSL is to set up the common data structures. For the graph DSL, there are three: `Nodes`, `Arcs` and of course the `Graphs` themselves. The fields of these data structures are divided into several categories, as depicted in figure 2.1. Structural fields hold the main graph structure, whereas auxiliaries just exist to facilitate computations. Convenience fields contain information that might come in handy, but isn't necessarily used for computations. Finally, decorative fields just are there to have some means of presenting the data in a clear, distinguishable way.

```java
class Node{
  // Structural:
  List<Arc> inputs  = new Vector<>();
  List<Arc> outputs = new Vector<>();

  // Convenience:
  List<Node> dsts = new Vector<>();
  List<Node> srcs = new Vector<>();

  // Decorative:
  int index     = 0;
  int mark      = 0;
  double weight = 1.0;
  String label  = "";

  // Computation Auxiliaries:
  int component   = 0;
  int betweenness = 0;
  int count       = 0;
  boolean visited = false;
  boolean blocked = false;
}
```

Figure 2.1: The 'Node' data structure

Since we now have all information to perform most of the complex computations, the fluent interface can be set up. The approach used here implements the (Java) method chaining technique. This enables the cascading of methods by making each method in the fluent interface return the reference to itself, namely the *this* reference.

Most internal DSL's can be used as a standalone language, as seen in figure 2.2, but the paper also gives some examples in which the DSL is used inside the host language, such as the repetitive removal of the most stressed node in the network to investigate network robustness.

```
public static void main( String args[] ){

  Graph g = Graph.New( args )
  .setLogging(true)
  .report()
  .removeLeaves()
  .computeDegrees()
  .computeClusteringCoefficient()
  .computeAdjacency()
  .computeComponents()
  .computePaths()
  .computeBetweenness()
  .computeDistances()
  .computeCircuits()
  .report()
  .write( "composite.graph" )
  ;

}
```

Figure 2.2: Example use of the internal DSL

2.2.2 A Little Language for Surveys: Constructing an Internal DSL in Ruby

A Little Language for Surveys [2] explores the use of the Ruby programming language to implement an internal domain-specific language. It checks how well the flexible and dynamic nature of the language accomodates for the implementation of a DSL for specifying and executing surveys. Two key features of the Ruby programming languages are exploited because they especially support defining internal DSLs : The flexibility of the syntax[1] and the support for blocks[4]. Figure 2.3 shows how function calls are easily readable, since the braces surrounding the arguments can be omitted and the arguments list can consist of a variable number of arguments (The latter is also supported in JavaScript[6]). It also shows how entire blocks can be attached to method calls. These blocks are passed unevaluated to the called method, enabling *deferred evaluation*.

```
question "What is your gender?" do
  response "Female" { @female = true }
  response "Male" { @female = false }
  action { @male = if @female then false
                              else true end }
end
```

Figure 2.3: Ruby method call syntax

A handy feature in programming languages is reflexive metaprogramming. The survey DSL makes use of the following Ruby reflexive metaprogramming facilities:

obj.instance_eval(str) takes a string `str` and executes it as Ruby code in the context of `obj`. This method allows internal DSL code from a string or file to be executed by the Ruby interpreter.

mod.class_eval(str) takes a string `str` and executes it as Ruby code in the context of module `mod`. This enables new methods and classes to be declared dynamically in the running program.

obj.method_missing(sym, *args) is invoked when there is an attempt to call an undefined method with the name `sym` and argument list `args` on the object `obj`. This enables the object to take appropriate remedial action.

obj.send(sym, *args) calls method `sym` on object `obj` with argument list `args`. In Ruby terminology, this sends a message to the object.

The design of the survey language is fairly simple. A survey consists of a title, some questions, some responses and finally a result. Each of these actions have a corresponding method in the DSL.

The developers of the survey language chose to split up the parsing and interpretation logic, following the *two-pass architecture*. The first-pass layer parses the file (which is read using `instance_eval`) and generates an abstract syntax tree. These parser classes are structured according to the *Object Scoping* pattern[3], using an approach called *sandboxing*. This architecture is depicted in figure 2.4. The `SurveyBuilder` class evaluates the statements it reads from the input file using it's superclass' methods. This evaluation parses the input file and builds the AST, which is stored in the superclass as well. In this way, the object scoping pattern is applied: All calls are directed to a single object (the superclass), and global namespace cluttering is avoided. When creating the AST nodes, all blocks that were passed to the method calls inside the top-level call are stored in the AST nodes, instead of being evaluated directly. These blocks will only be evaluated in the second-pass phase (hence *deferred evaluation*). This is illustrated in figure 2.3, where the block passed to `question` is evaluated in the first-pass layer, and the blocks passed to `response` and `action` are stored in the AST, ready to be evaluated in the second-pass layer. Note that sandboxing occurs by calling `instance_eval` inside the `SurveyBuilder` object. In this way, harm can only be done *inside* this object.

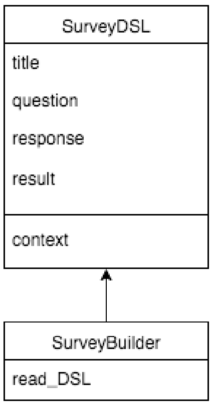

Figure 2.4: Ruby method call syntax

The actual interpretation of the created AST happens in the second-pass layer. This is a design decision which allows for different interpretation layers to be plugged in/swapped on-the-fly. However, for the survey language a simple *visitor* pattern implementation suffices to process the AST. Every AST node must provide an `accept` method which takes a `SurveyVisitor` as an argument. This is the only condition that has to be met by the interpretation layer. In this concrete example, there could be a `SurveyConsoleVisitor` and a `SurveyGUIVisitor` class, each representing the survey in their own specific way.

2.2.3 Dagoba: an in-memory graph database

2.3 Conclusion

Chapter 3

Context

3.1 Static analysis

3.2 Conclusion

Chapter 4

The JS-QL query language

4.1 The query language

4.1.1 Motivation

4.1.2 Syntax and structure

4.1.3 Defining policies

4.2 Conclusion

Chapter 5

The query engine

5.1 Architecture

5.2 Types of queries

5.3 Recursion

5.4 Conclusion

Chapter 6

Implementation

6.1 Used technologies

6.2 Design of the query system

6.3 Conclusion

Chapter 7

Evaluation

In this chapter we validate and evaluate the expressiveness of the JS-QL query language by expressing some existing security policies, described in other related work, in our own query language. We will then compare these policies in terms of expressiveness and flexibility. The concept and approach for creating a new, domain-specific language for security policies is explained in chapter 4. Chapter 5 discusses the underlying query engine and how it works together with the query language to process the application-specific policies. In chapter 6 we explain how our approach was instantiated.

We start this chapter by expressing 9 security policies distilled from 3 papers in sections 7.1.1, 7.1.2 and 7.1.3 respectively. Every JS-QL policy will be evaluated by comparing how well it matches the policy expressed in the original paper. Finally, in section 7.2, we evaluate the query framework by specifying its advantages, disadvantages and limitations. We will also briefly compare the .. presented in this chapter in terms of expressiveness, verbosity and conciseness (*LOC*).

7.1 Expressing policies

7.1.1 The PidginQL language

7.1.2 the GateKeeper language

7.1.3 The Conscript language

7.2 Evaluation

Chapter 8

Conclusion and future work

8.1 Summary

8.2 Future work

Bibliography

[1] Jamis Buck. *Writing Domain Specific Languages*. 2006. URL: http://weblog.jamisbuck.org/2006/4/20/writing-domain-specific-languages.html (visited on 04/16/2016).

[2] H. Conrad Cunningham. “A little language for surveys: constructing an internal DSL in Ruby”. In: *Proceedings of the 46th Annual Southeast Regional Conference on XX*. ACM-SE 46. Auburn, Alabama: ACM, 2008, pp. 282–287. ISBN: 978-1-60558-105-7. DOI: 10.1145/1593105.1593181. URL: http://dx.doi.org/10.1145/1593105.1593181.

[3] Martin Fowler. *Domain Specific Languages*. 1st. Addison-Wesley Professional, 2010. ISBN: 0321712943, 9780321712943.

[4] Jim Freeze. *Creating DSLs with Ruby*. 2006. URL: http://www.artima.com/rubycs/articles/ruby_as_dsl2.html (visited on 04/16/2016).

[5] K. A. Hawick. “Fluent Interfaces and Domain-Specific Languages for Graph Generation and Network Analysis Calculations”. In: *Proceedings of the International Confonference on Software Engineering (SE?13)*. Innsbruck, Austria: IASTED, Nov. 2013, pp. 752–759.

[6] Ecma International. *ECMAScript 2015 Language Specification*. 2015. URL: http://www.ecma-international.org/ecma-262/6.0/index.html (visited on 04/16/2016).