POLITECNICO DI MILANO

Facoltá di Ingegneria dell'Informazione

Corso di Laurea Specialistica in Ingegneria Informatica

Dipartimento di Elettronica e Informazione

Testing Rule-Driven Applications

Relatore:

Prof. Carlo Ghezzi

Correlatore:

Andrea Mocci

Autore:

Andrea Bonisiol - 736108

AA 2010-2011

To My Family, that supported and assisted me all the way since the beginning of my studies.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor, Dr. Mark Grechanik, for offering me the opportunity to work on this project and for his continuous support during the whole writing process.

My sincere thanks also goes to Dr. Chen Fu, that guided me through the design and implementation of our solution.

In addition I would like to thank Dr. Carlo Ghezzi and Andrea Mocci that collaborated with me to refine the thesis for the defence in Italy at Politecnico di Milano. Furthermore I want to thank my family for supporting and assisting me throughout my life.

AB

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AI	Artificial Intelligence
ANTLR	ANother Tool for Language Recognition
CFG	Control Flow Graph
DSL	Domain Specific Language
IR	Intermediate Representation
KB	Knowledge Base
KBS	Knowledge Based Systems
LHS	Left Hand Side
RDA	Rule-Driven Applications
RHS	Right Hand Side
RBES	Rule Based Expert Systems
SDG	System Dependece Graph
SET	Symbolic Execution Tree
TERLATO	TEsting RuLe-driven ApplicaTiOns
XML	Extensible Markup Language

SOMMARIO

Rule-driven applications sono applicazioni basate su una tecnologia chiamata Rule-based Expert Systems(RBESes) che permette agli sviluppatori di implementare la logica di programmazione utilizzando un linguaggio di alto livello strutturato sottoforma di regole. "Se il semaforo e' rosso l'automobile deve fermarsi" e' un esempio di regola, dove "il semaforo e' rosso" e' chiamata premessa della regola "la macchina deve fermarsi" e' chiamata conclusione o azione. I RBESes sono non deterministici per loro natura, di conseguenza si possono verificare situazioni in cui la stessa applicazione produca risultati diversi con uno stesso input. Il processo di testing di queste applicazione e' un problema complesso, dal momento che non solo richiede l'esplorazione di un alto numero di stati, ma deve tenere in considerazione la natura indeterministica di questi sistemi.

Questa tesi descrive un approccio innovativo per il TEsting di RuLe-driven ApplicaTiOns (TERLATO) che combina le tecniche di data e control-flow analysis per identificare situazioni in cui un comportamento indeterministico possa verificarsi e risultare in output diversi. Queste situazioni verranno descritte con il nome di conflitti tra regole, dal momento che si verificano in situazioni in cui piu' regole possono essere eseguite contemporaneamente, ma a seconda dell'ordine con cui vengono eseguite il risultato finale varia.

La nostra soluzione e' stata testata su un sistema attualmente in uso presso "State Farm" una delle piu' importanti compagnie di assicurazione negli Stati Uniti. Eseguendo TERLATO su un campione di piu' di mille regole, che costituiscono approssimativamente il dieci per cento

del sistema vero e proprio sono stati trovati numerosi conflitti con un tempo di esecuzione ridotto.

CHAPTER 1

INTRODUCTION

Widely used in different software applications, *rule-based expert systems (RBESes)* allow engineers to represent programming logic using rules described in some high-level declarative language (1; 2; 3). These rules are conditional sentences that relate statements of facts. *Modus ponens* is a primary inference rule, which is used by expert systems to add new facts to their facts database that represents the knowledge of the system.

An example of rule could be:

$$\text{A and B} \Rightarrow \text{C}$$

Once the premises are true, that in this case is when the two facts A and B are present in the fact database, the rule is fired and a new fact C is produced. This new fact will trigger other rules realizing a long chain of activations or firing of the rules and fact productions.

The logic that regulate the firing of rules and fact inference is managed by a specific component of rule-based architecture, called the inference engine. This component separates the execution logic of the inference engine from the high-level declarative nature of rules. It is one of the main advantages of rule-based systems, since it enables users to reason about facts and rules without worrying about low-level details of their implementation.

With RBESes, engineers and business analysts add new rules without making laborious and error-prone changes to the underlying low-level implementation that connects these rules in the

business logic of rule-driven applications that reside on top of these RBESes.

They are used by most major insurance companies and government agencies. An example of their application is fraud detection: 90% of all credit card transactions in the USA are constantly checked to identify the fraud as it takes place(4).

The market for RBES engines alone is estimated to be over $300Mil and is growing at the annual rate of 10.5%, expecting to reach $500Mil worldwide by 2013 (3).

1.1 Problem Description

The goal of this project is to implement a new technique that allows the identification of errors in a rule-based expert system.

Testing rule-driven application is wrought with many fundamentally difficult problems that stem from the inherent nature of RBESes. First, rules are often inconsistent, incomplete, and plain contradictory (5). Most literature on testing rule-based systems focuses on checking the completeness, consistency, and redundancy of rules(6). Second, a rule-driven application can contain up to k^n states if this application has k rules that are fired n times per condition change (6). The exponential number of states is even more complicated to analyze because of the inherent nondeterministic nature of RBESes, that lead to situations where the same facts result in different firing sequences in the rule-base and therefore different final states. While functions are invoked directly in programs that are written in imperative languages, rules in RBESes are fired asynchronously by RBES engines in response to facts (7).

In addition, rule engines use different technique to choose among the rules to fire when more than one rule could be fired at a given time. This means that in a situation in which

multiple rules are applicable we don't know exactly their execution sequence and consequently there might be the possibility that they interact in conflicting ways, with different outcomes. Furthermore, every rule can generate new facts in the knowledge base and other rules will be fired as a result of infer new facts. If some shared properties or variables exist between rules, the different execution sequence will not be consistent and there might be a conflict. For example, if rule 1 is: `if(PK == 3){PN = 2}` and rule 2 is: `if(PK == 3){PN == 3}`. When `PK==3` is true both rules can fire, however different sequences of their execution will result in a different value of `PN`.

Testing for these situations is a fundamentally difficult problem, since it requires not only exploring an enormous number of states, but also incorporating RBES engine methodology and timings into a testing methodology.

In reality, the situation is even more complicated. Many RBES languages include imperative constructs in addition to their declarative bases, ostensibly to enable programmers to write code at different levels of granularity, but in fact to increase the expressive power of the programmers who write rule-driven applications (8; 9; 10).

For example, programmers can define their own functions and call them from different rules, or define facts as certain values assigned to global variables. There are limitations on what programmers can do, for example, they cannot spawn threads or use synchronization mechanisms to prevent race conditions. These and other limitations are dictated by the fundamental nature of RBESes, specifically, to delegate the job of determining sequences of rule firings to RBES engines while allowing programmers to concentrate on higher-level business logic that they encode

in rules. Furthermore, the use of synchronization mechanisms, where some rules will wait on other rules, will result in defeating the purpose of the underlying RBES engines. This will also introduce significant complexity to rule-driven applications and defeat the purpose of RBESes.

1.2 Solution

In this dissertation, we want to exploit the inherent nondeterministic nature of RBESes and we want to offer a novel solution for generating tests for rule-driven applications. We combine data-flow analysis with symbolic execution to detect situations where nondeterministic behavior may lead to different results.

As the first step, we want to determine how inferring some facts may affect the execution of different rules. To do this we apply data-flow analysis to construct a data structure to store the variables that are used by a rule. Then, we try to uncover scenarios that may lead to inconsistent results when running these applications and this is done by querying the data structure we obtained. The queries lead to the identification of the values that those rules read or modify whenever they are executed. Combining these results with the detection of concurrent rules, whose premises are always verified at the same time, we are able to identify two possible conflicts that may cause faults in the system:

- Write-write conflict: when two concurrent rules access a shared or global variable and they both modify its value.
- Read-write conflict: when a rule R1 modify the value of a shared or global variable and a second rule R2 reads the same variable and R1 and R2 are fired concurrently.

After the identification we want to verify the conflict through symbolic execution. As the last step we generate tests for these applications based on the results of these analyses, and we execute these tests to confirm whether applications behave inconsistently.

In conclusion, the novel solution we propose for testing rule-based expert systems can be summarized in the following steps:

1. Identification of concurrent rules, that can be fired at the same time.
2. Performing data-flow analyses on the rule source code to identify reads and writes on variables.
3. Performing control-flow analysis combined with symbolic execution to identify the path conditions that lead to these reads and writes.
4. Detection of possible conflicts between concurrent rules.
5. Generation of a directed graph to represents the whole system.
6. Evaluation of the approach on a large-scale rule-driven application at a major insurance company.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 Rule Based Expert Systems

Rule-driven applications are widely used in different software applications in the fields of medicine, health care, financial service, and they are based on *rule-based expert systems (RBESes)* that allow developers to represent programming logic using high-level rules described in some specific declarative language. The following sections provide a background knowledge about RBESes and the problem of testing this kind of applications.

2.1.1 Expert Systems

An expert system is a software that attempts to provide an answer to a problem, or clarify uncertainties where normally one or more human experts would need to be consulted(11). Expert systems are mostly applied to a specific problem domain, and they are derived from a branch of computer science research called *Artificial Intelligence (AI)*, whose scientific goal is to build computer programs that exhibit intelligent behaviors.

These systems are also addressed as *knowledge-based systems(KBS)*, even though the term expert systems is reserved for programs who perform tasks that are usually assigned to human experts. The central component of a KBS is its *knowledge base(KB)* that informally, is a set of facts that represents the domain knowledge of the system(12) and is combined with an inference engine to simulate the process of reasoning.

Calculating the value of an equation or computing a customers bill at a shopping mall are problems that do not require an expert system, as they are all variations of a linear, deterministic process, in which each step follows inevitably from the last(13). Programs that solve these set of problems are usually characterized by a predictable linear control flow, and are often called procedural programs and they behave in a predictable way.

On the other hand, expert systems are suited for problems where there are more decisions than actions, such that the control flow includes many loops and branches and it would be almost impossible to write instructions covering every possible situation(13).
That is why in these situations, declarative approach is the best choice: the program is provided with the set of sentences that model the KB and the reasoning is separate from the knowledge. In other words, the programmer describes what the computer should do, without specifying how to do it.

Expert systems are used when the problem area is complex enough such that a traditional procedural approach is insufficient or too difficult to realize. They can include different types of reasoning like rule-based, fuzzy logic, probabilistic reasoning, neural networks and bayesian networks. This project deals with a Rule-Based Expert System (RBES) therefore the following section will provide a more detailed description of this technology.

2.1.2 Rule-based approach

A rule-based system is defined by a system that uses rules to derive conclusions from premises(13). Expert systems are usually rule-based systems, as they use rules to represent

the knowledge they have about the domain. A rule is a kind of instruction or command that applies in certain situations (e.g. No crossing with red light).

Rules have the same structure as if-then statements of traditional programming languages and are formed by two parts: an "if" clause and a "then" clause. The "if" part of a rule is usually called its *left-hand side(LHS)*, predicate, or premises while the "then" part is the *right hand side (RHS)*, actions, or conclusions(13). This is a declarative approach, in fact rules specify what the program should do, however it is not defined how it will reach the conclusion from the premises. This task is performed by another program, the rule engine or inference engine, which determines the rules to apply at any given time and executes them. A rule engine is like an empty box, as it does not contain any rules until they are inserted by the developers, it only knows how to follow rules and the inference process.

The inference process can be performed in two ways: forward-chaining, also known as data-driven reasoning, or backward chaining also called goal-driven reasoning. In forward chaining, the initial facts are processed first, and use the rules in the KB to draw new conclusions until a goal is reached. In backward chaining, the rule engine starts from the hypothesis(or goal) we are trying to reach and keep looking for rules which allow to conclude to that hypothesis (14) in a backward fashion.

The main components of a rule-based expert system are illustrated in Figure 1. The rule base contains all relevant information, data, rules, that represent the basic knowledge of the system. The working memory contains all the facts asserted by the inference engine starting

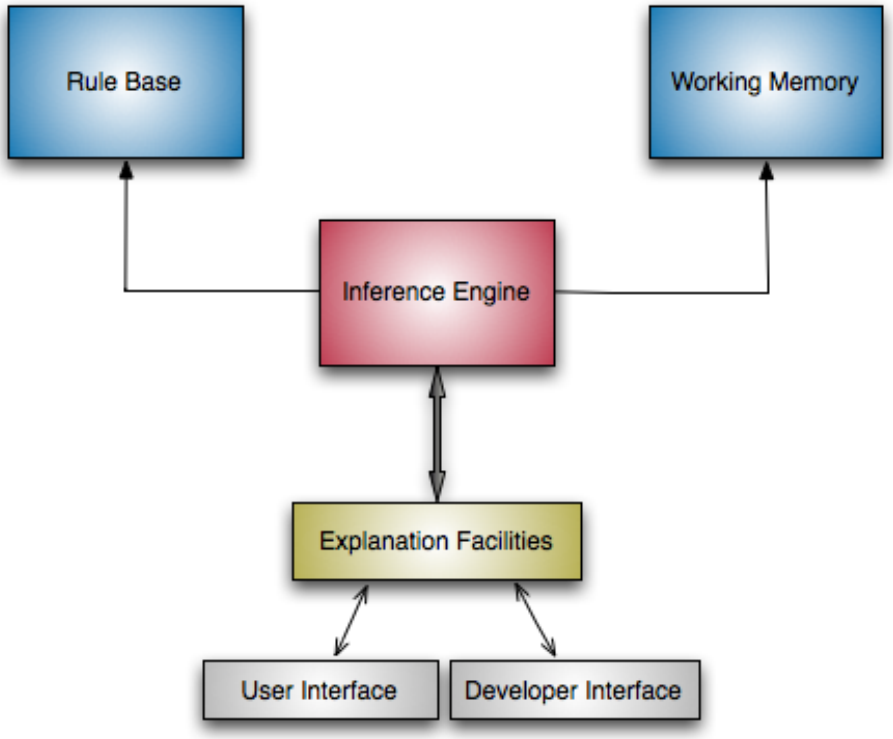

Figure 1. Expert System architecture

from the rule base. The inference engine has the role to find the right facts, and combine them with the rules to assert new facts and store them in the working memory.

2.1.3 Testing rule-based expert systems

The task of testing rule-based expert system is generally characterized by the absence of clear specifications. This is due to the nature of the system that relies on the subjective opinion of a human expert, therefore the verification of the system is frequently based on a

comparison between the system and the domain expert, rather than the implementation and the specification.

Current attempt to verify rule-based expert systems focus on the concerns of *Conflict* and *Incompleteness*(15). Conflicts arise when rules have premises that may hold at the same time, and the consequences of the rules generate different results. Incompleteness appears in the situation in which none of the rules in the knowledge base can fire.

We are more interested in the first concept of conflict, that is caused by nondeterminism: if two rules can fire at the same time it's impossible to predict which one will be executed first. To verify that an action occurs it is necessary to examine a large number of components since different sequences of rule firing could perform to the specific action but lead to a different state. Therefore in this thesis we focus the attention on this problem and we propose a way to identify possible conflicts that can arise from the implicit nondeterminism of rule engines, that can eventually manifest as faults of the system.

CHAPTER 3

OVERVIEW OF OUR SOLUTION

3.1 The Problem

In this section, we introduce a language model that is widely used for programming *rule-driven applications (RDAs)*, discuss its features, specify sources of conflicts, their resolutions, and effects on reliability of RDAs, show an illustrative example that demonstrates the problem that we address in this paper, and give the problem statement.

3.1.1 A Language Model

Early languages for RBESes were purely declarative (16), however, over years, tight integration of RDAs with legacy systems made vendors mix imperative and declarative constructs in languages for RDAs (3; 9). Most prominent examples of open source RBESes with mixed languages are CLIPS[1] and JESS[2] where functions written in C and Java methods are invoked from declarative rules. Commercial RBESes with mixed language constructs include BizTalk by Microsoft Corp. and Fusion by Oracle Corp. Key elements of a language model for programming RDAs include asserting facts, triggering rules, invoking system services, and sharing data among different rules.

---

[1]http://clipsrules.sourceforge.net

[2]http://www.jessrules.com

3.1.2 Asserting Facts

Facts are asserted by assigning values to properties, where properties are RDA variables that represent certain physical elements of a system. For example, assigning the value three to the property CONDO_RESIDENTS asserts a fact that there are three residents in a condo. Properties are declared externally to the scopes of specific rules, and rules can manipulate properties by assigning values to them and reading their values. Properties can be declared read-only, in which case their values are assigned once and stay immutable over the lifecycle of the RDA. In large RDAs, properties are grouped into hierarchically organized packages, where branches of these packages represent separated logical concerns of the system. Parent-child relationship between properties is generally used to designate different levels of abstractions between concepts that these properties describe, however, in some cases it is used to specify the containment relationship, where a property may be a part of a form, for example.

Consider a rule from the RDA Renters at a major insurance insurance company that is shown in Figure Figure 2. Even though this rule is slightly modified to remove sensitive information, it is representative of rules that we worked with in this and other RDAs. The header of the rule is located in lines 1–5, where line 1 links the rule to its description in the requirements specification, line 2 gives the description of the rule in plain English, line 3 specifies the name of the rule, and line 4 designates the start of the section where properties are declared in line 5. The property is specified by its full name where the parent properties are separated from their children by period. When the value of this property is set to `true`, the rule is fired by the RBES engine. The body of the rule is defined in lines 6–15, where input parameters `Container1` and

```
1   RULE SPEC ID: SVFF506D897
2   RULE LONG NAME: Delete Potential Applicant
3   SYSTEM NAME: DLTE_CONTAINERS
4           PROPERTIES:
5           APPLY_FOR_POLICY.POTNL_APLCTS.POTNL_APLCT
6   SOURCE
7           Receives
8                   Container1 Boolean
9                   ContainerArray() Boolean
10          uses i integer
11                  Container1 = false
12          for i = 1 to ubound(ContainerArray)
13                          ContainerArray(i) = false
14                  next
15          CONDO_RESIDENTS = 0
```

Figure 2. An example of a rule from Renters RDA.

`ContainerArray` are specified in lines 8–9, and the code in lines 11–14 sets the values of these parameters to `false`. The value of the property CONDO_RESIDENTS is set to zero in line 15.

3.1.3 Rule Firing Conflicts

When conditions of two or more rules are satisfied, these rules are fired, and during execution of the bodies of these rules more conditions are satisfied, which leads to firing other rules until there are no more rules whose conditions are satisfied. Enterprise-strength RDAs contain tens of thousands of rules, many of which are fired at any given time since their conditions are satisfied simultaneously during execution of RDAs. The set of rules whose conditions are satisfied at any given time is called the *conflict set*. RBES engines employ different strategies for conflict

resolution, that is for selecting rules for firing from the conflict set. Conflict resolution strategies can be classified into the following categories (16, pages 85-87).

Refractoriness: rules are not allowed to fire more than once on the same data. Using refractoriness, loops can be prevented, however, it may lead to undesirable situation when the same rule must be fired on the same data multiple times by design.

Randomness: rules are chosen to fire at random. This is a popular mechanism since it is fast and prevents RBESes from favoring some rules over the others. However, it makes it difficult for programmers to reason about the behavior of RDAs.

Recency: rules are ranked higher if they use data that were most recently created or modified in memory, thus following the "leading edge" of the computation. Depending on the timing of updating data, different rules may be fired for executions of RDAs with the same input data.

Specificity: preference is given to more specific rules, that is, those rules that have a greater number of conditions and are more difficult to satisfy. Of course, when there are two or more rules that have the same number of condition, resolving conflicts must be done using a different strategy.

3.1.4 Nondeterminacy and Loss of Reliability

Nondeterminacy occurs when different results are obtained from executions of the same program with the same input values. In RDAs, nondeterminacy arises when different rules

from conflict sets are fired in different orders as a result of different timings in asserting facts or using conflict resolution strategies.

Consider a situation where the values are set of two properties P_n and P_k, leading an RDA to eventually execute rules R_n and R_k, that is $P_n \rightsquigarrow R_n$ and $P_k \rightsquigarrow R_k$. Between setting the values of these properties and executing rules R_n and R_k, many other properties are written to and other rules are fired as it is indicated by using the squiggly arrows $\rightsquigarrow$. Suppose that the rule R_n contains code fragment `if(PROPERTY_CONDO_RATE>0.5){NEWRATE=true}` `else{OLDRATE=true}` and the rule R_k contains the assignment `PROPERTY_CONDO_RATE = 0.1`. Depending on the order in which these rules are fired, either the property `OLDRATE` or `NEWRATE` will obtain the boolean value `true`, resulting in a different result of the execution, thus leading to nonreliable behavior of the RDA.

We classify conflicts that should be caught with our approach into the following general categories:

- *Read-Write (RW)* errors occur when a fired rule reads the value of some property that may be written to by some other fired rule concurrently.
- *Write-Write (W^2)* errors occur when fired rules can assign different values to the same property concurrently.

These two kinds of errors are common in concurrent multithreaded programming, and a common practice among programmers to prevent race conditions is to synchronize accesses to shares resources. However, using synchronization mechanisms is not an option in RBESes where the engine determines the order of firing rules. While existing work concentrates on

different aspects of reliability of RBESes, such as correctness of reasoning, performance, and the explosion of the number test cases (6; 17; 18), we concentrate in this paper on the conflicts that lead to nondeterminacy and nonreliable behavior of RDAs.

3.1.5 The Problem Statement

Our goal is to determine situations with a high degree of automation and precision where RDAs exhibit nondeterminacy that leads to unreliable behavior. The problem is to find and report some situations at compile time in which rules can be fired leading to RW and W^2 conflicts. Currently, no tool checks RDAs for these conflicts.

We do not attempt to make our approach either sound or complete. A sound approach ensures the absence of conflicts in RDAs if it reports that no conflicts exist and all reported conflicts are indeed present in RDAs (i.e., no false positives), and a complete approach reports all conflicts or no conflicts for correct rules. Our approach should detect RW and W^2 conflicts with high automation and good precision, and it should output test cases whose execution will lead to these conflicts, and test personnel should be able to follow these executions to validate the reported conflicts.

3.2 Our Solution

In this section, we discuss key ideas of our solution, explain the architecture of Terlato, and describe how Terlato can be used.

3.2.1 Key Ideas

Our key idea of finding RW and W^2 conflicts in RDAs is threefold. First, we locate uses of properties in rules. Second, we specify all possible RW and W^2 conflicts by matching corre-

sponding reads and writes for all properties. Naturally, there will be many false positives, since this basic analysis does not take into consideration that many rules cannot be fired simultaneously.

To overcome this limitation, we perform symbolic execution of each rule to determine conditions under which certain statements and expressions can be reached. These statements and expressions include those where conflicts are located. While symbolic execution suffers from exponential complexity, this problem can be tamed well since rules are small.

Once path conditions to conflicts are computed, we construct a graph of rule dependencies where nodes specify conflicts in rules and edges specify firing sequences between rules. This graph will be traversed to input properties so that test cases can be generated automatically. Executing these test cases will verify whether conflicts are possible.

3.2.2 TERLATO Architecture and Workflow

The architecture of TERLATO is shown in Figure 3. Solid arrows show command and data flows between components, and numbers in circles indicate the sequence of operations in the workflow. The input to TERLATO is the set of rules of the RDA, and it is specified with the thick arrow labeled `(1)`. The output of TERLATO is the test suite that is specified with the arrow labeled `(10)`.

Even though rules come in different language flavors, their basic structure makes it easy to translate into Java-based code using a language translator `(2)`. This is the only component of TERLATO that should be customized for different RBESes that use different rule languages. It is much easier to translate these languages into Java-based code and use facilities for control

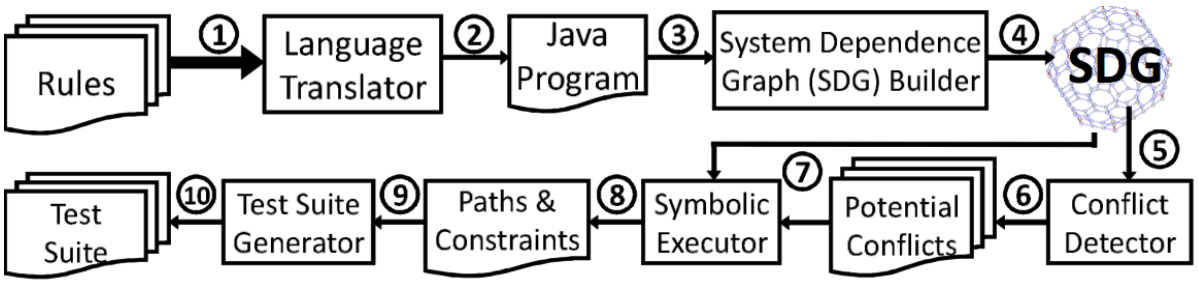

Figure 3. Terlato's architecture and workflow.

and data-flow analyses than to build these analyses libraries for each RBES language. Next, the translated Java program is supplied to the *system dependence graph (SDG)* builder `(3)` to construct the SDG of the RDA `(4)`.

SDG is a graph that contains subgraphs for all procedures in which nodes are statements or expressions, data dependence edges represent flow of data between nodes, and control dependence edges represent conditions on which execution of these nodes depend (19). SDG is supplied `(5)` into the conflict resolution component that outputs `(6)` the list of potential conflicts. This list is supplied as the input `(7)` to the symbolic executor along with the SDG. Symbolic executor outputs `(8)` the list of constraints for the paths that lead to potential conflicts. This list is processed `(9)` by the test suite generator that outputs `(10)` a test suite that is executed by personnel to verify the detected conflicts. This thesis focuses on the steps 1 to 9 that result in the identification of potential conflicts and a graph data structure that describes the relationships between rules and the conditions that leads to the conflicts found.

3.2.3 Symbolic Execution

Symbolic execution is a path-oriented evaluation method that describes data dependencies for a path (20)(21)(22). Program variables are represented using symbolic expressions that serve as abstractions for concrete instances of data that these variables may hold. The state of a symbolically executed program includes values of symbolic variables. When a program is executed symbolically its state is changed by evaluating its statements in the sequential order.

Historically, symbolic evaluation is used for analyzing and testing programs that perform numerical computations. We illustrate it on a simple example. Consider two consecutive statements `x=2*y` and `y=y+x` in a program. Initially, variables `x` and `y` are assigned symbolic values `X` and `Y` respectively. After symbolically executing the first statement, `x` has the value `2*Y`, and after executing the second statement the value of `y` is `Y+2*Y`. When symbolically executing numerical programs, variables obtain symbolic values of polynomial expressions.

Symbolic execution trees (SETs) are graphs characterizing the execution paths followed during the symbolic executions of a program. Nodes in these graphs correspond to executed statements, and edges correspond to transitions between statements. Each node in the SET describes the current state of execution that includes values of symbolic variables and the statement counter. Nodes for branching statements (e.g., `if` or `while` statements) have two edges that connect to nodes with different condition predicates.

CHAPTER 4

IMPLEMENTATION

4.1 ANTRL

ANother Tool for Language Recognition(ANTLR) is a language tool that provides a framework for constructing recognizers, interpreters, compilers, and translators from grammatical descriptions containing actions in a variety of target languages(23).

One of the main uses of ANTLR is to deal with domain-specific languages(DSLs), that are generally high level languages devoted to specific tasks(24). Rule-based systems make use of a DSL to code the rules in the knowledge base, that is why they are a good example of domain where ANTLR is useful.

The syntax and semantic of the rule language is mostly specific of the system and most of the applications used in the industry utilize proprietary languages. When dealing with this kind of systems, a translation is necessary to perform testing, as testers need to use automatic tools, that are usually compatible with the best known languages. This is the reason why ANTLR was chosen to be used in this research to build a parser for a proprietary language, extract information from it and translate it into Java. The translated code was then used to perform data flow analysis using Soot, a tool that will be described in detail in the next section.

4.2 Parser

The rule-based system we utilized for testing our approach uses a proprietary language for declaring the rules and defining the rule body, therefore the first component that we had to implement was a parser.

Tools for data and control flow analysis, that are available on the market, are usually compatible with the most popular programming languages, that's why translating this proprietary language into one of them was a necessary step for applying our solution.

The language we chose to use for our analysis is Java. We made this choice because the market offers a wide range of tool for analyzing Java code and, more specifically, it offered the tool that we selected to perform our data and control flow analysis,that will be described in details later. The documentation of the system we analyzed was comprehensive of a grammar, which was written using ANTLR, therefore we developed our parser starting from it.

Before describing the components of the parser, we want to show an example of how a rule is coded in the system(Figure 4). As we can see a rule is defined using a personalized language and is mainly divided into two components:

- *Rule header*, that defines the rule by specifying its name, relationships with other rules in the system, inputs and outputs and other informations that we are not considering in our analysis.
- *Rule body* is the actual source code of the rule that is run when the rule is fired and contains normal programming language statements.

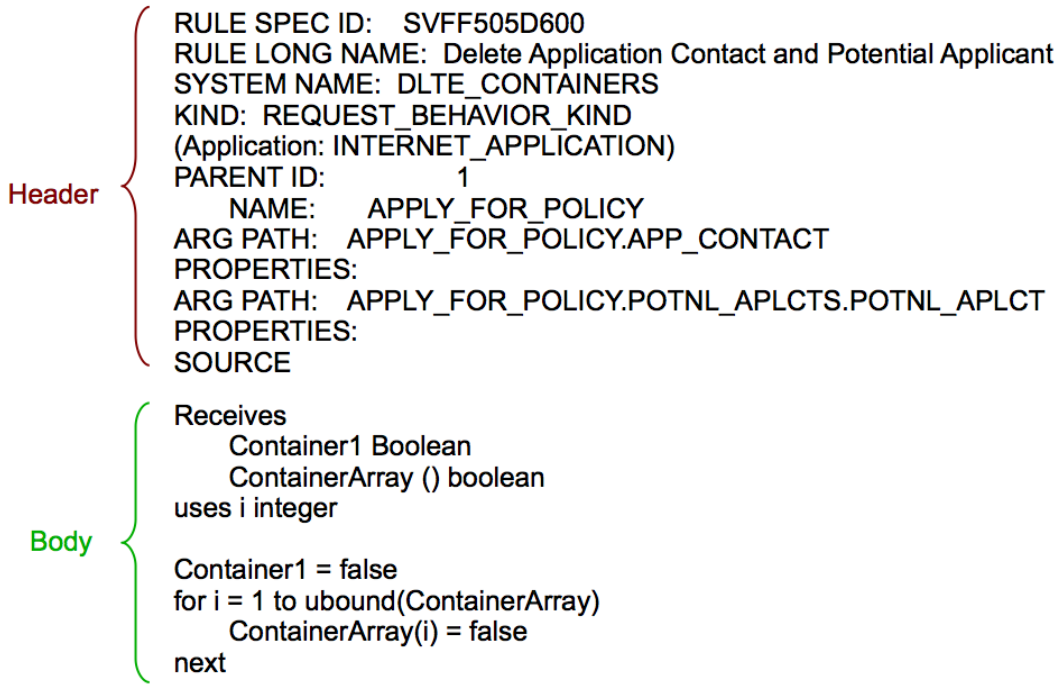

Figure 4. An example of Rule

This grammar dealt with the part of the rule that we defined as *Rule Body*. Not only did we extend the parser to manage the rule header part, but we also had to modify the grammar we were provided. This was probably due to the fact that when the system was modified the documentation wasn't updated to reflect the changes.

The parsing process that we applied to the rules can be summarized in two operations:

1. Performing a translation of the *Rule body* into Java code, and more specifically into a Java method.

2. *S*aving the information contained in the *Rule header* into a data structure, that will be used in the next steps of the analysis.

The following sections will describe in detail the implementation of these two operations.

4.2.1 Rule Header

The *Rule Header* contains the information to identify the specific rule inside the system and the relationship between this rule and the other rules.

The objective of parsing the rule header is to store the information of the rule in a data structure, so that this data can be reused during the next phases of the project.
As shown in Figure 4 the header consists of many fields, but the ones that we want to save are:

- RULE KIND: defines the type of the rule, and this type influences how the rule will be triggered by other rules. A *POST_SET_KIND* rule, for instance, will be fired once the property attached to the rule is written by another rule or by an external event.
- PARENT ID and NAME: they identify a property to which the rule is attached to, and this property together with the kind influences how this rule will fire.
- ARG PATH and PROPERTY: they define the properties that this rule will use when it is fired and their full path within the system.
- RULE SPEC ID, RULE LONG NAME, SYSTEM NAME are only used to identify the rules and we used it to grant the unicity of the java method we translate the rule in.

What we wanted to do is to store all this information that defines the rule in a data structure, to use it in our conflict identification. The two possible approaches that we evaluated were to

save the information in a database, or to use an XML file. A database is more efficient and fast in managing queries to the data, while saving XML strings in a file is simpler in terms of implementation. For our problem the interrogations we needed to perform on the data were very simple, therefore our choice was to use an XML based data structure.

This is an example of the rule header data that are stored:

```
<rule>
  <ruleSpecId>LFZ7P23F000</ruleSpecId>
  <ruleLongName>RULE LONG NAME:  Apply for Policy Read </ruleLongName>
  <systemName>APPLY_FOR_POLICY_READ</systemName>
  <kind>IS_READ_ONLY_KIND</kind>
  <parent id="1" name="APPLY_FOR_POLICY"/>
  <argpaths>
    <argpath path="APPLY_FOR_POLICY">
      <properties>
        <property>STATUS</property>
      </properties>
    </argpath>
  </argpaths>
  <source>1720</source>
  <methodName>
  LFZ7P23F000APPLY_FOR_POLICY_READAPPLY_FOR_POLICYAPPLY_FOR_POLICY
  </methodName>
</rule>
```

To realize the data structure we implemented a grammar using ANTLR that could parse all the fields in the rule header and we saved the information as a *String* variable xmlString. The variable contains the XML expression for each field and is returned by each production rule until the whole header is parsed.

Some special fields were added such as the length of the source code and the *methodName*. The latter is used to identify univocally a rule, and it is obtained by concatenating RULE SPEC ID, SYSTEM NAME,PARENT NAME and ARG PATHs. Finally, the xmlString variable is printed to a file. This is an example of production rule that store the header information :

```
systemNameDeclaration returns[String xmlString]:
  {$xmlString = "<systemName>";}
  SYSTEM_NAME (INTEGER_LITERAL{$xmlString += $INTEGER_LITERAL.text;})?
    (IDENTIFIER{ String s = "";
    s = removeChar($IDENTIFIER.text,':');
    s = removeChar(s,'/');
    $rule::methodName += s;
    $xmlString += $IDENTIFIER.text;})?
  {$xmlString += "</systemName>\n";}
;
```

This example shows how the system name is processed and saved in a grammar rule.
As we can see *$xmlString* is the string that will contain the xml code, while *$rule::methodName* is the variable that contains the methodName field in the data structure. SYSTEM NAME is

concatenated to it after some preprocessing to eliminate character that are not allowed as a Java method name.

4.2.2 Rule Body

The *Rule Body* represents the real source code of the rule and uses a Basic-like language to define what the rule does when it is fired by the rule engine. Our objective is to parse the source code into Java language to make it available for data and control flow analysis and more specifically, a every rule is translated into a Java function. Going into the details of the mapping of the variables between the rule body and the Java method we can see that at the beginning of rule source code we can find three declarations:

- **Receives**: defines the parameters that are provided as inputs from the rule engine.
- **Returns**: can be present or not and defines the return variable of the rule after its execution.
- **Uses**: declares the local variables and their types used inside a rule body.

A Java method offers the same structure as we can distinguish:

- **Parameters**: that are written in the method declaration.
- **Local variables**: that are declared within the method body.
- **Return variable**: defined in the method declaration as well.

As a result the mapping between rule variables and the Java method is defined as follows: if we have a rule R that **Receives** two variables *v0*, *v1*, **Returns** a variable *r* and **Uses** two

variables *l0*, *l1* the correspondent Java method will have return type defined by the type of *r*, will receive as parameters *v0*, *v1* and *r*, and inside its body we will find the declaration of the local variables *l0*, *l1*.

Figure 5 shows an example of rule and the correspondent Java method produced as an output.

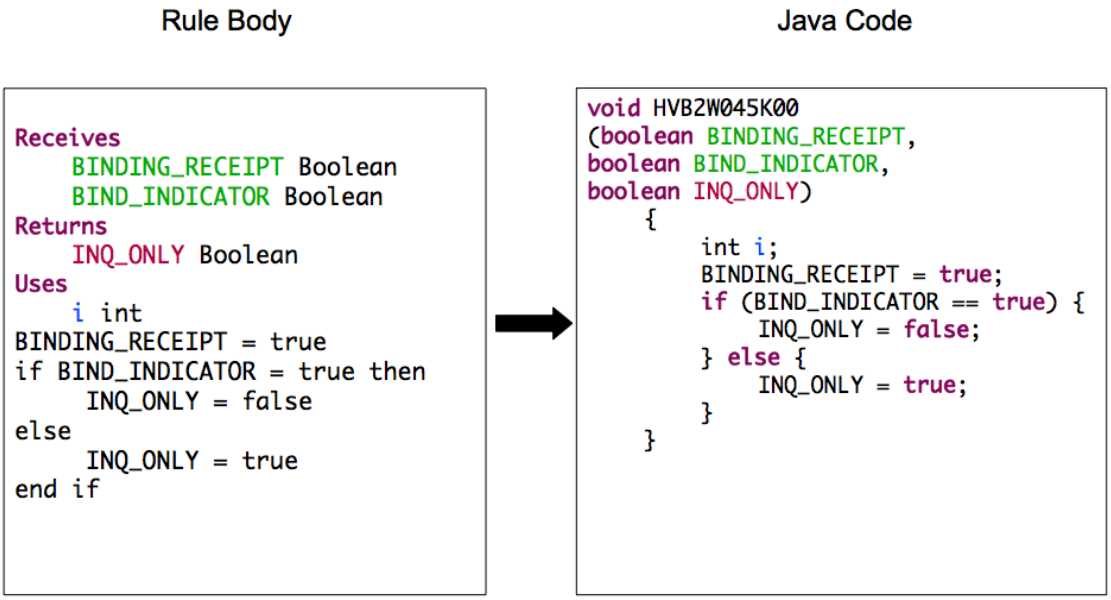

Figure 5. Variable mapping from a rule to a java method

Now let's deal with the statements in the rule source code and their translations into the method body. The kinds of statement used by this proprietary language are very similar to common imperative languages ones. We can find assign statements, conditional statements,

loop statements, switch statements, function call statements and return statements. These statements have a one to one correspondence with Java statements, therefore we will not provide the details of their translation into Java. The major issues arise in fact from the management of types and their use in expressions. The types used by this proprietary language are quite common but to be precise we will provide the list of all the possible types and their correspondent Java ones (Table I).

TABLE I

TYPE TRANSLATION.

Rule language	Java
Boolean	boolean
Currency	float
Float	float
Date	java.util.Date
Integer	int
String	String
Percentage	java.lang.Float

As we mentioned before, the main issues are related to the translation of the expressions, that is dependent on its type. To deal with this problem, we introduced an attribute type for every expression in the ANTLR grammar, and we had to implement a *Symbol Table*.
The symbol table is a data structure used by language translators to keep track of each variable

of function and contains information relating to its declaration or appearance in the source(25). This is the data structure implementation we chose:

```
public class SymbolEntry {
  private String variableName = "";
  private Type type;
  private boolean isProperty;
  private int dimension;
}
public class SymbolTable {
  private ArrayList<SymbolEntry> symbolTable;
  ...
  public Type findEntryType(String variableName) {
      Type type = Type.NULL;
      for (SymbolEntry s : symbolTable) {
        if (s.getVariableName().compareTo(variableName) == 0) {
          // System.out.println("FOUND");
          type = s.getType();
          break;
        }
      }
      return type;
  }
}
```

Furthermore, for what concerns the functions used within the code, it is not allowed to declare new function inside the rule body, but we were provided a list of allowed function and their return type that can be seen in Table II.

The data structure used to keep track of the functions is another symbol table, with the only difference that its entries are initialized to the function list mentioned before.

After initializing the function and the symbol tables (that is complete after parsing the three declarations mentioned above), we can obtain the type of every kind of expression. The type is returned as an attribute for every kind of expression contained in the code, with an exception with conditionalExpression.

The purpose of propagating the type is not to perform type-checking on expression, but to generate correct Java code, as the semantic of the two languages is different. In fact, one of the major issues was dealing with the relational expressions: the rule language uses the relational operators(>, >=, <, <=, =, <>) with every data type, while Java just allows this with primitive types.

Consequently the grammar rule that deals with the relational expression checks the types of the two operators and then decides how to perform the parsing: if the operators type is not a primitive one, that is in the case of String, Date or Percentage types, the comparison is realized through the function *java.lang.compareTo(Object o)*.

The following code fragment shows how in presence of the "NOT_EQUAL" operator we perform a check to see whether the type of the first operator was also a primitive one or not and perform the right action.

TABLE II

FUNCTION LIST.

Function Name	Return type
dateAdd	DATE
date	DATE
dateDiff	INTEGER
dateSerial	DATE
dateValue	DATE
day	INTEGER
month	INTEGER
year	INTEGER
getDate	DATE
getTimestamp	STRING
left	STRING
mid	STRING
right	STRING
trim	STRING
addTime	STRING
trimLeft	STRING
trimRight	STRING
replace	STRING
ubound	INTEGER
nextDimensionElementCount	INTEGER
elementCount	INTEGER
round	FLOAT
isNull	BOOLEAN
initializeValue	NULL
error	INTEGER
message	INTEGER
attachError	INTEGER
attachMessage	INTEGER
getGlobal	STRING
getGlobalInt	INTEGER
getGlobalFloat	FLOAT

```
NOT_EQUAL {
        isNotEqualFound = true;
        if(!isFrontString && !isFrontDate && !isFrontPercentage)
          $javaString += "!=";
        else{
          //call to method compareTo
          $javaString += ".compareTo(";
        }
    }
```

After checking that the second operator is also compatible with the first one(this is partially type-checking) the parser adds the right relational operation to conclude the expression.

```
if(isFrontString||isFrontDate||isFrontPercentage){
    $javaString += ")";
    if(isNotEqualFound)
      $javaString += "!= 0";
    else if(isAssignFound)
      $javaString += "== 0";
    else if(isLtFound)
      $javaString += "< 0";
    else if(isLeFound)
      $javaString += "<= 0";
    else if(isGtFound)
      $javaString += "> 0";
```

```
    else if(isGeFound)
      $javaString += ">= 0";
  }
```

This works with both *java.lang.String*, *java.lang.Float*, *java.util.Date* and didn't require any method overriding.

Finally, as we did with the rule header for the XML code, the Java code is saved as a string and returned at the end of the parsing. Every production rule returns a String attribute called javaString that contains the code.

4.3 Soot a Java Bytecode Optimization Framework

Soot is a framework for optimizing Java bytecode, realized by the Sable research group from McGill University. The framework is implemented in Java and provides four different Intermediate Representations (IRs) for analysis and optimization purposes. Each of the IRs have different levels of abstraction that give different benefits when analyzing, but the only one that was used for our analysis was Jimple, a typed three-address intermediate representation usually used for improving the code performance(26).

An example of Jimple code is the following.

```
private final int nextDimensionElementCount(boolean)
  {
      test.test this;
      boolean b;
      int temp$0;
```

```
    this := @this: test.test;
    b := @parameter0: boolean;
    temp$0 = 0;
    return temp$0;
}
```

Soot can be used as a standalone tool to optimize java bytecode, but the functionality that was useful for this research is the ability of realizing intra-procedural data flow analysis. To represent the function blocks, soot data-flow framework uses as structure any form of control flow graph(CFG) implementing the interface empfsoot.toolkits.graph.DirectedGraph(26). Performing a data flow analysis using soot can be described mostly in five steps:

- Choose the type of analysis that can be forward analysis, backward analysis.
- Decide whether it is a may or a must analysis, or in other words implement the function that merges information.
- Defining the equations that represent the flow through the graph.
- Decide the initial state of the entry node and of the inner nodes, generally the empty set or the full set, depending on the analysis.
- Saving result analysis in a data structure or file for future uses.

Within the framework we can already find three analysis implementation: *ForwardFlowAnalysis*, *BackwardFlowAnalysis* and *ForwardBranchedFlowAnalysis*. The first one starts from the

start node and follows the control flow graph until it reaches the exit node, while the second one behaves the same way, but starting from the exit node and going backward to the start one. On the other hand, ForwardBranchedAnalysis enables us to perform an analysis that depends on the branch nodes, and propagates differently the information to the branches. The second step is to decide the approximation level by defining how to join sets in case of merge nodes and how to propagate information between consecutive lattice elements. The implementations mentioned above are already provided with methods to do this:

```
protected void merge(Object in1, Object in2, Object out) {
  //personalized implementation
}

protected void copy(Object source, Object dest) {
  //personalized implementation
}
```

By overriding these two functions, we can provide our own implementation for the "join" of two sets and the propagation of information within the graph. Additionally, to perform the generation or elimination of information through the analysis, we need implement the *flowThrough()* function:

```
protected void flowThrough(Object in, Object node, Object out) {
  kill(in, u, out);
  gen(out, u);
}
```

This is the main function that deals with the information flow within the nodes: the object *in* represents the information before the node, *out* the information after. Furthermore, *out* will contain the information in *in* plus the information generated by *gen()*, excluding the one removed by the function *kill()*. The last step is defining the initialization of the entry point and of each node of the graph, and this requires us to override:

```
protected Object entryInitialFlow() {
  return entrySet;
}
protected Object newInitialFlow() {
  return initialSet();
}
```

The most common initializations are an empty set or a universal set but personalized ones can be implemented as needed.

After this background section that introduced the framework used in this project we will present the two analyses implementation that we realized: *Read Write Analysis* and *Path Condition Analysis*.

4.4 Read and Write analysis

After parsing the rule code into Java, that was a necessary step, the main part of the solution we propose to test rule-based expert systems is data-flow analysis. More precisely we are interested in discovering what parameters each rule read and write, to identify the possible conflicts.

Starting from the Java method we want to apply and intra-procedural analysis that will result in the identification of four categories of interaction between the rule and the variable. We define the interactions as :

- **Write** if the function modify the value of a parameter, and the instruction where the write happens is executed for sure.
- **Read** if the function uses the value of a parameter, and the instruction where the read happens is executed for sure.
- **May Write** if the function modify the value of a parameter, and the instruction where the write happens could be executed.
- **May Read** if the function uses the value of a parameter, and the instruction where the read happens could be executed.

To identify these four sets of variables, we use four different implementation of *ForwardFlow-Analysis*, one for each set. First, we want to explain what we assumed to be a write and a read in the code. A *write* can be found in two kinds of statements:

- **Assignment Statement** that is written in the form $A = B$; in this case the variable A is written with the value contained in B.
- **Return Statement** which appears as *return C*. Variable C is therefore said to be written because of the mapping hypothesis that we made that is C is passed as a parameter to the java method and a return overwrite that value.

A *read* can be instead found in a:

- **Assignment statement** that is written in the form $A = B$; in this case the variable B is read.
- **Function call statement** which appears as *C.functionName(D)*. Variable C and variable D that are used in the function call are both considered read.
- **Conditional statement** or more specifically if statement : if A==0 goto label1. A is compared to a value therefore it is read.

A first implementation of the **Write** and **Read** analyses was realized using the dominator analysis to find which statements performing a read or a write, were dominating the exit point of the graph. This implementation was not correct as it had some limitations: in fact, if a value is written in all the paths from a node n1 to a node n2, this value will be written for sure, even if the statements that modify the variable are inside a conditional block.

To clarify the problem let's look at a simple example.

```
public void foo(int i, String s1, String s2) {
  i=i+1;
  if(i<0){
    s1 = "a";
  }
  else{
    s1 = "b";
    s2 = "c";
  }
}
```

If we apply the dominator analysis to see which statements are executed without any condition in the previous code fragment, we will obtain the only statement i=i+1, therefore the only written variable is i. This is not correct, as s1 is written in both the paths that lead to the exit point of the method. That is why we used a personalized data-flow analysis that propagates the information along the branches and performs the intersection of the two sets at merge point. This allows the correct identification of all the read or written variables.

As shown in Figure 6, starting from the first instruction of the graph when we find the first assignment statement, we add a the variable i to the out-set of the node, we propagate the set through the two branches. Inside each of the branches we add the variables when we find the assignment statement that modifies them: s and i are written in both branches. Finally, in the merge node we perform the intersection of the two sets and we end up with both i and s. Using the soot analysis framework illustrated in the previous section, we implemented our analysis by extending the class *ForwardFlowAnalysis*. This kind of analysis starts from the entry point of our control flow graph and flows through the nodes until it reaches the exit point.

Both the read and the write analyses have the same implementation of the copy and merge function. The copy function is the one that propagates the information in the in-set of a graph node to the out-set of it and the implementation was already provided by the soot class *FlowSet*. The source code for this function is the following:

```
protected void copy(Object source, Object dest) {
  FlowSet sourceSet = (FlowSet) source, destSet = (FlowSet) dest;
```

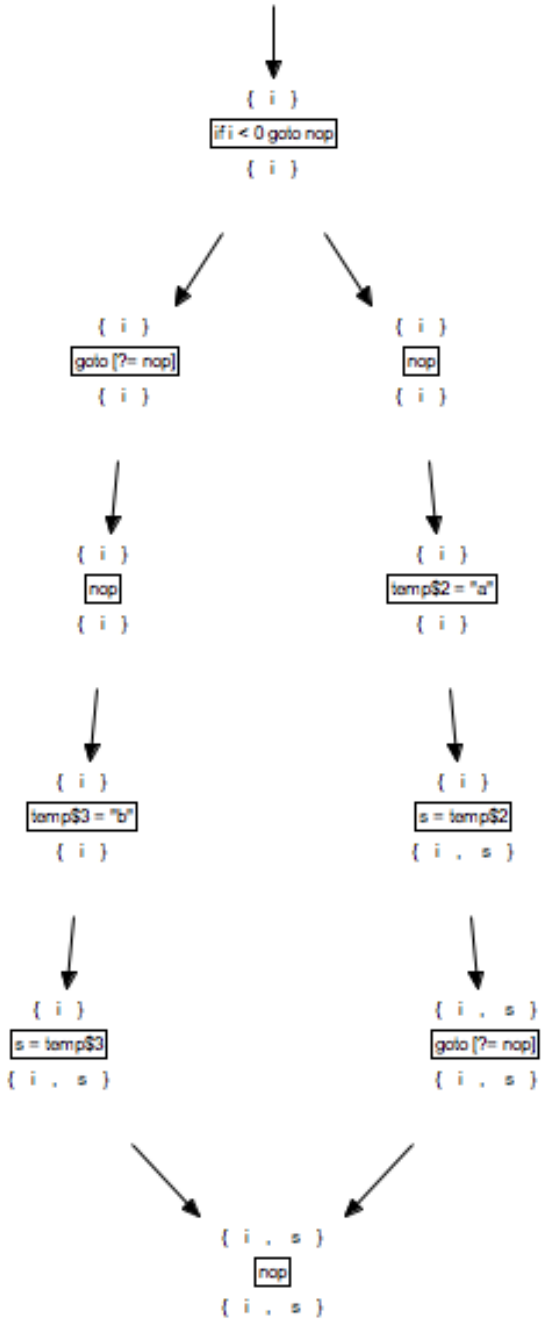

Figure 6. Write analysis information propagation

```
    sourceSet.copy(destSet);
  }
```

The implementation of the function *Flowset.copy()* perform a copy of the information from the Object *source* to *dest*. The merging function, that join the sets of two lattice elements when the flow of two branches converges in one node, is the same in the Read and Write analysis and the merging technique that we chose to use is the intersection, provided also by the *FlowSet* class.

```
  protected void merge(Object in1, Object in2, Object out) {
    FlowSet inSet1 = (FlowSet) in1, inSet2 = (FlowSet) in2, outSet = (FlowSet) out;

    inSet1.intersection(inSet2, outSet);
  }
```

One of the main functions that were realized to analyze correctly the code, is a recursive function that deals with the array variables. Soot produces a three-address code that we use in our analyses, and a problem of this code is that the assignment statement may not contain the real variable name and this can be seen in array variables assignments. For instance, let's look at a simple Java assignment like:

```
  array[i][j] = 2;
```

The corresponding Jimple code produced by Soot is:

```
  temp$0 = array;
  temp$1 = i;
```

```
temp$2 = temp$0[temp$1];

temp$3 = j;

temp$4 = 2;

temp$2[temp$3] = temp$4;
```

As we can notice, if we want to add the variable array to our set we need to start from the last instruction and go backward in the assignments chain until we find it.

That is why we needed to implement the following function, that recursively resolve the array bases temporary variables into the real variable name in the java source.

```
private String findValueWrittenIn(Unit u, ValueBox arrayBase) {
  // find the unit where the array base is defined
  DominatorsFinder df = new MHGDominatorsFinder(graph);
  Unit unitFound = null;
  for (Iterator domsIt = df.getDominators(u).iterator(); domsIt.hasNext();) {
    Unit dom = (Unit) domsIt.next();
    Iterator arrit = dom.getDefBoxes().iterator();
    while (arrit.hasNext()) {
      ValueBox arr = (ValueBox) arrit.next();
      if (arr.getValue().equals(arrayBase.getValue())) {
        unitFound = dom;
      }
    }
  }
```

```
    // Recursion
    if (((Stmt) unitFound).containsArrayRef()) {
      String s = findValueWrittenIn(unitFound, ((Stmt) unitFound)
        .getArrayRef().getBaseBox());
      return s;
    } else {
      String s1 = "";
      for (Iterator paramfoundIt = unitFound.getUseBoxes().iterator(); paramfoundIt.h
        ValueBox paramFound = (ValueBox) paramfoundIt.next();
        s1 += paramFound.getValue().toString();
      }
      return s1;
    }
  }
```

On the other hand, for what concern the read analysis, we had the opposite problem: when a temporary variable temp1 is assigned to a second one temp2, we are not sure that temp2 is read yet. In fact, if temp1 is used as an array base afterwards and it is on the left side of an assignment statement, meaning written, temp2 doesn't have to be added to the read variable set.

To understand it better let's look at the following example:

```
temp$0 = arraybase;
temp$1 = i;
```

```
temp$2 = 0;
temp$0[temp$1] = temp$2;
```

When analyzing the statements in forward order, we would include *arraybase* in the read variables set, but if we look ahead in the code we can see that *temp$0* is actually written because it is used as an array base in the last statement. To solve this problem, in each statement, we need to perform a check to see if the possible read is an actual read and we do this by looking ahead and seeing whether it is a written array base in another statement, if not we can add it to the read variables set.

This operation is computational intensive if performed during the analysis, because for every assignment statement we should look at all the following ones. To limit the complexity and consequently reduce the execution time, the current implementation stores all the array written bases in a data structure before starting the analysis, so that for each statement we only check whether the variable is contained in this variables list.

The function that deals with this pre-analysis is the following:

```
private List<String> getArrayWrittenBases(UnitGraph g) {
  ...
  // For all the statements in the method body
  while (unitIt.hasNext()) {
    ...
    // if it is an assignment statement and contains an array
    if (st.containsArrayRef() && st instanceof AssignStmt) {
```

```
        while (defIt.hasNext()) {
          ValueBox defBox = (ValueBox) defIt.next();
          ...
          while (i1.hasNext()) {
            ValueBox useBox = (ValueBox) i1.next();
            // add the array base to the list
            if (useBox.getValue().equals(
                ((AssignStmt) st).getArrayRef().getBase()))
              arrayBases.add(st.getArrayRef().getBase()
                  .toString());
          }
        }
      }
      // if it is a return statement and an array is returned add the base
      else if (st instanceof ReturnStmt) {
        arrayBases.add(findArrayBase(st,
            ((ReturnStmt) st).getOpBox()));
      }
    }
    return arrayBases;
  }
```

For what concern the **May Write** and **May Read** analyses the generations of the variables set are the same respectively as the the **Write** and **Read** ones and the implementations of the

merge and copy functions is not relevant as we are only interested in finding read and writes that were not detected by the previous analyses.

4.5 Path conditions analysis

As we stated in the previous chapters, our final goal is to identify conflicts between rules and generate automatically test cases to verify them. To achieve this goal we, not only need to know what a rule might read or write, but also the conditions that leads to this action. These conditions are obviously empty for the variables identified by the previous **Write** and **Read** analyses, as they will be executed for sure within the code, but it is relevant for the other two sets detected by the **May Write** and **May Read** analyses.

Hence, the path condition analysis deals with finding the path conditions that leads to a possible write or read of a property by a certain rule.

This kind of analysis is implemented by extending *soot.toolkits.scalar.ForwardBranchedFlowAnalysis* that, as we described before, allows the propagation of different information to the different branches of a graph. This feature is very relevant, because, when we find a conditional statement *if A>2 then B else C*, we want to set the path condition to *A>2* in the *then* branch and *A<=2* in the *else* branch.

The implementation of these operations can be seen in the following code fragment that shows the generation and propagations of the condition through the different branches.

```
if (u instanceof IfStmt) {
    ConditionExpr condition = (ConditionExpr) ((IfStmt) u)
        .getCondition();
```

```
        ValueBox op1 = condition.getOp1Box();
        ValueBox op2 = condition.getOp2Box();
        String operator1 = new String();
        String operator2 = new String();

        operator1 = this.getSymbolicValue(u, op1);
        operator2 = this.getSymbolicValue(u, op2);

        outbranch.add(operator1 + condition.getSymbol().trim() + operator2);

        outfall.add(operator1
            + this.getNegation(condition.getSymbol().trim())
            + operator2);
    }
```

Whenever we find an If statement in the Jimple code, we take its condition expression, that is composed of two operands and a symbol, we perform the *symbolic execution* of the operands and we propagate the condition as it is to the one branch, and the complementary condition, calculated by negation of the symbol, to the other one.

The two main function that are used in this code are *getNegation(String symbol)*, and *getSymbolicValue(u, op2)*. The first is very simple: its role is to return the negation of a relational operator that is passed as a parameter.

```
  private String getNegation(String symbol) {
```

```
    if (symbol.compareTo("<") == 0)
      return ">=";
    else if (symbol.compareTo(">") == 0)
      return "<=";
    else if (symbol.compareTo("<=") == 0)
      return ">";
    else if (symbol.compareTo(">=") == 0)
      return "<";
    else if (symbol.compareTo("==") == 0)
      return "!=";
    else if (symbol.compareTo("!=") == 0)
      return "==";
    return null;
  }
```

The second one requires a more accurate description as it simulates a symbolic execution of the Jimple method. Symbolic execution (also symbolic evaluation) refers to the analysis of programs by tracking symbolic rather than actual values(27) that are symbolic formulas over the input symbols(28).

Instead of implementing a symbolic execution function that starts from the parameters of the method and propagates their symbolic value in a forward fashion, we decided to implement a recursive function that receives as an input a variable and the statement in which it is used,

and go backward to the previous instruction to find its symbolic value.

The pseudo-code related to this function is the following:

```
private String getSymbolicValue(Unit u, ValueBox localVar) {
  String s1 = "";
  ...
    //find the previous instruction where localVar is assigned
    ...
    if (previousInstruction instanceof AssignStmt) {
      if (rightOperand instanceof CallFunctionInvocation) {
        s1 += getSymbolicValue(rightOperand.getFirstVariable());
        s1 += "."+ rightOperand.getFunctionName()+ "(";
        // for each parameter get the its symbolic value
        for (parameter in rightOperand.getParameters()) {
          s1 += getSymbolicValue(previousInstruction,parameter);
        }
        s1 += ")";
      } else if (rightOperand instanceof ArrayReference) {
        s1 += getSymbolicValue(unitFound,rightOperand.getArrayBase());
        s1 += "[";
        s1 += getSymbolicValue(unitFound,rightOperand.getArrayIndex());
        s1 += "]";
      } else if (rightOperand instanceof BinopExpr) {
        s1 += getSymbolicValue(rightOperand.getFirstOperand());
```

```
            s1 += rightOperand.getSymbol();
            s1 += getSymbolicValue(unitFound,rightOperand.getSecondOperand());
          } else if (rightOperand instanceof Variable) {
            s1 += getSymbolicValue(unitFound, rightOperand);
          } else {
            s1 += rightOperand.toString();
          }
        }
      ...
    }
```

The symbolic value is calculated by recursively finding the previous assignment of a local variable and substituting the symbolic value assigned to it.

As for the read and write analyses, we had to provide the implementation of the copy and merge functions. The first is not changed from the previous one, while the second one has been changed:

```
  protected void copy(Object source, Object dest) {
    FlowSet sourceSet = (FlowSet) source, destSet = (FlowSet) dest;
    sourceSet.copy(destSet);
  }

  protected void merge(Object in1, Object in2, Object out) {
      FlowSet inSet1 = (FlowSet) in1, inSet2 = (FlowSet) in2,
```

```
    outSet = (FlowSet) out;
    if (inSet1.isEmpty())
      inSet2.copy(outSet);
    else if (inSet2.isEmpty())
      inSet1.copy(outSet);
    else {
      if (PATHCONSTANT == 1)
        inSet1.copy(outSet);
      else
        inSet2.copy(outSet);
    }
  }
```

As we can see, the merge function propagates the value of one of the two in-sets that it receives as inputs. More precisely, if one of the two insets are empty, the other one is propagated, if they both contain some path conditions the propagation depends on a constant PATHCONSTANT, that is passed as a parameter to the analysis. This constant has been introduced to extract two different path conditions for each statement and this is done by running the analysis twice with a different value of PATHCONSTANT.

Figure 7 illustrates the flow of the information and how the different conditions are propagated to different branches: the if statement node generates two conditions that are complementary. One is copied to the *If* branch while the other one is copied to the *Else* branch. When

the two branches converge the conditions from the *If* branch are propagated to the following node.

4.5.1 Rule Data Structure

The results from the previous analyses are stored in a data structure represented by the class *Rule*.

```
public class Rule {


  private String ruleMethodName;
  private String kind;
  private Property parent;
  private List<Property> writtenVariables;
  private List<Property> readVariables;
  private List<Property> mayWrittenVariables;
  private List<Property> mayReadVariables;
  }
```

The main component of this data structure are:

- RuleMethodName is a unique identifier for a rule and is also the name of Java function in which the rule was parsed.
- parent is the property to which the rule is attached to, that could therefore influence its firing.
- kind is a string that represents the rule kind, which decides how the rule will be triggered.

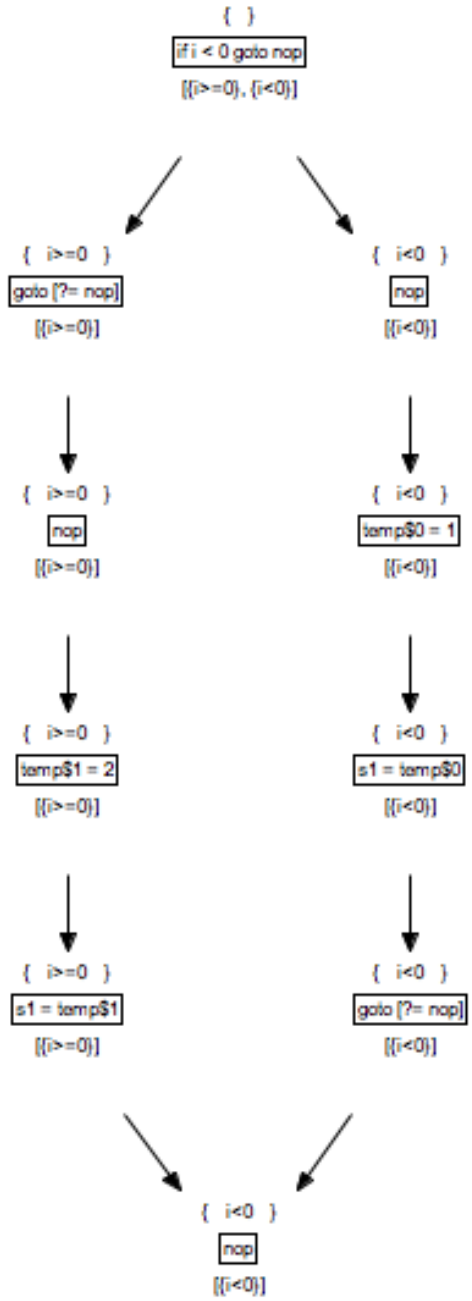

Figure 7. Path condition analysis

- Four lists of properties that are the written,read, mayWritten and mayRead sets.

The *Property* class reported below, represents a variable in the system: variableName is the name within the Java code, while fullPathName that is the real unique name to identify it within the rule-based system. The path conditions analysis fills the last two fields in the case of possibly written or possibly read variables.

```
public class Property {
  private String variableName;
  private String fullPathName;
  private List<String> firstPathConditions;
  private List<String> secondPathConditions;
}
```

As the previous ones, this data structure is exported as an XML file to be reused in the following step. An example of XML representation of a rule is provided in the next section.

4.6 Conflict Identification

The conflict identification phase uses the results from the previous analysis to detect possible conflicts between rules. A conflict arises when two rules R1,R2 can be triggered concurrently and one of the following situation is verified:

- A property p is read by R1 and written by R2.
- A property p is read by R2 and written by R1.
- A property p is written by both R1 and R2.

In the case of possible read or write we need to keep track of the path conditions that lead to it. What we want to present as an output is a list of the possible conflicts with the conditions, if present, that must be verified to let the conflict happen.

The process of identification of conflicts can be divided into two main steps:

1. Identification of rules that can be triggered concurrently.
2. Conflict detection within the concurrent rules.

The first step is based on the following properties: all the rules that have *ReadTriggeredKind* are fired when the property that is attached to them (the parent rule) is read, all the rules that have *WriteTriggeredKind* are fired when the parent property is written. *ReadTriggeredKind* and *WriteTriggeredKind* are illustrated in Table III.

TABLE III

READ TRIGGERED KINDS.

Read triggered	Write triggered
IS_VALID_KIND	POST_SET_KIND
SUGGESTED_VALUE_KIND	POST_ADD_KIND
IS_REQUIRED_KIND	LOWER_LIMIT_KIND
LOWER_LIMIT_KIND	UPPER_LIMIT_KIND
UPPER_LIMIT_KIND	ALLOWED_VALUE_KIND
ALLOWED_VALUE_KIND	

These properties allow us to easily identify all the groups of rules that can be fired by reading or writing on a certain property and the groups are represented by the class *PossibleConflict*. The functions *getRConflicts()* and *getWConflicts()* of the class *XMLQueryUtility* return respectively the read and write triggered *PossibleConflict*s.

```
public class PossibleConflict {
  private String parentProperty;
  private List<Rule> rules;
}

public List<PossibleConflict> getRConflicts() {

  List<PossibleConflict> possibleConflicts = new ArrayList<PossibleConflict>();
  List<String> parents = this.parents;
  List<Rule> rules = this.rules;
  List<String> kinds = getReadTriggeredKinds();

  for (String parent : parents) {

    PossibleConflict conflict = new PossibleConflict(parent);

    for (Rule r : rules) {
      // if it is a read Triggered Rule
      if (kinds.contains(r.getKind())) {
        // if it is attached to parent
```

```
            if (parent.compareTo(r.getParent().getFullPathName()) == 0) {
              // initialize the rule properties
              conflict.addRule(r);
            }
          }
        }
        if (conflict.containsAtLeastTwoRules()) {
          possibleConflicts.add(conflict);
        }

      }
```

The second step is performed by the method *IdentifyConflicts()* of the class *ConflictIdentificator* and consists of taking every *PossibleConflict*, and find the pairs of rules R1,R2 in which one of the three previously described situations is verified. Once we find such a pair we find what we defined a conflict.

The structure that was chosen to represent a conflict is the following:

```
public class ConflictPair {
  private Rule rule1;
  private Rule rule2;
  private Property property1;
  private Property property2;
  private String actionRule1;
```

```
  private String actionRule2;
}
```

As in the previous phases, XML was chosen to represent the data structure for reusability reasons. This is an example of the XML output of our conflict identification:

```
<conflict>
  <property>NAMED_INSURED.SEQUENCE_NUMBER</property>
  <rule1>
    <ruleMethodName>PPFM3049D00DLTE</ruleMethodName>
    <action>mayWrite</action>
    <pathCondition>
      [this.isNull(NAME_TYPE)==0, NAME_TYPE.compareTo("I")!=0]
    </pathCondition>
  </rule1>
  <rule2>
    <ruleMethodName>PPFM3049D00DLTE</ruleMethodName>
    <action>mayRead</action>
    <pathCondition>
      [this.isNull(NAME_TYPE)==0, NAME_TYPE.compareTo("I")!=0]
    </pathCondition>
  </rule2>
</conflict>
```

The conflict arises on the property *NAMED_INSURED.SEQUENCE_NUMBER*, that might be written by the rule *PPFM3049D00DLTE* if the conditions *this.isNull(NAME_TYPE)==0, NAME_TYPE.compareTo("I")!=0* are verified and might be read by rule *PPFM3049D00DLTE* if the conditions *this.isNull(NAME_TYPE)==0, NAME_TYPE.compareTo("I")!=0* are verified.

4.7 Reachability Graph Construction

The pairs that we identified in the previous section are, in reality, potential conflicts. We found a situation in which a conflict can arise but we still need to discover whether this situation can really happen in a real execution of the system, therefore we needed to perform a reachability analysis.

To implement this analysis we decided to build a triggering graph of all the rules in the system.

This graph has two kinds of nodes:

- Rule nodes that represent rules in the application.
- Property nodes that represents the various properties in the system. For each rule we create two nodes: a ReadPropertyNode and a WritePropertyNode that indicate respectively a read and a write on the property.

The implementations of these two nodes the following:

```
public class RuleNode {
```

```
    private Rule rule;
  }

  public class PropertyNode {
    private String propertyName;
  }
```

The rule node doesn't need any explanation while the property node can be instance of two classes depending on the action The graph has also different kinds of edges:

- From a property to rule node: this edge exists if the rule is triggered when the property is read or written. In the first case the property node will be a ReadPropertyNode, in the second a WritePropertyNode.
- From a rule to a property node: in the case or readPropertyNode this means that the rule reads the property while in case of writePropertyNode this means a write on the property. This edge is also labelled with the path condition that leads to the action on the property.

The first kind of edges is implemented using a *java.util.Map* structure that associate a property with a set of rules (*Map<PropertyNode, Set<RuleNode>>propertyToRulesEdges*).
The second one is realized by the class *ConditionalEdge*:

```
  public class ConditionalEdge {
    private RuleNode source;
    private PropertyNode dest;
```

```
    private List<String> conditions;
  }
```

Finally the graph implementation is contained in the *TriggeringGraph* class and it contains all the functions to add nodes and edges, retrieve them from the graph, perform checking on the existence of an edge or a node to make sure not to generate duplicate nodes or edges. The source code for this class is the following:

```
  public class TriggeringGraph {
    protected Set<RuleNode> rules;
    protected Set<PropertyNode> properties;

    protected Set<ConditionalEdge> ruleTopropertiesEdges;
    protected Map<PropertyNode, Set<RuleNode>> propertyToRulesEdges;
  }
```

To verify the correctness of the graph and give a visual representation of the conflict we found we decided to actually draw the graph. The visualization makes use of GraphML, a comprehensive and easy-to-use file format for graphs based on XML(29).

The function of the *TriggeringGraph* class deals with coding the graph into a GraphML stream that after being saved as a file, can be easily visualized by many tools.

We used this conventions to represent the graph:

1. Rules are represented by round nodes and labelled with a unique identifier.

2. Properties are represented with square nodes and the label is the full path name of the property preceded by "Read" or "Write" depending on the action associated with it.

Figure 8 illustrates a portion of the graph that we generated and more specifically a conflicts that is identified by the red edges.

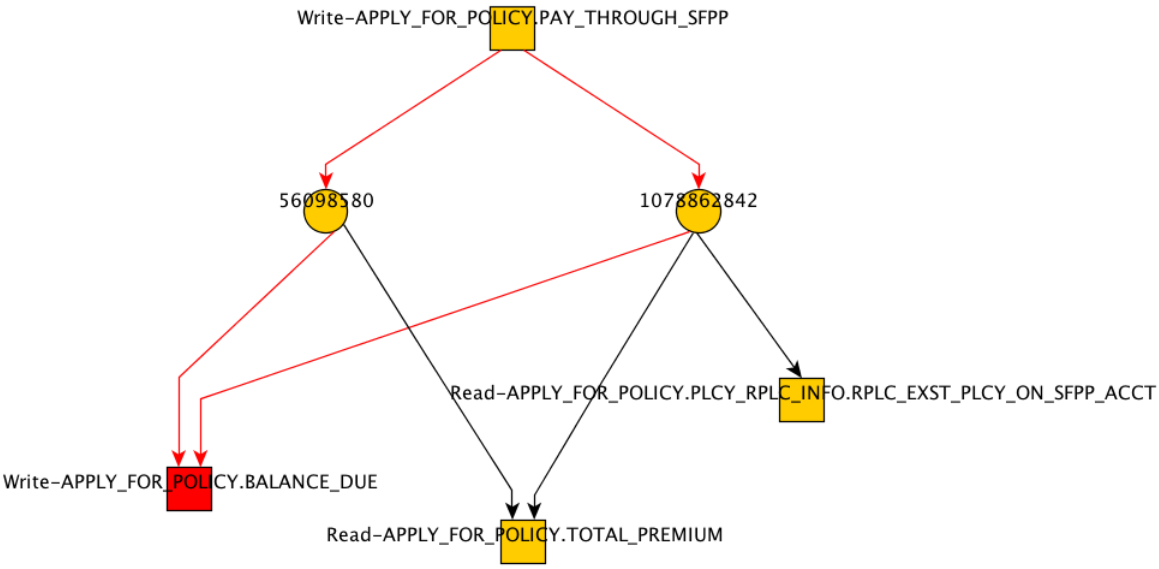

Figure 8. An example of conflict in the graph

The two rules are triggered by a write on APPLY_FOR_POLICY.PAY_THROUGH_SFPP, therefore they can be fired concurrently. In addition, they are both connected to the same

property APPLY_FOR_POLICY.BALANCE_DUE and they perform a write on it. Hence, a different triggering sequence results in different values of the property.

CHAPTER 5

RESULTS

We evaluated TERLATO on a large-scale rule-driven application that is currently in use by a major insurance company and we identified several potential conflicts. We were not provided the entire application rule base, but we were able to apply our solution to a subset of it, that consisted of 1292 rules.

5.1 System configuration and execution time

Our application has proven to be fairly lightweight during our experiments, as we were able to run it in a limited amount of time for more than a thousands rules on a personal computer.

To evaluate the system requirements and the complexity of our tool we will describe the system configuration and the execution time required to execute the conflict generation process on the rule set that we were provided. The system configuration of the machine we used to perform the conflict generation is illustrated in Figure 9.

The tool execution consists of three main phases that were executed separately:

- Parsing the rules into java code.
- Data and control flow analysis on each rule.
- Conflict list generation starting from the results of the previous steps.
- Rule-Graph generation.

The execution times of each of these phase is shown in Table IV.

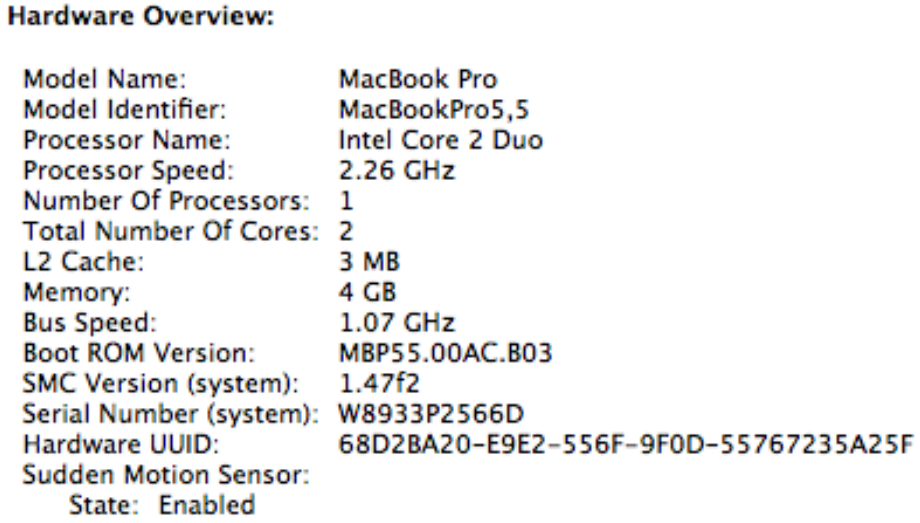

Figure 9. System Configuration.

As we can see, the parsing phase takes 8 seconds to execute while the analysis phase is the most complex one (27 sec). The long execution time is due to multiple data and control flow analysis performed to each rule. In fact, this process consists of six analyses performed with *Soot* framework: four that results in the four sets of *Written, May written, Read* and *May Read* variables and two that output the path conditions associated with each Read or Write. The conflict list generation is the process that identifies the potential conflicts between rules and it is not very time consuming (8 sec). The graph generation, on the other hand seems to be the phase that requires the longest time to execute. In reality, this execution time is due to the fact that we are producing a graphML file as an output so that we are able to draw the graph to get a visual representation of it. In fact, the java function toGraphMLString() that

TABLE IV

TERLATO'S EXECUTION TIME.

Phase	Execution Time
Parser	8 s
Rule Analysis	27 s
Conflict Generation	7.5 s
Graph Generation	230 s

generates this file is very time consuming: by isolating this function and measuring its execution time, we found out that it takes 230 seconds. This reduces the actual graph generation to less than a second. As this function is not needed for the tool to work, but only to give a visual representation of the graph, we do not include this time when we consider the overall execution time of TERLATO.

We also measured the execution time growth with the respect to the number of rules. We run TERLATO on 25%, 50% of the rule set we were provided and compared the time needed to execute to the initial test that was performed on 1293 rules it and the results of this test are shown in Table V.

These results show that the execution time grows more than linearly with the respect to the number of rules. We can also say, that the growth is less than quadratic with the respect to the number of rules. These are obtained with simple empirical tests, therefore they do not provide a precise result but they give an idea of the complexity of TERLATO.

TABLE V

TERLATO'S EXECUTION TIME GROWTH.

Phase	25% Rule Set	50% Rule Set	100% Rule Set
Parser	0.2s	2.3s	7.9s
Rule Analysis	7.1s	10.7s	26.8s
Conflict Generation	1.0 s	3.5 s	7.6s
Total Time	**8.3s**	**16.4s**	**42.3s**

5.2 Conflicts found

By analyzing this set of 1292 rules with TERLATO, we identified 48 conflicts. Figure 10 shows more precisely the results of the analyses: the majority of the conflicts are write-write conflicts, both the rules try to write the same variable.

We also wanted to point out the conflicts that are associated with a path condition and the ones that are not.

The subset of the system that we utilized constitutes about 10% of the complete rule-driven application, therefore by applying the analysis to the entire rule base the number of potential conflicts is estimated to increase with a more than linear growth, since an increase in the number of nodes in the graph might results in a quadratic increase of the number of edges and therefore in the number of conflicts.

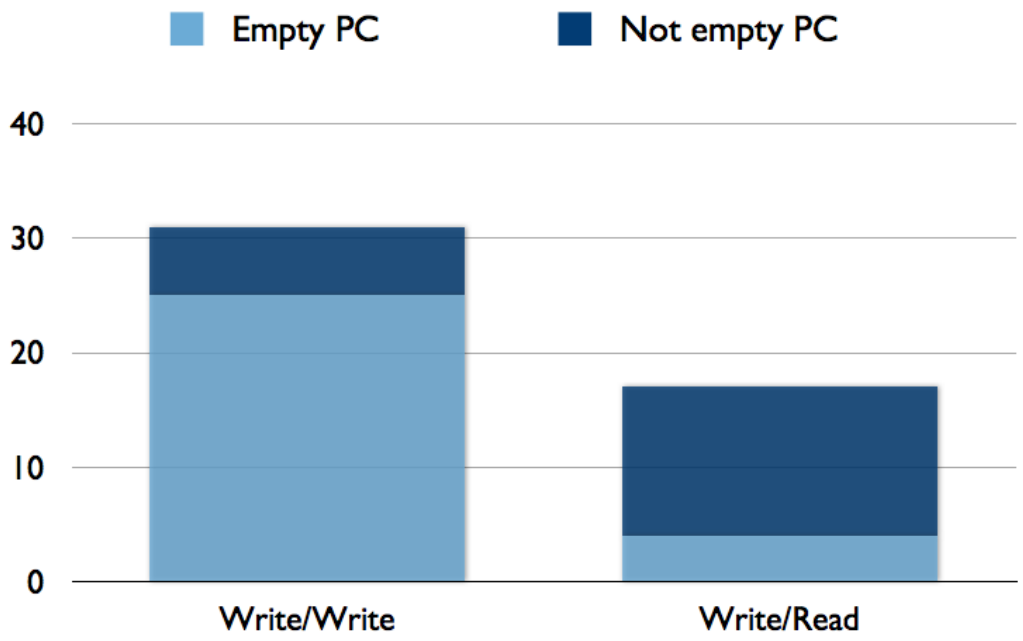

Figure 10. Conflicts types.

CHAPTER 6

CONCLUSION AND FUTURE WORK

The goal of this thesis is to propose a new approach to test rule-driven applications. Through the identification of concurrent rules and the application of data and control flow analysis, combined with symbolic execution of the rule code, we were able to detect several conflicts within a major insurance company application.

Our solution does not claim to be either sound nor complete. Future improvements could therefore go in this direction. Removing conflicts that cannot be reached because of unsatisfiable path conditions, will increase the soundness of our approach, while discovering new kinds of conflicts would contribute to make it more complete. The main work that needs to be done is to start from the results we came up with, and perform an automatic generation of test cases that will allow us to verify the potential conflict on the real application.

Testing rule-driven application is still an open research topic where a lot of work still need to be done. However TERLATO proved to be an effective tool for this kind of applications and created a new way to approach this problem.

APPENDICES

Appendix A

PARSER CODE

This is the source code that constitutes the parser for the proprietary language of the rule based system that we analyzed. As the parser source code is too long to be reported here we will show the grammars that deal with the rule header and body parts. We omitted the action of each production rule for readability reasons. The following grammar manage the parsing of the rules:

```
1  grammar RuleParser;
2
3
4  options {
5    language = Java;
6  /*   output = AST;*/
7  }
8
9  tokens{
10   FLOAT_LITERAL;
11 }
12
13
14 @header{
15
```

Appendix A (Continued)

```
16 package com.accenture.techlabs.productmodel;
17
18 import java.util.HashSet;
19 import com.accenture.techlabs.productmodel.SymbolTable;
20 import java.io.BufferedWriter;
21 import java.io.FileWriter;
22
23 }
24
25 @lexer::header{
26 package com.accenture.techlabs.productmodel;
27 }
28
29 @members {
30
31
32 static public class TypedDeclaration{
33     String declareType;
34     String variableName;
35     String javaString;
36     String initialization;
37 }
38
39 private int indent = 0;
40
41 private String getIndent(){
```

Appendix A (Continued)

```
42      String spaces = "";
43      for (int i=0; i<indent; i++){
44          spaces += " ";
45      }
46      return spaces;
47  }
48
49  //remove a specific char form a string
50  public String removeChar(String s, char c) {
51
52          String r = "";
53
54          for (int i = 0; i < s.length(); i ++) {
55              if (s.charAt(i) != c) r += s.charAt(i);
56          }
57
58          return r;
59        }
60
61  //resolves the problems of multidimensional array variables, that in the
        declaration are declared as
62  //unidimensional and then are used with more than one index, by modifying their
        declaration
63  public String fixArrayDimensions(String declaration) {
64    String result = "";
65    ArrayList<SymbolEntry> arrays = $rule::symbolTable.getMultiDimArrays();
```

Appendix A (Continued)

```
66    for(SymbolEntry entry : arrays){
67      int dim = entry.getDimension();
68      String arrayName = entry.getVariableName();
69      for(int i=0; i<dim;i++)
70        arrayName+="[]";
71
72      declaration = declaration.replace(" "+entry.getVariableName()+"[]"," "+
            arrayName);
73    }
74    return declaration;
75  }
76
77  //print a string to a specified file
78  public void printToFile(String xmlString,String fileName){
79     try{
80        FileWriter fstream = new FileWriter(fileName);
81        BufferedWriter out = new BufferedWriter(fstream);
82        out.write(xmlString);
83        out.close();
84     }
85     catch (Exception e){
86        System.err.println("Error: " + e.getMessage());
87      }
88  }
89
90  public enum Type {
```

Appendix A (Continued)

```
91      STRING,BOOLEAN,INTEGER,FLOAT,DATE,PERCENTAGE,ERROR,NULL;
92  }
93
94  private boolean insideCondition = false;
95
96  private String returnVariable = null;
97
98  private HashSet<String> variables = null;
99
100 }
101
102 language returns[String javaString]
103   scope{
104     SymbolTable functionTable;
105     }
106   @init{
107     $language::functionTable = new SymbolTable();
108     $language::functionTable.initializeFunctionTable();
109   }
110   :
111   {int count = 0;
112   String xmlString = "<ruleList>\n";
113   $javaString = "package com.accenture.techlabs.productmodel.parsedrules;\n\n";
114   $javaString += "import java.util.Date;\n\n";
115   $javaString += "public class Rules{\n";
116   }
```

Appendix A (Continued)

```
117   (rule{System.out.println(count++);
118         $javaString += $rule.javaString;
119         xmlString += $rule.xmlString;})+
120   EOF
121   {$javaString += "}";
122   xmlString +="</ruleList>";
123
124   printToFile(xmlString,"Rulelist.xml");
125   }
126   ;
127
128 rule returns[String javaString, String xmlString]
129   scope {String methodName; SymbolTable symbolTable;}
130   @init{
131     $rule::methodName = "";
132     $rule::symbolTable = new SymbolTable();
133   }
134   :
135   {$javaString = ""; $xmlString="";}
136   (ruleHeader) {$xmlString+=$ruleHeader.xmlString;}
137   (ruleDefinition{$javaString += $ruleDefinition.javaString;
138                   $xmlString += "<source>\n"+$ruleDefinition.javaString.length()+
                        "\n</source>\n";})?
139   {
140   //add the methodName that is the unique identifier for a rule
141   $xmlString += "<methodName>\n"+$rule::methodName+"\n</methodName>\n";
```

Appendix A (Continued)

```
142   $xmlString += $rule::symbolTable.toXmlString();
143   $xmlString += "</rule>\n";
144   }
145   ;
146
147 ruleDefinition returns [String javaString]
148   scope{
149     boolean multipleDimentionArrayFound;
150   }
151   @init{
152     $ruleDefinition::multipleDimentionArrayFound = false;
153   }
154   :
155   { boolean  hasReturns = false, hasReceives = false, hasLocal = false;
156     variables = new HashSet<String>();
157     String methodDeclaration; String variablesDeclaration; String statements="";
158   }
159   (receivesDeclaration { hasReceives = true; } )?
160   (returnsDelcaration  { hasReturns = true; } )?
161   (usesDeclaration  { hasLocal = true; } )?
162   { indent += 4; }
163   statements
164   //EOF
165   {
166     indent -= 4;
167     if (hasReturns)
```

Appendix A (Continued)

```
168         methodDeclaration = $returnsDelcaration.declaration.declareType;
169     else methodDeclaration = "void";
170     methodDeclaration += " " + $rule::methodName;
171
172     variablesDeclaration="(";
173
174     if (hasReceives){
175         variablesDeclaration += $receivesDeclaration.javaString;
176         if(hasReturns)
177             variablesDeclaration += ",";
178
179     }
180     if (hasReturns) {
181         variablesDeclaration += $returnsDelcaration.declaration.javaString;
182     }
183     variablesDeclaration += "){\n";
184
185     if (hasLocal) {
186         variablesDeclaration += $usesDeclaration.javaString;
187     }
188     //if the returnVariable is an array and has a dimension add its
          initialization
189     if(hasReturns)
190       if($returnsDelcaration.declaration.initialization!=null)
191         statements += $returnsDelcaration.declaration.initialization;
192
```

Appendix A (Continued)

```
193     statements += $statements.javaString;
194     //If the function returns something I add the return Statement because they
            don't explicitly write it
195     if (hasReturns) {
196         statements += "return "+$returnsDelcaration.declaration.variableName+";\n
                ";
197     }
198     statements += "}\n";
199     returnVariable = null;
200
201     //If multidimensional Array variables are used modify the array declaration
202     if($rule::symbolTable.getMultiDimArrays().size()>0){
203       variablesDeclaration = fixArrayDimensions(variablesDeclaration);
204     }
205
206     $javaString = methodDeclaration + variablesDeclaration + statements;
207     //System.out.println($javaString);
208   }
209 ;
210
211 ruleHeader returns[String xmlString]:
212   {$xmlString="<rule>\n";}
213   (ruleSpecIdDeclaration) {$xmlString += $ruleSpecIdDeclaration.xmlString;}
214   (ruleLongNameDeclaration) {$xmlString += $ruleLongNameDeclaration.xmlString;}
215   (systemNameDeclaration) {$xmlString += $systemNameDeclaration.xmlString;}
216   (kindDeclaration) {$xmlString += $kindDeclaration.xmlString;}
```

Appendix A (Continued)

```
217   (applicationDeclaration {$xmlString += $applicationDeclaration.xmlString;})?
218   (parentsDeclaration {$xmlString += $parentsDeclaration.xmlString;})
219   (interfaceDeclaration {$xmlString += $interfaceDeclaration.xmlString;})?
220     {$xmlString += "<properties>\n";}
221   (argPathDeclarations {$xmlString += $argPathDeclarations.xmlString;})*
222     {$xmlString += "</properties>\n";}
223   (sourceDeclaration)?
224  ;
225
226 receivesDeclaration returns [String javaString]:
227   RECEIVES parameterDeclarations
228   { $javaString = $parameterDeclarations.javaString; }
229   ;
230
231 parameterDeclarations returns [String javaString]:
232   first=parameterDeclaration
233   {$javaString = $first.javaString;}
234   (follow=parameterDeclaration { $javaString += ", "+$follow.javaString; } )*
235   ;
236
237
238 parameterDeclaration returns [String javaString]:
239   {boolean array = false;}
240   IDENTIFIER (LPAREN RPAREN {array = true;} )? dataType (REQUIRED)?
241   {
242     //TODO!! Deal with REQUIRED?!
```

Appendix A (Continued)

```
243       $javaString = $dataType.javaString + " "+$IDENTIFIER.text+(array?"[]":"") ;
244       variables.add($IDENTIFIER.text);
245
246       // add parameter to the symboltable
247       $rule::symbolTable.addEntry($IDENTIFIER.text,$dataType.type,true);
248       if(array)
249         $rule::symbolTable.increaseDimension($IDENTIFIER.text);
250     }
251     ;
252
253
254
255 returnsDelcaration returns [TypedDeclaration declaration]:
256   RETURNS returnVariableDeclaration
257     {
258       $declaration = $returnVariableDeclaration.declaration;
259       returnVariable = $returnVariableDeclaration.declaration.variableName;
260       variables.add(returnVariable);
261     }
262     ;
263
264
265 usesDeclaration returns [String javaString]:
266   USES variableDeclarations
267   {$javaString = $variableDeclarations.javaString;}
268     ;
```

Appendix A (Continued)

```
269
270 variableDeclarations returns [String javaString]:
271   first=variableDeclaration
272   { $javaString = $first.declaration.javaString;}
273   (COMMA follow=variableDeclaration {$javaString +=$follow.declaration.javaString
          ;} )*
274   ;
275
276
277 variableDeclaration returns [TypedDeclaration declaration]
278   :
279   { boolean array = false;  String dimension= null; declaration = new
          TypedDeclaration();}
280   IDENTIFIER (LPAREN (INTEGER_LITERAL {dimension=$INTEGER_LITERAL.text;})?
281               RPAREN {array = true;})? dataType
282   {
283     $declaration.declareType = $dataType.javaString;
284     $declaration.variableName = $IDENTIFIER.text;
285     $declaration.javaString = getIndent() + $declaration.declareType+" "+
286         $IDENTIFIER.text+(array?"[]":"");
287
288     // add parameter to the symboltable
289     $rule::symbolTable.addEntry($IDENTIFIER.text,$dataType.type);
290
291     //initializations of local variables
292     if (array){
```

Appendix A (Continued)

```
293         //set dimension = 1 in the symbolTable
294         $rule::symbolTable.increaseDimension($IDENTIFIER.text);
295         if (dimension!=null){
296           $declaration.javaString += " = new "+$dataType.javaString+"["+dimension+"
                ]";
297         }
298       }
299       else if($dataType.type==Type.INTEGER||$dataType.type==Type.FLOAT)
300         $declaration.javaString += " = 0";
301       else if($dataType.type==Type.BOOLEAN)
302         $declaration.javaString += " = false";
303       else
304         $declaration.javaString += " = null";
305
306       $declaration.javaString += ";\n";
307       variables.add($IDENTIFIER.text);
308
309
310     }
311     ;
312
313   returnVariableDeclaration returns [TypedDeclaration declaration]
314     :
315     { boolean array = false;  String dimension= null; declaration = new
          TypedDeclaration();}
316     IDENTIFIER (LPAREN (INTEGER_LITERAL {dimension=$INTEGER_LITERAL.text;})?
```

Appendix A (Continued)

```
317                 RPAREN {array = true;})? dataType
318   {
319     $declaration.declareType = $dataType.javaString+(array?"[]":"");
320     $declaration.variableName = $IDENTIFIER.text;
321     $declaration.javaString = getIndent() + $declaration.declareType+" "+
            $IDENTIFIER.text;
322
323      // add parameter to the symboltable
324     $rule::symbolTable.addEntry($IDENTIFIER.text,$dataType.type,true);
325
326     if (array){
327       $rule::symbolTable.increaseDimension($IDENTIFIER.text);
328       if (dimension!=null){
329         $declaration.initialization = $declaration.variableName
330         +" = new "+$dataType.javaString+"["+dimension+"];\n";
331       }
332     }
333     variables.add($IDENTIFIER.text);
334   }
335   ;
336
337
338
339 dataType returns [String javaString, Type type]:
340     BOOLEAN  {$javaString = "boolean"; $type = Type.BOOLEAN;}
341   | CURRENCY  {$javaString = "float"; $type = Type.FLOAT;}
```

Appendix A (Continued)

```
342    | DATE  {$javaString = "Date"; $type = Type.DATE;}
343    | FLOAT {$javaString = "float"; $type = Type.FLOAT;}
344    | INTEGER {$javaString = "int"; $type = Type.INTEGER;}
345    | STRING  {$javaString = "String"; $type = Type.STRING;}
346    | PERCENTAGE {$javaString = "Float"; $type = Type.PERCENTAGE;}
347    ;
348
349  statements returns [String javaString]:
350    {$javaString = "";}
351    ( statement  {$javaString += $statement.javaString;} )+
352    ;
353
354  statement returns [String javaString]:
355      assignmentStatement  {$javaString = $assignmentStatement.javaString;}
356    | callStatement  {$javaString = $callStatement.javaString; }
357    | doStatement {$javaString = $doStatement.javaString; }
358    | forStatement  {$javaString = $forStatement.javaString; }
359    | ifStatement {$javaString = $ifStatement.javaString;}
360    | redimStatement  {$javaString = $redimStatement.javaString;}
361    | returnStatement  {$javaString = $returnStatement.javaString;}
362    | selectCaseStatement {$javaString = $selectCaseStatement.javaString;}
363    ;
364
365  assignmentStatement returns [String javaString]:
366    variableReference ASSIGN expression{
367    boolean intCast = false;
```

Appendix A (Continued)

```
368  boolean floatCast = false;
369
370  //manage assignments between Integer and Float type variables and vice-versa
371  if($variableReference.type==Type.INTEGER && $expression.type == Type.FLOAT)
372    intCast = true;
373  else if($variableReference.type==Type.FLOAT && $expression.type == Type.INTEGER
         )
374    floatCast = true;
375
376   $javaString = getIndent() + $variableReference.javaString +"=";
377   if(intCast)
378      $javaString += "(int)";
379   else if(floatCast)
380      $javaString += "(float)";
381   $javaString += $expression.javaString+";\n";
382   }
383  |
384  arrayReference ASSIGN expression
385  {
386  boolean intCast = false;
387  boolean floatCast = false;
388
389  //manage assignments between Integer and Float type variables and vice-versa
390  if($arrayReference.type==Type.INTEGER && $expression.type == Type.FLOAT)
391    intCast = true;
392  else if($arrayReference.type==Type.FLOAT && $expression.type == Type.INTEGER)
```

Appendix A (Continued)

```
393     floatCast = true;
394
395   if($arrayReference.isArrayAssignment==false){
396     $javaString = getIndent() + $arrayReference.javaString +"=";
397     if(intCast)
398       $javaString += "(int)";
399     else if(floatCast)
400       $javaString += "(float)";
401     $javaString += $expression.javaString+";\n";
402     }
403   else{
404     //eliminate the [] from the array name
405     String arrayName=$arrayReference.javaString.substring(0, $arrayReference.
          javaString.length()-2);
406
407     $javaString = getIndent() +"for(int i=0;i<"+ arrayName+".length;i++){\n";
408     indent+=4;
409     $javaString+= getIndent()+arrayName+"[i]=";
410
411     if(intCast)
412       $javaString += "(int)";
413     else if(floatCast)
414       $javaString += "(float)";
415
416     $javaString += $expression.javaString+";\n";
417     indent-=4;
```

Appendix A (Continued)

```
418       $javaString+= getIndent()+"}\n";
419     }
420     }
421     ;
422
423
424 callStatement returns [String javaString]:
425     CALL IDENTIFIER LPAREN
426     {
427       $javaString = getIndent()+ $IDENTIFIER.text+"(";
428     }
429     (first=expression  {$javaString+=$first.javaString;}
430      (COMMA follow=expression {$javaString+=", "+$follow.javaString;})*
431     )? RPAREN
432     {$javaString += ");\n";}
433     ;
434
435
436 doStatement returns [String javaString]:
437       DO WHILE expression {indent+=4;} statements LOOP
438       {
439         indent -= 4;
440         $javaString = getIndent() + "while ("+$expression.javaString+"){\n"
441         + $statements.javaString + "}\n";
442       }
443     | DO UNTIL expression {indent+=4;} statements LOOP
```

Appendix A (Continued)

```
444     {
445       indent -= 4;
446       $javaString =getIndent() + "do {\n"+$statements.javaString+"}\n"
447       +" while (!("+$expression.javaString+"));\n";
448     }
449   ;
450
451 forStatement returns [String javaString]:
452     FOR variableReference ASSIGN start=expression TO end=expression
453     {indent += 4;}
454     statements
455     NEXT
456     {
457       indent -= 4;
458       $javaString = getIndent() + "for (" +$variableReference.javaString+"="+
              $start.javaString+";"
459               + $variableReference.javaString +"<=" +$end.javaString+";"+
                    $variableReference.javaString+"++){\n"
460               + $statements.javaString
461               + "}\n";
462     }
463     ;
464
465 ifStatement returns [String javaString]:
466   IF conditionalExpression thenClause  {$javaString = getIndent() +
467    "if (" + $conditionalExpression.javaString+")"+$thenClause.javaString;}
```

Appendix A (Continued)

```
468   (elseifClause {$javaString+= $elseifClause.javaString;})*
469   (elseClause {$javaString += $elseClause.javaString;})?
470   END IF
471   ;
472
473 thenClause returns [String javaString]:
474   THEN {indent += 4;} statements
475   {
476     indent -= 4;
477     $javaString = getIndent() + "{\n"+$statements.javaString+getIndent()+"}\n";
478   }
479   ;
480
481 elseifClause returns [String javaString]:
482   ELSEIF conditionalExpression
483   thenClause
484   {
485     $javaString = getIndent()+" else if ("+ $conditionalExpression.javaString+")"
          +
486     $thenClause.javaString;
487   }
488   ;
489
490 elseClause returns[String javaString]:
491   ELSE {indent += 4;}statements
492   {
```

Appendix A (Continued)

```
493       indent -= 4;
494       $javaString = getIndent() + "else {\n"+$statements.javaString+getIndent()+"}\
              n";
495     }
496     ;
497
498 redimStatement returns[String javaString]:
499   REDIM IDENTIFIER LPAREN expression RPAREN
500     {
501       $javaString = getIndent() + "redim(" +$IDENTIFIER.text+", "+$expression.
              javaString+");\n";
502     }
503     ;
504
505 returnStatement returns [String javaString]:
506   RETURN
507     {
508       $javaString = getIndent()+"return "+(returnVariable!=null?returnVariable:"")+
              ";\n";
509       //TODO!!! add return variable here!
510     }
511     ;
512
513 selectCaseStatement returns [String javaString]
514 scope{
515       boolean useIfClause;
```

Appendix A (Continued)

```
516     int countCase;
517     String expression;
518     }
519   @init{
520     $selectCaseStatement::useIfClause = false;
521     $selectCaseStatement::countCase = 0;
522     $selectCaseStatement::expression = "";
523
524   }
525 :
526   SELECT CASE expression { //check the type of the expression
527     if($expression.type == Type.STRING){
528       $selectCaseStatement::useIfClause = true;
529       $selectCaseStatement::expression += $expression.javaString;
530       $javaString="";
531     }
532     else if($expression.type == Type.BOOLEAN){
533       $selectCaseStatement::useIfClause = true;
534       $selectCaseStatement::expression = null;
535       $javaString="";
536     }
537     //if the type of expression is float cast to int
538     else if($expression.type == Type.FLOAT){
539             $javaString = getIndent()+"switch ((int)"+$expression.javaString+"){\
                n"; indent+=4;
540     }
```

Appendix A (Continued)

```
541     else{
542             $javaString = getIndent()+"switch ("+$expression.javaString+"){\n";
                    indent+=4;
543     }
544     }
545   (caseClause  {
546     $selectCaseStatement::countCase++;
547     $javaString += $caseClause.javaString;})+
548   (caseElseClause  {$javaString += $caseElseClause.javaString;})?
549   END SELECT {indent -= 4;
550     if(!$selectCaseStatement::useIfClause)
551       $javaString += getIndent() + "}\n";
552     }
553   ;
554
555 caseClause returns [String javaString]:
556   (CASE expression{indent += 4;}  statements)
557   {if(!$selectCaseStatement::useIfClause){
558     indent-=4;
559     $javaString = getIndent() + "case "+$expression.javaString+" : \n"+
560     $statements.javaString +getIndent()+"     break;\n";
561   }
562   else{
563     if($selectCaseStatement::expression!=null){
564       indent-=4;
565       if($selectCaseStatement::countCase==0)
```

Appendix A (Continued)

```
566           $javaString = getIndent() + "if ("+$selectCaseStatement::expression+
567           ".compareTo("+$expression.javaString+")==0){\n"+$statements.javaString+
                  getIndent()+"}\n";
568         else
569           $javaString = getIndent() + "else if ("+$selectCaseStatement::expression+
570           ".compareTo("+$expression.javaString+")==0){\n"+$statements.javaString+
                  getIndent()+"}\n";
571         }
572       else{
573         indent-=4;
574         if($selectCaseStatement::countCase==0)
575           $javaString = getIndent() + "if ("+$expression.javaString+")"+
576           "{\n"+$statements.javaString+getIndent()+"}\n";
577         else
578           $javaString = getIndent() + "else if ("+$expression.javaString+")"+
579           "{\n"+$statements.javaString+getIndent()+"}\n";
580         }
581       }
582       }
583
584     ;
585
586
587   caseElseClause returns [String javaString]:
588     CASE ELSE {indent += 4;} statements
589     {indent-=4;
```

Appendix A (Continued)

```
590     if(!$selectCaseStatement::useIfClause)
591      $javaString = getIndent() + "default: \n"+$statements.javaString;
592
593     else
594       $javaString = getIndent() + "else{\n"+$statements.javaString+getIndent()+"}\
              n";
595
596   }
597   ;
598
599 conditionalExpression returns [String javaString]:
600   {insideCondition = true;}
601   expression
602   {$javaString = $expression.javaString;  insideCondition = false;}
603   ;
604
605 expression returns [String javaString, Type type]:
606   {$type = Type.ERROR;}
607   front=relationalExpression  {$javaString = $front.javaString;
608                                $type = $front.type;
609                                }
610   (
611    ( AND {$javaString += " && ";}
612    | OR  {$javaString += " || ";}
613    )
614     follow=relationalExpression {$javaString += $follow.javaString;}
```

Appendix A (Continued)

```
615    )*
616    ;
617
618
619  relationalExpression returns [String javaString, Type type]:
620    {$type = Type.ERROR;
621    boolean isFrontString = false;
622    boolean isFrontDate = false;
623    boolean isFrontPercentage = false;
624    boolean isNotEqualFound = false;
625    boolean isAssignFound = false;
626    boolean isLtFound = false;
627    boolean isLeFound = false;
628    boolean isGtFound = false;
629    boolean isGeFound = false;
630    }
631
632    front=additiveExpression {
633          try{
634              if($front.type == Type.STRING)
635                isFrontString = true;
636              else if($front.type == Type.DATE)
637                isFrontDate = true;
638              else if($front.type == Type.PERCENTAGE)
639                      isFrontPercentage = true;
640                      $javaString = $front.javaString;
```

Appendix A (Continued)

```
641                     $type = $front.type;
642             }//try
643             catch(Exception e){
644               System.out.println("ERROR---------- "+ $front.javaString+" "+ $front.
                      type.toString());
645             }
646
647     }
648     (
649       ( ASSIGN {  //oldcode
650                   //$javaString += insideCondition?"==":"=";
651                   isAssignFound = true;
652                   if(!isFrontString && !isFrontDate && !isFrontPercentage)
653                     if(insideCondition)
654                       $javaString += "==";
655                     else
656                       $javaString += "=";
657                   else{
658                     $javaString += ".compareTo(";
659                   }
660                 }
661       | NOT_EQUAL {
662                     isNotEqualFound = true;
663                     if(!isFrontString && !isFrontDate && !isFrontPercentage)
664                       $javaString += "!=";
665                     else{
```

Appendix A (Continued)

```
666                     //call to method compareTo
667                     $javaString += ".compareTo(";
668                   }
669                   }
670     | LT {
671           isLtFound = true;
672           if(!isFrontString && !isFrontDate && !isFrontPercentage)
673             $javaString += "<";
674           else{
675             //call to method compareTo
676             $javaString += ".compareTo(";
677           }
678         }
679     | LE {isLeFound = true;
680           if(!isFrontString && !isFrontDate && !isFrontPercentage)
681             $javaString += "<=";
682           else{
683             //call to method compareTo
684             $javaString += ".compareTo(";
685           }
686         }
687     | GT {isGtFound = true;
688           if(!isFrontString && !isFrontDate && !isFrontPercentage)
689             $javaString += ">";
690           else{
691             //call to method compareTo
```

Appendix A (Continued)

```
692             $javaString += ".compareTo(";
693           }
694         }
695     | GE {isLeFound = true;
696           if(!isFrontString && !isFrontDate && !isFrontPercentage)
697             $javaString += ">=";
698           else{
699             //call to method compareTo
700             $javaString += ".compareTo(";
701           }
702         } )
703     follow=additiveExpression  {
704       //manage comparison between Integer and Float type variables and vice-versa
705       if($front.type==Type.INTEGER && ($follow.type == Type.FLOAT||$follow.type
            == Type.PERCENTAGE))
706         $javaString += "(int)";
707       else if(($front.type==Type.FLOAT||$front.type==Type.PERCENTAGE) && $follow.
            type == Type.INTEGER)
708             $javaString += "(float)";
709             $javaString += $follow.javaString;
710             //if the follow is present the whole expression will be a boolean
                    expression
711             $type = Type.BOOLEAN;
712
713       //if one of the two is a string check if the types match
714       if(($follow.type==Type.STRING && (!isFrontString))||
```

Appendix A (Continued)

```
715         (!($follow.type==Type.STRING) && (isFrontString))){
716         System.out.println($follow.javaString+" : "+$follow.type.toString());
717         System.out.println($front.javaString+" : "+$front.type.toString());
718         throw new IllegalArgumentException("Relational expression not compatible
                with the grammar:\n"+
719                                            "One of the two expression operands
                                                is not a string\n");
720       }
721       if(($follow.type==Type.DATE && (!isFrontDate))||
722         (!($follow.type==Type.DATE) && (isFrontDate))){
723         System.out.println($follow.javaString+" : "+$follow.type.toString());
724         System.out.println($front.javaString+" : "+$front.type.toString());
725         throw new IllegalArgumentException("Relational expression not compatible
                with the grammar:\n"+
726                                           "One of the two expression operands is
                                                not a Date type\n");
727       }
728
729       if(isFrontString||isFrontDate||isFrontPercentage){
730         $javaString += ")";
731       if(isNotEqualFound)
732         $javaString += "!= 0";
733       else if(isAssignFound)
734         $javaString += "== 0";
735       else if(isLtFound)
736         $javaString += "< 0";
```

Appendix A (Continued)

```
737         else if(isLeFound)
738           $javaString += "<= 0";
739         else if(isGtFound)
740           $javaString += "> 0";
741         else if(isGeFound)
742           $javaString += ">= 0";
743       }
744       }
745     )*
746     ;
747
748
749 additiveExpression returns [String javaString, Type type]:
750   {$type = Type.ERROR;}
751   front=multiplicativeExpression  {$javaString = $front.javaString;
752                                    $type = $front.type;}
753   ((  PLUS {$javaString+="+";}
754      |MINUS{$javaString+="-";}
755      |CONCAT{$javaString+="+";})
756   follow=multiplicativeExpression {
757   $javaString += $follow.javaString;
758   //In case of addition between integers and float the type changes
759   if($front.type == Type.INTEGER){
760     if($follow.type ==Type.FLOAT)
761       $type = Type.FLOAT;
762     else if($follow.type != Type.INTEGER){
```

Appendix A (Continued)

```
763                 System.out.println($front.javaString+ " "+$front.type);
764                 System.out.println($follow.javaString+ " "+$follow.type);
765                 throw new IllegalArgumentException("Additive expression not
                        compatible with the grammar:\n");
766             }
767     }
768     })*
769     ;
770
771 multiplicativeExpression returns [String javaString, Type type]:
772     {$type = Type.ERROR;}
773     se=signedExpression {$javaString = $se.javaString;$type = $se.type;}
774     (
775      (STAR {$javaString+="*";}
776      |DIV {$javaString+="/";}
777     )
778     follow=signedExpression {
779       $javaString += $follow.javaString;
780       //In case of moltiplication between integers and float the type changes
781       if($se.type==Type.INTEGER){
782         if($follow.type ==Type.FLOAT)
783           $type = Type.FLOAT;
784         else if($follow.type != Type.INTEGER){
785                 System.out.println($se.javaString+ " "+$se.type);
786                 System.out.println($follow.javaString+ " "+$follow.type);
```

Appendix A (Continued)

```
787             throw new IllegalArgumentException("Multiplicative expression not
                    compatible with the grammar:\n");
788             }
789     }
790   })*
791   ;
792
793 signedExpression returns [String javaString, Type type]:
794   {$javaString = "";}(MINUS {$javaString += "-";})?
795   negationExpression {$javaString += $negationExpression.javaString;
796                       $type = $negationExpression.type;}
797   ;
798
799 negationExpression returns [String javaString, Type type]:
800   {$javaString = "";}( NOT {$javaString += "!";})?
801   primaryExpression {$javaString += $primaryExpression.javaString;
802                       $type = $primaryExpression.type;}
803   ;
804
805 primaryExpression returns[String javaString, Type type]:
806     functionCall {$type = $functionCall.type;
807                     $javaString = $functionCall.javaString;}
808   | variableReference {$javaString = $variableReference.javaString;
809                         $type = $variableReference.type;}
810   | literal {$javaString = $literal.javaString;
811               $type = $literal.type;}
```

Appendix A (Continued)

```
812    | LPAREN expression RPAREN  {$javaString = "("+$expression.javaString+")";
813                                    $type = $expression.type;}
814    ;
815
816  functionCall returns [String javaString, Type type]:
817    {boolean function = true;
818     $type = Type.NULL;
819     String functionName="";}
820
821    (DATE {$javaString="date";
822            functionName="date";
823            $type = $language::functionTable.findEntryType(functionName);}
824      |
825     IDENTIFIER {$javaString=$IDENTIFIER.text; function = !variables.contains(
           $IDENTIFIER.text);
826                 functionName = $IDENTIFIER.text;} )
827     LPAREN { if (function)
828                 { $javaString+="(";
829                   $type = $language::functionTable.findEntryType(functionName);
830                 }
831              else{  $javaString+="[";
832                     $type = $rule::symbolTable.findEntryType($IDENTIFIER.text);
833                     //System.out.print($IDENTIFIER.text+" ");
834                     //System.out.println($type);
835                 }
836            }
```

Appendix A (Continued)

```
837         (first=expression {$javaString+=$first.javaString;}
838           (COMMA follow=expression {
839           if(function)
840             $javaString+=", "+$follow.javaString;
841           else{
842             $javaString+="]["+follow.javaString;
843             $rule::symbolTable.increaseDimension($IDENTIFIER.text);
844           }
845           } )*
846         )?
847         RPAREN { if (function) $javaString+=")";  else  $javaString+="]";}
848     ;
849
850
851 literal returns [String javaString,Type type]:
852   {$type = Type.ERROR;}
853     INTEGER_LITERAL{$javaString =$INTEGER_LITERAL.text; $type = Type.INTEGER;}
854   | FLOAT_LITERAL{$javaString = "(float)"+$FLOAT_LITERAL.text; $type = Type.FLOAT
        ;}
855   | STRING_LITERAL{$javaString = $STRING_LITERAL.text; $type = Type.STRING;}
856   | NULL{$javaString = "null"; $type = Type.NULL;}
857   | TRUE{$javaString = "true"; $type = Type.BOOLEAN;}
858   | FALSE{$javaString = "false"; $type = Type.BOOLEAN;}
859   ;
860
861 variableReference returns [String javaString, Type type]:
```

Appendix A (Continued)

```
862   IDENTIFIER {$javaString = $IDENTIFIER.text;
863
864                $type = $rule::symbolTable.findEntryType($IDENTIFIER.text);
865                }
866   ;
867
868 arrayReference returns [String javaString, Type type, boolean isArrayAssignment]:
869     {boolean isArrayAss=true;}
870     IDENTIFIER {$javaString=$IDENTIFIER.text; $type = $rule::symbolTable.
            findEntryType($IDENTIFIER.text);}
871     LPAREN { $javaString+="[";
872            }
873       (first=expression {$javaString+=$first.javaString; isArrayAss=false;}
874         (COMMA follow=expression {
875           $javaString+="][”+follow.javaString;
876           $rule::symbolTable.increaseDimension($IDENTIFIER.text);
877         } )*
878       )?
879       RPAREN { $javaString+="]";
880                $isArrayAssignment=isArrayAss;}
881 ;
882
883
884 //Rule Header
885 //stored in a xml file
886 ruleSpecIdDeclaration returns[String xmlString]:
```

Appendix A (Continued)

```
887   {$xmlString = "<ruleSpecId>";}
888   RULE_SPEC_ID
889   (INTEGER_LITERAL{$xmlString += $INTEGER_LITERAL.text;})?
890   IDENTIFIER{ String s = "";
891               s = removeChar($IDENTIFIER.text,':');
892               s = removeChar(s,'/');
893               $rule::methodName += s;
894               $xmlString += $IDENTIFIER.text;
895               $xmlString += "</ruleSpecId>\n";
896               }
897   ;
898
899
900 ruleLongNameDeclaration returns[String xmlString]:
901   RULE_LONG_NAME{$xmlString = "<ruleLongName>"+ $RULE_LONG_NAME.text +"</
          ruleLongName>\n";}
902   ;
903
904 systemNameDeclaration returns[String xmlString]:
905 {$xmlString = "<systemName>";}
906   SYSTEM_NAME (INTEGER_LITERAL{$xmlString += $INTEGER_LITERAL.text;})?
907               (IDENTIFIER{ String s = "";
908                            s = removeChar($IDENTIFIER.text,':');
909                            s = removeChar(s,'/');
910                            $rule::methodName += s;
911                            $xmlString += $IDENTIFIER.text;})?
```

Appendix A (Continued)

```
912 {$xmlString += "</systemName>\n";}
913   ;
914
915 kindDeclaration returns[String xmlString]:
916   'KIND:' IDENTIFIER{$xmlString = "<kind>"+ $IDENTIFIER.text +"</kind>\n";}
917   ;
918
919 applicationDeclaration returns[String xmlString]:
920   {$xmlString = "<application>";}
921   (('(' 'Application:' (i1 = IDENTIFIER{$xmlString += $i1.text;$rule::methodName
        += $i1.text;})+
922   (';' i2=IDENTIFIER{$xmlString += ";"+$i2.text;$rule::methodName += $i2.text;})*
        ')')
923   |(i3=IDENTIFIER{$xmlString += $i3.text;$rule::methodName += $i3.text;} ')'))
924   {$xmlString += "</application>\n";}
925   ;
926
927 parentsDeclaration returns[String xmlString]:
928   {$xmlString = "<parent ";}
929   (parentIdDeclaration){$xmlString += "id='"+$parentIdDeclaration.xmlString+"'";}
930   (nameDeclaration){$xmlString += " name='"+$nameDeclaration.xmlString+"'";}
931   {$xmlString += "/>\n";}
932   ;
933
934 parentIdDeclaration returns[String xmlString]:
935   'PARENT ID:' INTEGER_LITERAL{$xmlString=$INTEGER_LITERAL.text;}
```

Appendix A (Continued)

```
936     ;
937
938  nameDeclaration returns[String xmlString]:
939     'NAME:' IDENTIFIER{       String s = "";
940                               s = removeChar($IDENTIFIER.text,'.');
941                               $rule::methodName += s;
942                               $xmlString=$IDENTIFIER.text;}
943     ;
944
945  interfaceDeclaration returns[String xmlString]:
946     {$xmlString = "<interface ";}
947     'INTERFACE NAME:' id1=IDENTIFIER{$xmlString +="name='"+$id1.text+"'";}
948     ('MODL:'){$xmlString +=" modl='";} (id2=IDENTIFIER{$xmlString +=$id2.text;})?{
          $xmlString +="'";}
949     ('PROTOCOL:' id3=IDENTIFIER){$xmlString +=" protocol='"+$id3.text+"'";}
950     ('TYPE:' id4=IDENTIFIER){$xmlString +=" type='"+$id4.text+"'";}
951     {$xmlString += "/>\n";}
952     ;
953
954  argPathDeclarations returns[String xmlString]
955  scope{
956        String argPath;
957     }
958     @init{
959        $argPathDeclarations::argPath="";
960     }
```

Appendix A (Continued)

```
961    :
962    {$xmlString="";}
963    argPathDecleration{
964      if($argPathDecleration.xmlString.length() > 0)
965        $argPathDeclarations::argPath = $argPathDecleration.xmlString+".";
966                          }
967    propertiesDeclaration{
968      $xmlString += $propertiesDeclaration.xmlString;
969      if($propertiesDeclaration.xmlString.length()==0)
970        $xmlString += "<property>"+$argPathDecleration.xmlString+"</property>\n";
971      }
972    ;
973
974  argPathDecleration returns[String xmlString]:
975    'ARG PATH:' IDENTIFIER{String s = "";
976      s = removeChar($IDENTIFIER.text,'.');
977      $rule::methodName += s;
978      $xmlString = $IDENTIFIER.text;}
979    ;
980
981  propertiesDeclaration returns[String xmlString]:
982    {$xmlString = "";}
983    'PROPERTIES:' (propertiesList{$xmlString += $propertiesList.xmlString;})?
984
985  ;
986
```

Appendix A (Continued)

```
987  propertiesList returns[String xmlString]:
988    (i1=IDENTIFIER{$xmlString = "<property>";$xmlString += $argPathDeclarations::
           argPath+$i1.text;$xmlString += "</property>\n";} ',' p=propertiesList{
           $xmlString += $p.xmlString;})
989    | (i2=IDENTIFIER{$xmlString = "<property>";$xmlString += $argPathDeclarations::
           argPath+$i2.text;$xmlString += "</property>\n";})
990  ;
991
992  sourceDeclaration:
993    'SOURCE'
994    ;
995
996  RECEIVES: 'RECEIVES' | 'Receives' | 'receives';
997
998  USES: 'uses' | 'Uses' | 'USES';
999
1000 CALL: 'CALL' | 'Call' | 'call';
1001
1002 IF: 'IF' | 'If' | 'if';
1003
1004 ELSEIF: 'elseif' | 'ELSEIF' | 'Elseif';
1005
1006 ELSE: 'ELSE' | 'Else' | 'else';
1007
1008 LOOP: 'loop' | 'LOOP' | 'Loop';
1009
```

Appendix A (Continued)

```
1010 NEXT: 'NEXT' | 'Next' | 'next';
1011
1012 REDIM: 'redim' | 'Redim' | 'REDIM';
1013
1014 FLOAT: 'FLOAT' | 'float' | 'Float';
1015
1016 AND: 'AND' | 'and' | 'And';
1017 OR: 'OR' | 'or' | 'Or';
1018 NOT: 'NOT' | 'Not' | 'not';
1019
1020 THEN: 'THEN' | 'Then' | 'then';
1021
1022 SELECT: 'SELECT'|'Select'|'select';
1023
1024 CASE: /*'CASE'|*/ 'Case' | 'case';
1025
1026 REQUIRED: 'REQUIRED' | 'Required' | 'required';
1027
1028 CURRENCY: 'CURRENCY' | 'Currency' | 'currency';
1029
1030 STRING: 'STRING' | 'String' | 'string';
1031
1032 DATE: 'DATE'| 'Date' | 'date';
1033
1034 INTEGER: 'INTEGER' | 'Integer' | 'integer';
1035
```

Appendix A (Continued)

```
1036 BOOLEAN: 'BOOLEAN' | 'Boolean' | 'boolean';
1037
1038 PERCENTAGE: 'PERCENTAGE' | 'Percentage' | 'percentage';
1039
1040 RETURNS: 'RETURNS' | 'Returns' | 'returns';
1041
1042 RETURN: 'RETURN' | 'Return' | 'return';
1043
1044 STRINGS: 'STRINGS' | 'Strings' | 'strings';
1045
1046 TRUE: 'true' | 'True' | 'TRUE';
1047 FALSE: 'FALSE' | 'False' | 'false';
1048 NULL: 'NULL' | 'Null' | 'null';
1049
1050 DO: 'DO' | 'Do' | 'do';
1051
1052 FOR: 'FOR' | 'For' | 'for';
1053
1054 TO: 'to' | 'TO' | 'To';
1055
1056 WHILE: 'WHILE' | 'While' | 'while';
1057
1058 UNTIL: 'UNTIL' | 'Until' | 'until';
1059
1060 END: 'end'| 'END' | 'End';
1061
```

Appendix A (Continued)

```
1062 CONCAT: '&';
1063 COMMA: ',';
1064 LPAREN: '(';
1065 RPAREN: ')';
1066 ASSIGN: '=';
1067 NOT_EQUAL: '<>';
1068 DIV: '/';
1069 PLUS: '+';
1070 MINUS: '-';
1071 STAR: '*';
1072 GE: '>=';
1073 GT: '>';
1074 LE: '<=';
1075 LT: '<';
1076
1077 INTEGER_LITERAL: ('0'..'9')+;
1078
1079 FLOAT_LITERAL: ('0'..'9')*('.'('0'..'9')+);
1080
1081 STRING_LITERAL: '"'('~'"')*'"';
1082
1083 IDENTIFIER:
1084    ('a'..'z'|'A'..'Z'|'_')('a'..'z'|'A'..'Z'|'_'|'0'..'9'|'.'|':'|'/')*  ;  //some
          symbols included
1085
1086 COMMENT:
```

Appendix A (Continued)

```
1087    '\'' (~('\n'))*('\n')?
1088    {$channel=HIDDEN;}
1089    ;
1090
1091 WS: ( ' '|'\t'|'\n'|'\r') {$channel=HIDDEN;};
1092
1093 RULE_SPEC_ID: 'RULE SPEC ID:';
1094
1095 SYSTEM_NAME : 'SYSTEM NAME:';
1096
1097 RULE_LONG_NAME: 'RULE LONG NAME:'(~('\n'))*('\n');
```

The other Java classes used in the parsing process are the following:

```
1  package com.accenture.techlabs.productmodel;
2
3  import java.util.ArrayList;
4
5  import com.accenture.techlabs.productmodel.RuleParserParser.Type;
6
7  public class SymbolTable {
8
9    private ArrayList<SymbolEntry> symbolTable;
10
11   public SymbolTable() {
12     symbolTable = new ArrayList<SymbolEntry>();
13   }
```

Appendix A (Continued)

```
14
15    public void addEntry(String name, Type type) {
16      SymbolEntry entry = new SymbolEntry(name, type, false);
17      symbolTable.add(entry);
18    }
19
20    public void addEntry(String name, Type type, boolean isProperty) {
21      SymbolEntry entry = new SymbolEntry(name, type, isProperty);
22      symbolTable.add(entry);
23    }
24
25    public Type findEntryType(String variableName) {
26      Type type = Type.NULL;
27      for (SymbolEntry s : symbolTable) {
28        if (s.getVariableName().compareTo(variableName) == 0) {
29          // System.out.println("FOUND");
30          type = s.getType();
31          break;
32        }
33      }
34      return type;
35    }
36
37    public ArrayList<SymbolEntry> getMultiDimArrays() {
38      ArrayList<SymbolEntry> arrays = new ArrayList<SymbolEntry>();
39
```

Appendix A (Continued)

```
40      for (SymbolEntry s : symbolTable) {
41        if (s.getDimension() > 1)
42          arrays.add(s);
43      }
44
45      return arrays;
46    }
47
48    public void increaseDimension(String arrayName) {
49      for (SymbolEntry s : symbolTable) {
50        if (s.getVariableName().compareTo(arrayName) == 0) {
51          s.setDimension(s.getDimension() + 1);
52        }
53      }
54    }
55
56    public void initializeFunctionTable() {
57      addEntry("addTime", Type.STRING);
58      addEntry("dateAdd", Type.DATE);
59      addEntry("date", Type.DATE);
60      addEntry("dateDiff", Type.INTEGER);
61      addEntry("dateSerial", Type.DATE);
62      addEntry("dateValue", Type.DATE);
63      addEntry("day", Type.INTEGER);
64      addEntry("month", Type.INTEGER);
65      addEntry("year", Type.INTEGER);
```

Appendix A (Continued)

```
66      addEntry("getDate", Type.DATE);
67      addEntry("getTimestamp", Type.STRING);
68      addEntry("left", Type.STRING);
69      addEntry("mid", Type.STRING);
70      addEntry("right", Type.STRING);
71      addEntry("trim", Type.STRING);
72      addEntry("addTime", Type.STRING);
73      addEntry("trimLeft", Type.STRING);
74      addEntry("trimRight", Type.STRING);
75      addEntry("replace", Type.STRING);
76      addEntry("ubound", Type.INTEGER);
77      addEntry("nextDimensionElementCount", Type.INTEGER);
78      addEntry("elementCount", Type.INTEGER);
79      addEntry("round", Type.FLOAT);
80      addEntry("isNull", Type.BOOLEAN);
81      addEntry("initializeValue", Type.NULL);
82      addEntry("error", Type.INTEGER);
83      addEntry("message", Type.INTEGER);
84      addEntry("attachError", Type.INTEGER);
85      addEntry("attachMessage", Type.INTEGER);
86      addEntry("getGlobal", Type.STRING);
87      addEntry("getGlobalInt", Type.INTEGER);
88      addEntry("getGlobalFloat", Type.FLOAT);
89
90  }
91
```

Appendix A (Continued)

```
92    public void printSymbolTable() {
93      for (SymbolEntry e : symbolTable) {
94        System.out.println(e.getVariableName() + " " + e.getDimension());
95      }
96    }
97
98    public String toXmlString() {
99      String xmlString = "<variablesName>\n";
100     for (SymbolEntry e : symbolTable) {
101       if (e.isProperty())
102         xmlString += "<variable>" + e.getVariableName()
103             + "</variable>\n";
104     }
105     xmlString += "</variablesName>\n";
106     return xmlString;
107   }
108 }
```

```
1  package com.accenture.techlabs.productmodel;
2
3  import com.accenture.techlabs.productmodel.RuleParserParser.Type;
4
5  public class SymbolEntry {
6
7    private String variableName = "";
8    private Type type;
```

Appendix A (Continued)

```
 9   private boolean isProperty;
10   private int dimension;
11
12   public SymbolEntry(String name, Type type, boolean isProperty) {
13     this.variableName = this.variableName.concat(name);
14     this.type = type;
15     this.dimension = 0;
16     this.isProperty = isProperty;
17   }
18
19   public void setVariableName(String variableName) {
20     this.variableName = variableName;
21   }
22
23   public String getVariableName() {
24     return variableName;
25   }
26
27   public void setType(Type type) {
28     this.type = type;
29   }
30
31   public Type getType() {
32     return type;
33   }
34
```

Appendix A (Continued)

```
35    public void setDimension(int dimension) {
36      this.dimension = dimension;
37    }
38
39    public int getDimension() {
40      return dimension;
41    }
42
43    public void setProperty(boolean isProperty) {
44      this.isProperty = isProperty;
45    }
46
47    public boolean isProperty() {
48      return isProperty;
49    }
50  }
```

```
1  package com.accenture.techlabs.productmodel;
2
3  import java.io.IOException;
4
5  import org.antlr.runtime.ANTLRFileStream;
6  import org.antlr.runtime.CharStream;
7  import org.antlr.runtime.CommonTokenStream;
8  import org.antlr.runtime.RecognitionException;
9
```

Appendix A (Continued)

```
10 public class Translator {
11
12   /**
13    * @param args
14    * @throws IOException
15    * @throws RecognitionException
16    */
17   public static void main(String[] args) throws IOException,
18       RecognitionException {
19
20     CharStream cs = new ANTLRFileStream("DOWN.txt");
21     RuleParserLexer lexer = new RuleParserLexer(cs);
22     CommonTokenStream tokens = new CommonTokenStream();
23     tokens.setTokenSource(lexer);
24     RuleParserParser parser = new RuleParserParser(tokens);
25     String javaString = parser.language();
26
27     System.out.println(javaString);
28
29   }
30
31 }
```

Appendix B

DATA AND CONTROL FLOW ANAYISIS SOURCE CODE

B.1 Read and Write Analysis

The read and write analysis part of our project consists of four different analyses that identify four categories of results: read, written, and possibly read and written variables. This is the source code that deals with the read and write data flow analysis:

```
1  import java.util.ArrayList;
2  import java.util.Collections;
3  import java.util.HashMap;
4  import java.util.Iterator;
5  import java.util.List;
6  import java.util.Map;
7
8  import soot.Body;
9  import soot.Unit;
10 import soot.ValueBox;
11 import soot.jimple.AssignStmt;
12 import soot.jimple.IfStmt;
13 import soot.jimple.InvokeStmt;
14 import soot.jimple.ReturnStmt;
15 import soot.jimple.Stmt;
16 import soot.jimple.TableSwitchStmt;
```

Appendix B (Continued)

```
17 import soot.toolkits.graph.DominatorsFinder;
18 import soot.toolkits.graph.MHGDominatorsFinder;
19 import soot.toolkits.graph.UnitGraph;
20 import soot.toolkits.scalar.ArraySparseSet;
21 import soot.toolkits.scalar.FlowSet;
22 import soot.toolkits.scalar.ForwardFlowAnalysis;
23
24 public class ReadWriteAnalyzer {
25
26   private Map writtenVars;
27   private Map readVars;
28   private Map mayWrittenVars;
29   private Map mayReadVars;
30
31   public ReadWriteAnalyzer(UnitGraph graph, List<String> properties) {
32
33     List<String> parameters = properties;
34     // Perform the analysis
35     WriteParametersAnalysis writeAnalysis = new WriteParametersAnalysis(
36         graph, parameters);
37     ReadParametersAnalysis readAnalysis = new ReadParametersAnalysis(graph,
38         parameters);
39     MayWriteParametersAnalysis mayWriteAnalysis = new MayWriteParametersAnalysis(
40         graph, parameters);
41     MayReadParametersAnalysis mayReadAnalysis = new MayReadParametersAnalysis(
42         graph, parameters);
```

Appendix B (Continued)

```
43
44      // data structure to keep the var lists
45      writtenVars = new HashMap(graph.size() * 2 + 1, 0.7f);
46      readVars = new HashMap(graph.size() * 2 + 1, 0.7f);
47      mayWrittenVars = new HashMap(graph.size() * 2 + 1, 0.7f);
48      mayReadVars = new HashMap(graph.size() * 2 + 1, 0.7f);
49
50      Iterator unitIt = graph.iterator();
51
52      while (unitIt.hasNext()) {
53        Unit s = (Unit) unitIt.next();
54
55        FlowSet set0 = (FlowSet) writeAnalysis.getFlowAfter(s);
56        FlowSet set1 = (FlowSet) readAnalysis.getFlowAfter(s);
57        FlowSet set2 = (FlowSet) mayWriteAnalysis.getFlowAfter(s);
58        FlowSet set3 = (FlowSet) mayReadAnalysis.getFlowAfter(s);
59
60        writtenVars.put(s, Collections.unmodifiableList(set0.toList()));
61        readVars.put(s, Collections.unmodifiableList(set1.toList()));
62        mayWrittenVars.put(s, Collections.unmodifiableList(set2.toList()));
63        mayReadVars.put(s, Collections.unmodifiableList(set3.toList()));
64      }
65    }
66
67    public List getWParameters(Unit s) {
68      return (List) writtenVars.get(s);
```

Appendix B (Continued)

```
69    }
70
71    public List getRParameters(Unit s) {
72      return (List) readVars.get(s);
73    }
74
75    public List getMWParameters(Unit s) {
76      return (List) mayWrittenVars.get(s);
77    }
78
79    public List getMRParameters(Unit s) {
80      return (List) mayReadVars.get(s);
81    }
82  }
83
84  class WriteParametersAnalysis extends ForwardFlowAnalysis {
85    FlowSet emptySet = new ArraySparseSet();
86    private List<String> parameters;
87
88    WriteParametersAnalysis(UnitGraph graph, List<String> parameters) {
89      super(graph);
90      this.parameters = parameters;
91      doAnalysis();
92    }
93
94    /**
```

Appendix B (Continued)

```
95     * All INs are initialized to the empty set.
96     **/
97    protected Object newInitialFlow() {
98      return emptySet.clone();
99    }
100
101   /**
102    * IN(Start) is the empty set
103    **/
104   protected Object entryInitialFlow() {
105     return emptySet.clone();
106   }
107
108   /**
109    * OUT is the same as IN plus the genSet.
110    **/
111   protected void flowThrough(Object inValue, Object unit, Object outValue) {
112     FlowSet in = (FlowSet) inValue, out = (FlowSet) outValue;
113     Unit u = (Unit) unit;
114
115     MHGDominatorsFinder df = new MHGDominatorsFinder(graph);
116
117     // check if this unit is the head of a loop or solve a problem with
118     // multi-headed graphs
119     List<Unit> heads = graph.getHeads();
120     List<Unit> preds = graph.getPredsOf(u);
```

Appendix B (Continued)

```
121     for (Unit a : preds) {
122       for (Object b : df.getDominators(a))
123         if (b.equals(u)) {
124           // then we are in the head of a loop and we copy the written
125           // Vars from the previous unit
126           for (Unit c : preds)
127             if (!c.equals(a)) {
128               for (Object var : ((FlowSet) this.getFlowAfter(c))
129                   .toList())
130                 in.add((String) var);
131             }
132
133         }
134       // solve a problem with multi headed graphs
135       if (heads.contains(a) && preds.size() > 1) {
136         for (Unit c : preds)
137           if (!c.equals(a)) {
138             for (Object var : ((FlowSet) this.getFlowAfter(c))
139                 .toList())
140               in.add((String) var);
141           }
142       }
143     }
144
145     copy(in, out);
146     // (kill set is empty)
```

Appendix B (Continued)

```
147     gen(out, u);
148   }
149
150   private void gen(FlowSet out, Unit u) {
151     Iterator defIt;
152     if (u instanceof AssignStmt) {
153       defIt = u.getDefBoxes().iterator();
154       while (defIt.hasNext()) {
155         ValueBox defBox = (ValueBox) defIt.next();
156         String var = defBox.getValue().toString();
157         if (parameters.contains(var))
158           out.add(var);
159         // array check
160
161         if (((AssignStmt) u).containsArrayRef()) {
162           Iterator i1 = defBox.getValue().getUseBoxes().iterator();
163           while (i1.hasNext()) {
164             ValueBox parameter = (ValueBox) i1.next();
165             if (parameter.getValue().equals(
166                 ((AssignStmt) u).getArrayRef().getBase()))
167               out.add(findValueWrittenIn(u, parameter));
168           }
169         }
170
171       }
172     } else if (u instanceof ReturnStmt) {
```

Appendix B (Continued)

```
173         ValueBox b = ((ReturnStmt) u).getOpBox();
174         String var = b.getValue().toString();
175         if (parameters.contains(var))
176           out.add(var);
177         else {
178           out.add(findValueWrittenIn(u, ((ReturnStmt) u).getOpBox()));
179         }
180
181       }
182     }
183
184     private String findValueWrittenIn(Unit u, ValueBox arrayBase) {
185       {
186         DominatorsFinder df = new MHGDominatorsFinder(graph);
187         Unit unitFound = null;
188         for (Iterator domsIt = df.getDominators(u).iterator(); domsIt
189             .hasNext();) {
190
191           Unit dom = (Unit) domsIt.next();
192
193           Iterator arrit = dom.getDefBoxes().iterator();
194           while (arrit.hasNext()) {
195             ValueBox arr = (ValueBox) arrit.next();
196             if (arr.getValue().equals(arrayBase.getValue())) {
197               unitFound = dom;
198             }
```

Appendix B (Continued)

```
199           }
200         }
201         // recursion
202         if (((Stmt) unitFound).containsArrayRef()) {
203           String s = findValueWrittenIn(unitFound, ((Stmt) unitFound)
204               .getArrayRef().getBaseBox());
205           return s;
206         } else {
207           String s1 = "";
208           for (Iterator paramfoundIt = unitFound.getUseBoxes().iterator();
                  paramfoundIt
209               .hasNext();) {
210             ValueBox paramFound = (ValueBox) paramfoundIt.next();
211             s1 += paramFound.getValue().toString();
212           }
213           return s1;
214         }
215
216       }
217
218     }
219
220     /**
221      * All paths == Intersection.
222      **/
223     protected void merge(Object in1, Object in2, Object out) {
```

Appendix B (Continued)

```
224     FlowSet inSet1 = (FlowSet) in1, inSet2 = (FlowSet) in2, outSet = (FlowSet)
            out;
225
226     inSet1.intersection(inSet2, outSet);
227   }
228
229   protected void copy(Object source, Object dest) {
230     FlowSet sourceSet = (FlowSet) source, destSet = (FlowSet) dest;
231
232     sourceSet.copy(destSet);
233   }
234 }
235
236 class MayWriteParametersAnalysis extends ForwardFlowAnalysis {
237   FlowSet emptySet = new ArraySparseSet();
238   private List<String> parameters;
239
240   MayWriteParametersAnalysis(UnitGraph graph, List<String> parameters) {
241     super(graph);
242     this.parameters = parameters;
243     doAnalysis();
244   }
245
246   /**
247    * All INs are initialized to the empty set.
248    **/
```

Appendix B (Continued)

```
249   protected Object newInitialFlow() {
250     return emptySet.clone();
251   }
252
253   /**
254    * IN(Start) is the empty set
255    **/
256   protected Object entryInitialFlow() {
257     return emptySet.clone();
258   }
259
260   /**
261    * OUT is the same as IN plus the genSet.
262    **/
263   protected void flowThrough(Object inValue, Object unit, Object outValue) {
264     FlowSet in = (FlowSet) inValue, out = (FlowSet) outValue;
265     Unit u = (Unit) unit;
266     // copy(in, out);
267     // (kill set is empty)
268     gen(out, u);
269   }
270
271   private void gen(FlowSet out, Unit u) {
272     Iterator defIt;
273     if (u instanceof AssignStmt) {
274       defIt = u.getDefBoxes().iterator();
```

Appendix B (Continued)

```
275         while (defIt.hasNext()) {
276           ValueBox defBox = (ValueBox) defIt.next();
277           String var = defBox.getValue().toString();
278           if (parameters.contains(var))
279             out.add(var);
280           // array check
281
282           if (((AssignStmt) u).containsArrayRef()) {
283             Iterator i1 = defBox.getValue().getUseBoxes().iterator();
284             while (i1.hasNext()) {
285               ValueBox parameter = (ValueBox) i1.next();
286               if (parameter.getValue().equals(
287                   ((AssignStmt) u).getArrayRef().getBase()))
288                 out.add(findValueWrittenIn(u, parameter));
289             }
290           }
291         }
292       } else if (u instanceof ReturnStmt) {
293         ValueBox b = ((ReturnStmt) u).getOpBox();
294         String var = b.getValue().toString();
295         if (parameters.contains(var))
296           out.add(var);
297         else {
298           out.add(findValueWrittenIn(u, ((ReturnStmt) u).getOpBox()));
299         }
300       }
```

Appendix B (Continued)

```
301   }
302
303   private String findValueWrittenIn(Unit u, ValueBox arrayBase) {
304
305     DominatorsFinder df = new MHGDominatorsFinder(graph);
306     Unit unitFound = null;
307     for (Iterator domsIt = df.getDominators(u).iterator(); domsIt.hasNext();) {
308
309       Unit dom = (Unit) domsIt.next();
310
311       Iterator arrit = dom.getDefBoxes().iterator();
312       while (arrit.hasNext()) {
313         ValueBox arr = (ValueBox) arrit.next();
314         if (arr.getValue().equals(arrayBase.getValue())) {
315           unitFound = dom;
316         }
317       }
318     }
319     // recursion
320     if (((Stmt) unitFound).containsArrayRef()) {
321       String s = findValueWrittenIn(unitFound, ((Stmt) unitFound)
322           .getArrayRef().getBaseBox());
323       return s;
324     } else {
325       String s1 = "";
```

Appendix B (Continued)

```
326         for (Iterator paramfoundIt = unitFound.getUseBoxes().iterator();
                paramfoundIt
327              .hasNext();) {
328           ValueBox paramFound = (ValueBox) paramfoundIt.next();
329           s1 += paramFound.getValue().toString();
330         }
331         return s1;
332       }
333     }
334
335     /**
336      * All paths == Intersection.
337      **/
338     protected void merge(Object in1, Object in2, Object out) {
339       FlowSet inSet1 = (FlowSet) in1, inSet2 = (FlowSet) in2, outSet = (FlowSet)
              out;
340
341       inSet1.intersection(inSet2, outSet);
342     }
343
344     protected void copy(Object source, Object dest) {
345       FlowSet sourceSet = (FlowSet) source, destSet = (FlowSet) dest;
346
347       sourceSet.copy(destSet);
348     }
349 }
```

Appendix B (Continued)

```
350
351 class ReadParametersAnalysis extends ForwardFlowAnalysis {
352   FlowSet emptySet = new ArraySparseSet();
353   private List<String> parameters;
354   private List<String> arrayWrittenBases;
355
356   ReadParametersAnalysis(UnitGraph graph, List<String> parameters) {
357     super(graph);
358     this.parameters = parameters;
359     this.arrayWrittenBases = getArrayWrittenBases(graph);
360     // this.arrayWrittenBases = new ArrayList<Value>();
361     doAnalysis();
362   }
363
364   /**
365    * All INs are initialized to the empty set.
366    **/
367   protected Object newInitialFlow() {
368     return emptySet.clone();
369   }
370
371   /**
372    * IN(Start) is the empty set
373    **/
374   protected Object entryInitialFlow() {
375     return emptySet.clone();
```

Appendix B (Continued)

```
376   }
377
378   /**
379    * OUT is the same as IN plus the genSet.
380    **/
381   protected void flowThrough(Object inValue, Object unit, Object outValue) {
382     FlowSet in = (FlowSet) inValue, out = (FlowSet) outValue;
383     Unit u = (Unit) unit;
384
385     MHGDominatorsFinder df = new MHGDominatorsFinder(graph);
386
387     // check if this unit is the head of a loop or solve a problem with
388     // multi-headed graphs
389     List<Unit> heads = graph.getHeads();
390     List<Unit> preds = graph.getPredsOf(u);
391     for (Unit a : preds) {
392       for (Object b : df.getDominators(a))
393         if (b.equals(u)) {
394           // then we are in the head of a loop and we copy the written
395           // Vars from the previous unit
396           for (Unit c : preds)
397             if (!c.equals(a)) {
398               for (Object var : ((FlowSet) this.getFlowAfter(c))
399                   .toList())
400                 in.add((String) var);
401             }
```

Appendix B (Continued)

```
402
403         }
404       // solve a problem with multi headed graphs
405       if (heads.contains(a) && preds.size() > 1) {
406         for (Unit c : preds)
407           if (!c.equals(a)) {
408             for (Object var : ((FlowSet) this.getFlowAfter(c))
409                 .toList())
410
411               in.add((String) var);
412           }
413       }
414     }
415     copy(in, out);
416     // (kill set is empty)
417     gen(out, u);
418   }
419
420   private void gen(FlowSet out, Unit u) {
421
422     if (u instanceof InvokeStmt || u instanceof IfStmt
423         || u instanceof TableSwitchStmt) {
424       Iterator defIt = u.getUseBoxes().iterator();
425       while (defIt.hasNext()) {
426
427         ValueBox useBox = (ValueBox) defIt.next();
```

Appendix B (Continued)

```
428         String var = useBox.getValue().toString();
429         if (parameters.contains(var))
430           out.add(var);
431       }
432     } else if (u instanceof AssignStmt) {
433
434       Iterator it = u.getUseBoxes().iterator();
435       while (it.hasNext()) {
436
437         ValueBox useBox = (ValueBox) it.next();
438         String var = useBox.getValue().toString();
439         // check if it is used in a following stmt as a base of an array
440         // that is actually written
441         // if not add the var to out
442         if (parameters.contains(var))
443           if (!arrayWrittenBases.contains(((AssignStmt) u)
444               .getLeftOp().toString()))
445             out.add(var);
446       }
447
448       it = u.getDefBoxes().iterator();
449       while (it.hasNext()) {
450         ValueBox defBox = (ValueBox) it.next();
451
452         // if it is an array index see what's written in the
453         // temporary var
```

Appendix B (Continued)

```
454         if (((AssignStmt) u).containsArrayRef()) {
455           Iterator i1 = defBox.getValue().getUseBoxes().iterator();
456           while (i1.hasNext()) {
457             ValueBox parameter = (ValueBox) i1.next();
458             if (parameter.getValue().equals(
459                 ((AssignStmt) u).getArrayRef().getIndex())) {
460               DominatorsFinder df = new MHGDominatorsFinder(graph);
461               for (Iterator domsIt = df.getDominators(u)
462                   .iterator(); domsIt.hasNext();) {
463
464                 Unit dom = (Unit) domsIt.next();
465
466                 Iterator arrit = dom.getDefBoxes().iterator();
467                 while (arrit.hasNext()) {
468                   ValueBox arr = (ValueBox) arrit.next();
469                   if (arr.getValue().equals(
470                       parameter.getValue())) {
471
472                     for (Iterator paramfoundIt = dom
473                         .getUseBoxes().iterator(); paramfoundIt
474                         .hasNext();) {
475                       ValueBox paramFound = (ValueBox) paramfoundIt
476                           .next();
477                       String var = paramFound.getValue()
478                           .toString();
479                       if (parameters.contains(var))
```

Appendix B (Continued)

```
480                             out.add(var);
481                         }
482                       }
483
484                     }
485
486                   }
487
488                 }
489               }
490             }
491
492           }
493
494       }
495
496   }
497
498   private List<String> getArrayWrittenBases(UnitGraph g) {
499     List<String> arrayBases = new ArrayList<String>();
500     Body b = g.getBody();
501     Iterator unitIt = b.getUnits().iterator();
502     // For all the statements in the method body
503     while (unitIt.hasNext()) {
504       Stmt st = (Stmt) unitIt.next();
505       Iterator defIt = st.getDefBoxes().iterator();
```

Appendix B (Continued)

```
506       // if it is an assignment statement and contains an array
507       if (st.containsArrayRef() && st instanceof AssignStmt) {
508         while (defIt.hasNext()) {
509           ValueBox defBox = (ValueBox) defIt.next();
510
511           Iterator i1 = defBox.getValue().getUseBoxes().iterator();
512           while (i1.hasNext()) {
513             ValueBox useBox = (ValueBox) i1.next();
514             // add the arraybase to the list
515             if (useBox.getValue().equals(
516                 ((AssignStmt) st).getArrayRef().getBase()))
517               arrayBases.add(st.getArrayRef().getBase()
518                   .toString());
519           }
520         }
521       }
522       // if it is a return statement and an array is returned add the base
523       // to the list
524       else if (st instanceof ReturnStmt) {
525         arrayBases.add(findArrayBase(st,
526             ((ReturnStmt) st).getOpBox()));
527       }
528     }
529     return arrayBases;
530   }
531
```

Appendix B (Continued)

```
532   private String findArrayBase(Unit u, ValueBox arrayBase) {
533     DominatorsFinder df = new MHGDominatorsFinder(graph);
534     Unit unitFound = null;
535     for (Iterator domsIt = df.getDominators(u).iterator(); domsIt.hasNext();) {
536
537       Unit dom = (Unit) domsIt.next();
538
539       Iterator arrit = dom.getDefBoxes().iterator();
540       while (arrit.hasNext()) {
541         ValueBox arr = (ValueBox) arrit.next();
542         if (arr.getValue().equals(arrayBase.getValue())) {
543           unitFound = dom;
544         }
545       }
546     }
547     // recursion
548     if (((Stmt) unitFound).containsArrayRef()) {
549       String s = findArrayBase(unitFound, ((Stmt) unitFound)
550           .getArrayRef().getBaseBox());
551       return s;
552     } else {
553       String s1 = "";
554       for (Iterator paramfoundIt = unitFound.getDefBoxes().iterator();
              paramfoundIt
555           .hasNext();) {
556         ValueBox paramFound = (ValueBox) paramfoundIt.next();
```

Appendix B (Continued)

```
557           s1 += paramFound.getValue().toString();
558         }
559         return s1;
560       }
561
562     }
563
564     /**
565      * All paths == Intersection.
566      **/
567     protected void merge(Object in1, Object in2, Object out) {
568       FlowSet inSet1 = (FlowSet) in1, inSet2 = (FlowSet) in2, outSet = (FlowSet)
              out;
569
570       inSet1.intersection(inSet2, outSet);
571     }
572
573     protected void copy(Object source, Object dest) {
574       FlowSet sourceSet = (FlowSet) source, destSet = (FlowSet) dest;
575
576       sourceSet.copy(destSet);
577     }
578 }
579
580 class MayReadParametersAnalysis extends ForwardFlowAnalysis {
581   FlowSet emptySet = new ArraySparseSet();
```

Appendix B (Continued)

```
582   private List<String> parameters;
583   private List<String> arrayWrittenBases;
584
585   MayReadParametersAnalysis(UnitGraph graph, List<String> parameters) {
586     super(graph);
587     this.parameters = parameters;
588     this.arrayWrittenBases = getArrayWrittenBases(graph);
589     // this.arrayWrittenBases = new ArrayList<Value>();
590     doAnalysis();
591   }
592
593   /**
594    * All INs are initialized to the empty set.
595    **/
596   protected Object newInitialFlow() {
597     return emptySet.clone();
598   }
599
600   /**
601    * IN(Start) is the empty set
602    **/
603   protected Object entryInitialFlow() {
604     return emptySet.clone();
605   }
606
607   /**
```

Appendix B (Continued)

```
608     * OUT is the same as IN plus the genSet.
609     **/
610    protected void flowThrough(Object inValue, Object unit, Object outValue) {
611      FlowSet in = (FlowSet) inValue, out = (FlowSet) outValue;
612      Unit u = (Unit) unit;
613      // copy(in, out);
614      // (kill set is empty)
615      gen(out, u);
616    }
617
618    private void gen(FlowSet out, Unit u) {
619
620      if (u instanceof InvokeStmt || u instanceof IfStmt
621          || u instanceof TableSwitchStmt) {
622        Iterator defIt = u.getUseBoxes().iterator();
623        while (defIt.hasNext()) {
624
625          ValueBox useBox = (ValueBox) defIt.next();
626          String var = useBox.getValue().toString();
627          // if (!out.contains(var))
628          if (parameters.contains(var))
629            out.add(var);
630        }
631      } else if (u instanceof AssignStmt) {
632
633        Iterator defIt = u.getUseBoxes().iterator();
```

Appendix B (Continued)

```
634         while (defIt.hasNext()) {
635
636           ValueBox useBox = (ValueBox) defIt.next();
637           String var = useBox.getValue().toString();
638           // check if it is used in a following stmt as a base of an array
639           // that is actually written
640           // if not add the var to out
641
642           if (parameters.contains(var))
643             if (!arrayWrittenBases.contains(((AssignStmt) u)
644                 .getLeftOp().toString()))
645               out.add(var);
646         }
647
648         defIt = u.getDefBoxes().iterator();
649         while (defIt.hasNext()) {
650           ValueBox defBox = (ValueBox) defIt.next();
651
652           // if it is an array index see what's written in the
653           // temporary var
654           if (((AssignStmt) u).containsArrayRef()) {
655             Iterator i1 = defBox.getValue().getUseBoxes().iterator();
656             while (i1.hasNext()) {
657               ValueBox parameter = (ValueBox) i1.next();
658               if (parameter.getValue().equals(
659                   ((AssignStmt) u).getArrayRef().getIndex())) {
```

Appendix B (Continued)

```
660             DominatorsFinder df = new MHGDominatorsFinder(graph);
661             for (Iterator domsIt = df.getDominators(u)
662                 .iterator(); domsIt.hasNext();) {
663
664               Unit dom = (Unit) domsIt.next();
665
666               Iterator arrit = dom.getDefBoxes().iterator();
667               while (arrit.hasNext()) {
668                 ValueBox arr = (ValueBox) arrit.next();
669                 if (arr.getValue().equals(
670                     parameter.getValue())) {
671
672                   for (Iterator paramfoundIt = dom
673                       .getUseBoxes().iterator(); paramfoundIt
674                       .hasNext();) {
675                     ValueBox paramFound = (ValueBox) paramfoundIt
676                         .next();
677                     String var = paramFound.getValue()
678                         .toString();
679                     if (parameters.contains(var))
680                       out.add(var);
681                   }
682                 }
683
684               }
685
```

Appendix B (Continued)

```
686                 }
687
688               }
689             }
690           }
691
692         }
693
694       }
695
696   }
697
698   private List<String> getArrayWrittenBases(UnitGraph g) {
699     List<String> arrayBases = new ArrayList<String>();
700     Body b = g.getBody();
701
702     Iterator unitIt = b.getUnits().iterator();
703     while (unitIt.hasNext()) {
704       Stmt st = (Stmt) unitIt.next();
705       Iterator defIt = st.getDefBoxes().iterator();
706       if (st.containsArrayRef() && st instanceof AssignStmt) {
707         while (defIt.hasNext()) {
708           ValueBox defBox = (ValueBox) defIt.next();
709
710           Iterator i1 = defBox.getValue().getUseBoxes().iterator();
711           while (i1.hasNext()) {
```

Appendix B (Continued)

```
712             ValueBox useBox = (ValueBox) i1.next();
713             if (useBox.getValue().equals(
714                 ((AssignStmt) st).getArrayRef().getBase()))
715               arrayBases.add(st.getArrayRef().getBase()
716                   .toString());
717           }
718         }
719       } else if (st instanceof ReturnStmt) {
720         arrayBases.add(findArrayBase(st, ((ReturnStmt) st).getOpBox()));
721       }
722     }
723     return arrayBases;
724   }
725
726   private String findArrayBase(Unit u, ValueBox arrayBase) {
727     {
728       DominatorsFinder df = new MHGDominatorsFinder(graph);
729       Unit unitFound = null;
730       for (Iterator domsIt = df.getDominators(u).iterator(); domsIt
731           .hasNext();) {
732
733         Unit dom = (Unit) domsIt.next();
734
735         Iterator arrit = dom.getDefBoxes().iterator();
736         while (arrit.hasNext()) {
737           ValueBox arr = (ValueBox) arrit.next();
```

Appendix B (Continued)

```
738             if (arr.getValue().equals(arrayBase.getValue())) {
739               unitFound = dom;
740             }
741           }
742         }
743         // recursion
744         if (((Stmt) unitFound).containsArrayRef()) {
745           String s = findArrayBase(unitFound, ((Stmt) unitFound)
746               .getArrayRef().getBaseBox());
747           return s;
748         } else {
749           String s1 = "";
750           for (Iterator paramfoundIt = unitFound.getDefBoxes().iterator();
                  paramfoundIt
751               .hasNext();) {
752             ValueBox paramFound = (ValueBox) paramfoundIt.next();
753             s1 += paramFound.getValue().toString();
754           }
755           return s1;
756         }
757
758       }
759
760   }
761
762   /**
```

Appendix B (Continued)

```
763     * All paths == Intersection.
764     **/
765    protected void merge(Object in1, Object in2, Object out) {
766      FlowSet inSet1 = (FlowSet) in1, inSet2 = (FlowSet) in2, outSet = (FlowSet)
           out;
767
768      inSet1.intersection(inSet2, outSet);
769    }
770
771    protected void copy(Object source, Object dest) {
772      FlowSet sourceSet = (FlowSet) source, destSet = (FlowSet) dest;
773
774      sourceSet.copy(destSet);
775    }
776  }
```

B.2 Path Condition Analysis

The path condition analysis deals with identifying a path that reaches a statement and discovering the conditions to be satisfied to reach that statement through that specific path. This analysis is implemented in the *PathConditionAnalyzer* class whose source code is the following:

```
1  import java.util.Collections;
2  import java.util.HashMap;
3  import java.util.Iterator;
4  import java.util.List;
```

Appendix B (Continued)

```
5  import java.util.Map;
6
7  import soot.Local;
8  import soot.Unit;
9  import soot.Value;
10 import soot.ValueBox;
11 import soot.jimple.ArrayRef;
12 import soot.jimple.AssignStmt;
13 import soot.jimple.BinopExpr;
14 import soot.jimple.ConditionExpr;
15 import soot.jimple.IdentityStmt;
16 import soot.jimple.IfStmt;
17 import soot.jimple.InvokeExpr;
18 import soot.jimple.VirtualInvokeExpr;
19 import soot.toolkits.graph.DominatorsFinder;
20 import soot.toolkits.graph.MHGDominatorsFinder;
21 import soot.toolkits.graph.UnitGraph;
22 import soot.toolkits.scalar.ArraySparseSet;
23 import soot.toolkits.scalar.FlowSet;
24 import soot.toolkits.scalar.ForwardBranchedFlowAnalysis;
25
26 public class PathConditionAnalyzer {
27   private Map conditions1, conditions2;
28
29   /**
30    * Perform the analysis and store the results in two hash maps
```

Appendix B (Continued)

```
31    **/
32   public PathConditionAnalyzer(UnitGraph graph) {
33
34     // Perform the analysis
35     PathConditionAnalysis pathConditionAnalysis1 = new PathConditionAnalysis(
36         graph, 1);
37     PathConditionAnalysis pathConditionAnalysis2 = new PathConditionAnalysis(
38         graph, 0);
39
40     // data structure to keep the var lists
41     conditions1 = new HashMap(graph.size() * 2 + 1, 0.7f);
42     conditions2 = new HashMap(graph.size() * 2 + 1, 0.7f);
43
44     Iterator unitIt = graph.iterator();
45
46     while (unitIt.hasNext()) {
47       Unit s = (Unit) unitIt.next();
48
49       FlowSet set0 = (FlowSet) pathConditionAnalysis1.getFlowBefore(s);
50       FlowSet set1 = (FlowSet) pathConditionAnalysis2.getFlowBefore(s);
51
52       conditions1.put(s, Collections.unmodifiableList(set0.toList()));
53       conditions2.put(s, Collections.unmodifiableList(set1.toList()));
54
55     }
56   }
```

Appendix B (Continued)

```
57
58    /**
59     * Returns the first path condition calculated with the analysis
60     **/
61    public List getFirstPathConditions(Unit s) {
62      return (List) conditions1.get(s);
63    }
64
65    /**
66     * Returns the second path condition calculated with the analysis
67     **/
68    public List getSecondPathConditions(Unit s) {
69      return (List) conditions2.get(s);
70    }
71
72  }
73
74  class PathConditionAnalysis extends ForwardBranchedFlowAnalysis {
75    private int PATHCONSTANT;
76    FlowSet emptySet = new ArraySparseSet();
77    MHGDominatorsFinder df = new MHGDominatorsFinder(graph);
78
79    PathConditionAnalysis(UnitGraph graph, int constant) {
80      super(graph);
81      this.PATHCONSTANT = constant;
82      doAnalysis();
```

Appendix B (Continued)

```
83    }
84
85    /**
86     * All INs are initialized to the empty set.
87     **/
88    protected Object newInitialFlow() {
89      return emptySet.clone();
90    }
91
92    /**
93     * IN(Start) is the empty set
94     **/
95    protected Object entryInitialFlow() {
96      return emptySet.clone();
97    }
98
99    /**
100    * The if conditions generate new path conditions.
101    **/
102   protected void flowThrough(Object inValue, Unit unit, List fallOut,
103       List branchOuts) {
104     Unit u = (Unit) unit;
105     FlowSet in = (FlowSet) inValue;
106     boolean nextIsLoopHeader = false;
107     // if we are in the head of a cycle don't propagate the cycle condition
108     // but the preceding ones
```

Appendix B (Continued)

```
109     List<Unit> preds = graph.getPredsOf(u);
110     for (Unit a : preds) {
111       for (Object b : df.getDominators(a))
112         if (b.equals(u)) {
113           // then we are in the head of a loop and we copy the written
114           // Vars from the previous unit
115           for (Unit c : preds)
116             if (!c.equals(a)) {
117               for (Object var : ((FlowSet) this.getFlowBefore(c))
118                   .toList())
119                 in.add((String) var);
120             }
121
122         }
123     }
124
125     List<Unit> post = graph.getSuccsOf(u);
126     for (Unit a : post) {
127       for (Object b : df.getDominators(u))
128         if (b.equals(a)) {
129           // then we are before a back edge on a loop head avoid the
130           // copy
131           nextIsLoopHeader = true;
132
133         }
134     }
```

Appendix B (Continued)

```
135     FlowSet outbranch = (FlowSet) in.clone();
136     FlowSet outfall = (FlowSet) in.clone();
137
138     if (u instanceof IfStmt) {
139       ConditionExpr condition = (ConditionExpr) ((IfStmt) u)
140           .getCondition();
141       ValueBox op1 = condition.getOp1Box();
142       ValueBox op2 = condition.getOp2Box();
143       String operator1 = new String();
144       String operator2 = new String();
145
146       operator1 = this.getSymbolicValue(u, op1);
147       operator2 = this.getSymbolicValue(u, op2);
148
149       outbranch.add(operator1 + condition.getSymbol().trim() + operator2);
150
151       outfall.add(operator1
152           + this.getNegation(condition.getSymbol().trim())
153           + operator2);
154     }
155
156     if (nextIsLoopHeader) {
157       outbranch.clear();
158       outfall.clear();
159     }
160     // now copy the computed info to all successors
```

Appendix B (Continued)

```
161     for (Iterator it = fallOut.iterator(); it.hasNext();) {
162       copy(outfall, it.next());
163     }
164     for (Iterator it = branchOuts.iterator(); it.hasNext();) {
165       copy(outbranch, it.next());
166     }
167     // (kill set is empty)
168
169   }
170
171   /**
172    * Returns the negation of a conditional expression as a string
173    **/
174   private String getNegation(String symbol) {
175     if (symbol.compareTo("<") == 0)
176       return ">=";
177     else if (symbol.compareTo(">") == 0)
178       return "<=";
179     else if (symbol.compareTo("<=") == 0)
180       return ">";
181     else if (symbol.compareTo(">=") == 0)
182       return "<";
183     else if (symbol.compareTo("==") == 0)
184       return "!=";
185     else if (symbol.compareTo("!=") == 0)
186       return "==";
```

Appendix B (Continued)

```
187     return null;
188   }
189
190   /**
191    * Recursive function that perform symbolic execution of a local variable
192    **/
193   private String getSymbolicValue(Unit u, ValueBox localVar) {
194     if (localVar.getValue() instanceof Local) {
195
196       DominatorsFinder df = new MHGDominatorsFinder(graph);
197       Unit unitFound = null;
198       for (Iterator domsIt = df.getDominators(u).iterator(); domsIt
199           .hasNext();) {
200
201         Unit dom = (Unit) domsIt.next();
202
203         Iterator arrit = dom.getDefBoxes().iterator();
204         while (arrit.hasNext()) {
205           ValueBox arr = (ValueBox) arrit.next();
206           if (arr.getValue().equals(localVar.getValue())) {
207             unitFound = dom;
208           }
209         }
210       }
211       String s1 = "";
212       if (unitFound instanceof AssignStmt) {
```

Appendix B (Continued)

```
213         Value rightOp = ((AssignStmt) unitFound).getRightOp();
214         ValueBox rightOpBox = ((AssignStmt) unitFound).getRightOpBox();
215         if (rightOp instanceof InvokeExpr) {
216           if (rightOp instanceof VirtualInvokeExpr)
217             s1 += getSymbolicValue(unitFound,
218                 ((VirtualInvokeExpr) rightOp).getBaseBox())
219                 + ".";
220           s1 += ((InvokeExpr) rightOp).getMethod().getName() + "(";
221           // for each parameter get the symbolic value of it
222           int count = ((InvokeExpr) rightOp).getArgCount();
223           for (int i = 0; i < count; i++) {
224             s1 += getSymbolicValue(unitFound,
225                 ((InvokeExpr) rightOp).getArgBox(i));
226             if (i != count - 1)
227               s1 += ",";
228           }
229           s1 += ")";
230         } else if (rightOp instanceof ArrayRef) {
231           s1 += getSymbolicValue(unitFound,
232               ((ArrayRef) rightOp).getBaseBox());
233           s1 += "[";
234           s1 += getSymbolicValue(unitFound,
235               ((ArrayRef) rightOp).getIndexBox());
236           s1 += "]";
237         } else if (rightOp instanceof BinopExpr) {
238           s1 += getSymbolicValue(unitFound,
```

Appendix B (Continued)

```
239                 ((BinopExpr) rightOp).getOp1Box());
240             s1 += ((BinopExpr) rightOp).getSymbol();
241             s1 += getSymbolicValue(unitFound,
242                 ((BinopExpr) rightOp).getOp2Box());
243           } else if (rightOp instanceof Local) {
244             s1 += getSymbolicValue(unitFound, rightOpBox);
245           } else {
246             s1 += rightOp.toString();
247           }
248
249         } else if (unitFound instanceof IdentityStmt) {
250           Value leftOp = ((IdentityStmt) unitFound).getLeftOp();
251           s1 += leftOp.toString();
252         }
253
254         return s1;
255       } else {
256         return localVar.getValue().toString();
257       }
258
259   }
260
261   /**
262    * Personalized Merge Function. Propagate one of the two condition path that
263    * enter the merge node
264    **/
```

Appendix B (Continued)

```
265   protected void merge(Object in1, Object in2, Object out) {
266     FlowSet inSet1 = (FlowSet) in1, inSet2 = (FlowSet) in2, outSet = (FlowSet)
          out;
267     if (inSet1.isEmpty())
268       inSet2.copy(outSet);
269     else if (inSet2.isEmpty())
270       inSet1.copy(outSet);
271     else {
272       if (PATHCONSTANT == 1)
273         inSet1.copy(outSet);
274       else
275         inSet2.copy(outSet);
276     }
277   }
278
279   protected void copy(Object source, Object dest) {
280     FlowSet sourceSet = (FlowSet) source, destSet = (FlowSet) dest;
281
282     sourceSet.copy(destSet);
283   }
284
285 }
```

B.3 Data Structures

These classes represent the data structure used to store the analyses information:

```
1 import java.util.ArrayList;
```

Appendix B (Continued)

```
2  import java.util.List;
3
4  public class Rule {
5
6    private String ruleMethodName;
7    private String kind;
8    private List<Property> writtenVariables;
9    private List<Property> readVariables;
10   private List<Property> mayWrittenVariables;
11   private List<Property> mayReadVariables;
12
13   public Rule(String ruleMethodName) {
14     this.ruleMethodName = ruleMethodName;
15     this.writtenVariables = new ArrayList<Property>();
16     this.readVariables = new ArrayList<Property>();
17     this.mayWrittenVariables = new ArrayList<Property>();
18     this.mayReadVariables = new ArrayList<Property>();
19   }
20
21   public void addWrittenVariable(String var) {
22     Property p = new Property(var);
23     this.writtenVariables.add(p);
24   }
25
26   public void addReadVariable(String var) {
27     Property p = new Property(var);
```

Appendix B (Continued)

```
28      this.readVariables.add(p);
29    }
30
31    public void addMayWrittenVariable(String var,
32        List<String> firstPathConditions, List<String> secondPathConditions) {
33      Property p = new Property(var);
34      p.setFirstPathConditions(firstPathConditions);
35      p.setSecondPathConditions(secondPathConditions);
36      this.mayWrittenVariables.add(p);
37    }
38
39    public void addMayReadVariable(String var,
40        List<String> firstPathConditions, List<String> secondPathConditions) {
41      Property p = new Property(var);
42      p.setFirstPathConditions(firstPathConditions);
43      p.setSecondPathConditions(secondPathConditions);
44      this.mayReadVariables.add(p);
45    }
46
47    public void setRuleMethodName(String ruleMethodName) {
48      this.ruleMethodName = ruleMethodName;
49    }
50
51    public String getRuleMethodName() {
52      return ruleMethodName;
53    }
```

Appendix B (Continued)

```
54
55    public List<Property> getWrittenVariables() {
56      return writtenVariables;
57    }
58
59    public List<Property> getReadVariables() {
60      return readVariables;
61    }
62
63    public List<Property> getMayWrittenVariables() {
64      return mayWrittenVariables;
65    }
66
67    public List<Property> getMayReadVariables() {
68      return mayReadVariables;
69    }
70
71    public boolean containsWrittenVariable(String variableName) {
72      for (Property p : this.getWrittenVariables()) {
73        if (p.getVariableName().compareTo(variableName) == 0)
74          return true;
75      }
76      return false;
77
78    }
79
```

Appendix B (Continued)

```
80    public boolean containsReadVariable(String variableName) {
81      for (Property p : this.getReadVariables()) {
82        if (p.getVariableName().compareTo(variableName) == 0)
83          return true;
84      }
85      return false;
86
87    }
88
89    public boolean containsMayWrittenVariable(String variableName) {
90      for (Property p : this.getMayWrittenVariables()) {
91        if (p.getVariableName().compareTo(variableName) == 0)
92          return true;
93      }
94      return false;
95
96    }
97
98    public boolean containsMayReadVariable(String variableName) {
99      for (Property p : this.getMayReadVariables()) {
100       if (p.getVariableName().compareTo(variableName) == 0)
101         return true;
102     }
103     return false;
104   }
105
```

Appendix B (Continued)

```
106   public String toXMLString() {
107     String xmlString = "<rule>\n";
108
109     xmlString += "<ruleMethodName>" + this.ruleMethodName
110         + "</ruleMethodName>\n";
111
112     xmlString += "<write>\n";
113     for (Property s : this.writtenVariables)
114       xmlString += "<property>" + s.getVariableName() + "</property>\n";
115     xmlString += "</write>\n";
116
117     xmlString += "<read>\n";
118     for (Property s : this.readVariables)
119       xmlString += "<property>" + s.getVariableName() + "</property>\n";
120     xmlString += "</read>\n";
121
122     xmlString += "<mayWrite>\n";
123     for (Property s : this.mayWrittenVariables) {
124       xmlString += "<property>\n<name>" + s.getVariableName()
125           + "</name>\n<path1>";
126       for (String s1 : s.getFirstPathConditions())
127         xmlString += "<condition>" + this.formatXml(s1)
128             + "</condition>";
129       xmlString += "</path1>\n" + "<path2>";
130       for (String s1 : s.getSecondPathConditions())
131         xmlString += "<condition>" + this.formatXml(s1)
```

Appendix B (Continued)

```
132             + "</condition>";
133         xmlString += "</path2>\n</property>\n";
134       }
135       xmlString += "</mayWrite>\n";
136
137       xmlString += "<mayRead>\n";
138       for (Property s : this.mayReadVariables) {
139         xmlString += "<property>\n<name>" + s.getVariableName()
140             + "</name>\n<path1>";
141         for (String s1 : s.getFirstPathConditions())
142           xmlString += "<condition>" + this.formatXml(s1)
143               + "</condition>";
144         xmlString += "</path1>\n" + "<path2>";
145         for (String s1 : s.getSecondPathConditions())
146           xmlString += "<condition>" + this.formatXml(s1)
147               + "</condition>";
148         xmlString += "</path2>\n</property>\n";
149       }
150       xmlString += "</mayRead>\n";
151
152       xmlString += "</rule>\n";
153
154       return xmlString;
155     }
156
157     public void setKind(String kind) {
```

Appendix B (Continued)

```
158       this.kind = kind;
159     }
160
161     public String getKind() {
162       return kind;
163     }
164
165     private String formatXml(String s) {
166       String xml = "";
167       xml = s.replace("<", "&lt;");
168       xml = xml.replace(">", "&gt;");
169       xml = xml.replace("\"", "&quot;");
170       return xml;
171     }
172
173 }
```

```
 1 import java.util.ArrayList;
 2 import java.util.List;
 3
 4 public class Property {
 5   private String variableName;
 6   private String fullPathName;
 7   private List<String> firstPathConditions;
 8   private List<String> secondPathConditions;
 9
```

Appendix B (Continued)

```
10    public Property(String variableName) {
11      this.setVariableName(variableName);
12      this.setFullPathName(new String());
13      this.setFirstPathConditions(new ArrayList<String>());
14      this.setSecondPathConditions(new ArrayList<String>());
15    }
16
17    public void setVariableName(String variableName) {
18      this.variableName = variableName;
19    }
20
21    public String getVariableName() {
22      return variableName;
23    }
24
25    public void setFullPathName(String fullPathName) {
26      this.fullPathName = fullPathName;
27    }
28
29    public String getFullPathName() {
30      return fullPathName;
31    }
32
33    public void setFirstPathConditions(List<String> pathConditions) {
34      this.firstPathConditions = pathConditions;
35    }
```

Appendix B (Continued)

```
36
37    public List<String> getFirstPathConditions() {
38      return firstPathConditions;
39    }
40
41    public void setSecondPathConditions(List<String> secondPathConditions) {
42      this.secondPathConditions = secondPathConditions;
43    }
44
45    public List<String> getSecondPathConditions() {
46      return secondPathConditions;
47    }
48
49  }
```

B.4 Result Collection

This is the code that uses the previous analyses and collect their results creating a data structure to store the information:

```
1  import soot.PackManager;
2  import soot.Scene;
3  import soot.SootClass;
4  import soot.Transform;
5
6  public class MyMain {
7    public static void main(String[] args) {
```

Appendix B (Continued)

```
8       if (args.length == 0) {
9         System.out.println("No args!");
10        System.exit(0);
11      }
12
13      PackManager.v().getPack("jtp")
14          .add(new Transform("jtp.analyzer", RuleAnalyzer.v()));
15
16      // Just in case, resolve the PrintStream and System SootClasses.
17      Scene.v().addBasicClass("java.io.PrintStream", SootClass.SIGNATURES);
18      Scene.v().addBasicClass("java.lang.System", SootClass.SIGNATURES);
19      soot.Main.main(args);
20    }
21  }
```

```
1  import java.io.BufferedWriter;
2  import java.io.FileWriter;
3  import java.util.ArrayList;
4  import java.util.Iterator;
5  import java.util.List;
6  import java.util.Map;
7
8  import soot.Body;
9  import soot.BodyTransformer;
10 import soot.Unit;
11 import soot.jimple.Stmt;
```

Appendix B (Continued)

```
12 import soot.toolkits.graph.CompleteUnitGraph;
13 import soot.toolkits.graph.ExceptionalUnitGraph;
14 import soot.toolkits.graph.UnitGraph;
15 import soot.util.Chain;
16
17 public class RuleAnalyzer extends BodyTransformer {
18   private String xmlString = "";
19
20   private static RuleAnalyzer instance = new RuleAnalyzer();
21
22   private RuleAnalyzer() {
23   }
24
25   public static RuleAnalyzer v() {
26     return instance;
27   }
28
29   private void printToFile(String s) {
30     try {
31       // I use the absolute path because I run it with soot main
32       FileWriter fstream = new FileWriter(
33           "/Users/Andrea/Desktop/file/AnalysisResult.xml");
34       BufferedWriter out = new BufferedWriter(fstream);
35       out.write(s);
36       out.close();
37     } catch (Exception e) {
```

Appendix B (Continued)

```
38          e.printStackTrace();
39        }
40    }
41
42    @Override
43    protected void internalTransform(Body body, String phaseName, Map options) {
44
45      List<String> writtenVars = new ArrayList<String>();
46      List<String> readVars = new ArrayList<String>();
47      List<String> mayWrittenVars = new ArrayList<String>();
48      List<String> mayReadVars = new ArrayList<String>();
49
50      List<String> writtenProp = new ArrayList<String>();
51      List<String> readProp = new ArrayList<String>();
52      List<String> mayReadProp = new ArrayList<String>();
53      List<String> mayWrittenProp = new ArrayList<String>();
54
55      Rule rule = new Rule(body.getMethod().getName());
56      List<String> parameters = this.getMethodParameterList(body);
57
58      UnitGraph graph1 = new CompleteUnitGraph(body);
59      UnitGraph graph2 = new ExceptionalUnitGraph(body);
60      PathConditionAnalyzer p = new PathConditionAnalyzer(graph1);
61      ReadWriteAnalyzer rw = new ReadWriteAnalyzer(graph2, parameters);
62
63      Chain units = body.getUnits();
```

Appendix B (Continued)

```
64
65      Iterator stmtIt = units.snapshotIterator();
66
67      // written variables
68      writtenVars = rw.getWParameters((Unit) units.getLast());
69      Iterator it = writtenVars.iterator();
70      while (it.hasNext()) {
71        String u = (String) it.next();
72        rule.addWrittenVariable(u);
73      }
74
75      // read variables
76      readVars = rw.getRParameters((Unit) units.getLast());
77      it = readVars.iterator();
78      while (it.hasNext()) {
79        String u = (String) it.next();
80        rule.addReadVariable(u);
81      }
82
83      // mayRead and mayWrite
84      while (stmtIt.hasNext()) {
85        Stmt s = (Stmt) stmtIt.next();
86        List<String> condition1 = new ArrayList<String>();
87        List<String> condition2 = new ArrayList<String>();
88        mayWrittenVars = rw.getMWParameters(s);
89        it = mayWrittenVars.iterator();
```

Appendix B (Continued)

```
90        while (it.hasNext()) {
91          String u = (String) it.next();
92          if (!rule.containsWrittenVariable(u)
93              && !rule.containsMayWrittenVariable(u)) {
94
95            // add Condition to the property
96            condition1 = p.getFirstPathConditions(s);
97            condition2 = p.getFirstPathConditions(s);
98            if (condition1.size() == 0 || condition2.size() == 0)
99              throw new NullPointerException("Condizioni vuote "
100                 + rule.getRuleMethodName() + " " + s.toString()
101                 + " " + u);
102           rule.addMayWrittenVariable(u, condition1, condition2);
103         }
104       }
105
106       mayReadVars = rw.getMRParameters(s);
107       it = mayReadVars.iterator();
108       while (it.hasNext()) {
109         String u = (String) it.next();
110         if (!rule.containsReadVariable(u)
111             && !rule.containsMayReadVariable(u)) {
112
113           // add Condition to the property
114           condition1 = p.getFirstPathConditions(s);
115           condition2 = p.getFirstPathConditions(s);
```

Appendix B (Continued)

```
116             if (condition1.size() == 0 || condition2.size() == 0)
117               throw new NullPointerException("Condizioni vuote "
118                   + rule.getRuleMethodName() + " " + s.toString());
119             rule.addMayReadVariable(u, condition1, condition2);
120           }
121         }
122
123       }
124
125       /*
126        * // mayWrittenVars = rw.getMWParameters((Unit) units.getLast()); it =
127        * mayWrittenVars.iterator(); while (it.hasNext()) { String u = (String)
128        * it.next(); if (!ruleRWProperties.getWrittenVariables().contains(u))
129        * ruleRWProperties.addMayWrittenVariable(u); }
130        *
131        * // mayReadVars = rw.getMRParameters((Unit) units.getLast()); it =
132        * mayReadVars.iterator(); while (it.hasNext()) { String u = (String)
133        * it.next(); if (!ruleRWProperties.getReadVariables().contains(u))
134        * ruleRWProperties.addMayReadVariable(u); }
135        */
136
137       // init is the last rule of a Jimple class by default
138       if (rule.getRuleMethodName().compareTo("<init>") != 0)
139         xmlString += rule.toXMLString();
140
141       // format and print the xmlFile
```

Appendix B (Continued)

```
142       else {
143         String xmlFileString = "<?xml version=\"1.0\"?>\n";
144         xmlFileString += "<root>\n" + xmlString + "</root>\n";
145         printToFile(xmlFileString);
146
147       }
148     }
149
150     /**
151      * Returns the methods parameter names as a list
152      **/
153     private List<String> getMethodParameterList(Body body) {
154       List<String> list = new ArrayList<String>();
155       int paramNum = body.getMethod().getParameterCount();
156       for (int i = 0; i < paramNum; i++) {
157         list.add(body.getParameterLocal(i).getName());
158       }
159
160       return list;
161     }
162
163 }
```

Appendix C

CONFLICT IDENTIFICATION SOURCE CODE

This is the source code that deals with the conflict identification within the system:

```
1  package conflictIdentificator;
2
3  import java.io.BufferedWriter;
4  import java.io.FileWriter;
5  import java.io.IOException;
6  import java.util.ArrayList;
7  import java.util.List;
8
9  import javax.xml.parsers.DocumentBuilder;
10 import javax.xml.parsers.DocumentBuilderFactory;
11 import javax.xml.parsers.ParserConfigurationException;
12 import javax.xml.xpath.XPath;
13 import javax.xml.xpath.XPathConstants;
14 import javax.xml.xpath.XPathExpression;
15 import javax.xml.xpath.XPathExpressionException;
16 import javax.xml.xpath.XPathFactory;
17
18 import org.w3c.dom.Document;
19 import org.w3c.dom.Element;
20 import org.w3c.dom.Node;
```

Appendix C (Continued)

```
21 import org.w3c.dom.NodeList;
22 import org.xml.sax.SAXException;
23
24 public class XmlQueryUtility {
25
26   private Document ruleList;
27   private Document readWriteProperties;
28   Element ruleListRoot;
29   Element readWritePropertiesRoot;
30   private List<Rule> rules;
31   private List<String> parents;
32   private List<Rule> emptyRules;
33
34   public XmlQueryUtility() {
35     DocumentBuilderFactory domFactory = DocumentBuilderFactory
36         .newInstance();
37     domFactory.setNamespaceAware(true);
38     DocumentBuilder builder;
39     this.rules = new ArrayList<Rule>();
40     this.emptyRules = new ArrayList<Rule>();
41     this.parents = new ArrayList<String>();
42
43     try {
44       builder = domFactory.newDocumentBuilder();
45       ruleList = builder.parse("RuleList.xml");
46       readWriteProperties = builder.parse("AnalysisResult.xml");
```

Appendix C (Continued)

```
47
48        ruleListRoot = ruleList.getDocumentElement();
49        readWritePropertiesRoot = readWriteProperties.getDocumentElement();
50        this.parseRules();
51
52        // print complete analysis result to a file
53        this.printToFile(this.getAnalysisResults());
54
55      } catch (ParserConfigurationException e) {
56        // TODO Auto-generated catch block
57        e.printStackTrace();
58      } catch (SAXException e) {
59        // TODO Auto-generated catch block
60        e.printStackTrace();
61      } catch (IOException e) {
62        // TODO Auto-generated catch block
63        e.printStackTrace();
64      }
65    }
66
67    public void parseRules() {
68      boolean found = false;
69
70      NodeList nodes = ruleListRoot.getElementsByTagName("rule");
71      for (int j = 0; j < nodes.getLength(); j++) {
72        Element rule = ((Element) nodes.item(j));
```

Appendix C (Continued)

```
73        String methodName = rule.getElementsByTagName("methodName").item(0)
74            .getTextContent().trim();
75        String parent = rule.getElementsByTagName("parent").item(0)
76            .getAttributes().item(1).getNodeValue();
77        String kind = rule.getElementsByTagName("kind").item(0)
78            .getTextContent();
79
80        Rule r = new Rule(methodName, kind, parent);
81
82        NodeList nodes2 = readWritePropertiesRoot
83            .getElementsByTagName("rule");
84        found = false;
85
86        for (int i = 0; i < nodes2.getLength(); i++) {
87          // necessary for the mapping method
88          NodeList propfullNames = rule.getElementsByTagName("property");
89          NodeList variables = rule.getElementsByTagName("variable");
90
91          Element rule2 = ((Element) nodes2.item(i));
92          String methodName2 = rule2
93              .getElementsByTagName("ruleMethodName").item(0)
94              .getTextContent().trim();
95          if (methodName.compareTo(methodName2) == 0) {
96            found = true;
97            NodeList properties = ((Element) rule2
98                .getElementsByTagName("write").item(0))
```

Appendix C (Continued)

```
99              .getElementsByTagName("property");
100           for (int k = 0; k < properties.getLength(); k++) {
101
102             r.addWrittenVariable(
103                 properties.item(k).getTextContent().trim(),
104                 getFullNameOf(properties.item(k)
105                     .getTextContent().trim(), parent,
106                     propfullNames, variables, kind));
107           }
108           NodeList properties1 = ((Element) rule2
109               .getElementsByTagName("read").item(0))
110               .getElementsByTagName("property");
111           for (int k = 0; k < properties1.getLength(); k++) {
112
113             r.addReadVariable(
114                 properties1.item(k).getTextContent().trim(),
115                 getFullNameOf(properties1.item(k)
116                     .getTextContent().trim(), parent,
117                     propfullNames, variables, kind));
118           }
119           NodeList properties2 = ((Element) rule2
120               .getElementsByTagName("mayWrite").item(0))
121               .getElementsByTagName("property");
122           for (int k = 0; k < properties2.getLength(); k++) {
123             Element prop = (Element) properties2.item(k);
124             NodeList conditions1 = ((Element) prop
```

Appendix C (Continued)

```
125                 .getElementsByTagName("path1").item(0))
126                 .getElementsByTagName("condition");
127             NodeList conditions2 = ((Element) prop
128                 .getElementsByTagName("path2").item(0))
129                 .getElementsByTagName("condition");
130
131             List<String> pathConditionsList1 = new ArrayList<String>();
132             List<String> pathConditionsList2 = new ArrayList<String>();
133
134             Property p = new Property(prop
135                 .getElementsByTagName("name").item(0)
136                 .getTextContent().trim(), getFullNameOf(prop
137                 .getElementsByTagName("name").item(0)
138                 .getTextContent().trim(), parent,
139                 propfullNames, variables, kind));
140             for (int t = 0; t < conditions1.getLength(); t++) {
141               pathConditionsList1.add(conditions1.item(t)
142                     .getTextContent().trim());
143             }
144             for (int t = 0; t < conditions2.getLength(); t++) {
145               pathConditionsList2.add(conditions2.item(t)
146                     .getTextContent().trim());
147             }
148             p.setFirstPathConditions(pathConditionsList1);
149             p.setSecondPathConditions(pathConditionsList2);
150
```

Appendix C (Continued)

```
151             r.addMayWrittenVariable(p);
152           }
153           NodeList properties3 = ((Element) rule2
154               .getElementsByTagName("mayRead").item(0))
155               .getElementsByTagName("property");
156
157           for (int k = 0; k < properties3.getLength(); k++) {
158             Element prop = (Element) properties3.item(k);
159             NodeList conditions1 = ((Element) prop
160                 .getElementsByTagName("path1").item(0))
161                 .getElementsByTagName("condition");
162             NodeList conditions2 = ((Element) prop
163                 .getElementsByTagName("path2").item(0))
164                 .getElementsByTagName("condition");
165
166             List<String> pathConditionsList1 = new ArrayList<String>();
167             List<String> pathConditionsList2 = new ArrayList<String>();
168
169             Property p = new Property(prop
170                 .getElementsByTagName("name").item(0)
171                 .getTextContent().trim(), getFullNameOf(prop
172                 .getElementsByTagName("name").item(0)
173                 .getTextContent().trim(), parent,
174                 propfullNames, variables, kind));
175             for (int t = 0; t < conditions1.getLength(); t++) {
176               pathConditionsList1.add(conditions1.item(t)
```

Appendix C (Continued)

```
177                     .getTextContent().trim());
178               }
179               for (int t = 0; t < conditions2.getLength(); t++) {
180                 pathConditionsList2.add(conditions2.item(t)
181                     .getTextContent().trim());
182               }
183               p.setFirstPathConditions(pathConditionsList1);
184               p.setSecondPathConditions(pathConditionsList2);
185
186               r.addMayReadVariable(p);
187             }
188           }
189         }
190         if (found) {
191           rules.add(r);
192           if (!parents.contains(parent))
193             parents.add(parent);
194         } else {
195           emptyRules.add(r);
196         }
197       }
198
199   }
200
201   private String getFullNameOf(String variable, String parentName,
202       NodeList props, NodeList vars, String kind) {
```

Appendix C (Continued)

```
203     int limit;
204     boolean parentMap = false;
205     int varLength = vars.getLength();
206     int propLength = props.getLength();
207     if (varLength <= propLength)
208       limit = varLength;
209     else {
210       limit = propLength;
211       parentMap = true;
212     }
213     for (int i = 0; i < limit; i++) {
214       if (variable.compareTo(vars.item(i).getTextContent().trim()) == 0)
215         return props.item(i).getTextContent().trim();
216     }
217     if (parentMap && kind.compareTo("IS_VALID_KIND") != 0
218         && kind.compareTo("IS_REQUIRED_KIND") != 0)
219       return parentName;
220     return variable;
221   }
222
223   public List<PossibleConflict> getRConflicts() {
224
225     List<PossibleConflict> possibleConflicts = new ArrayList<PossibleConflict>();
226     List<String> parents = this.parents;
227     List<Rule> rules = this.rules;
228     List<String> kinds = getReadTriggeredKinds();
```

Appendix C (Continued)

```
229
230     for (String parent : parents) {
231
232       PossibleConflict conflict = new PossibleConflict(parent);
233
234       for (Rule r : rules) {
235         // if it is a read Triggered Rule
236         if (kinds.contains(r.getKind())) {
237           // if it is attached to parent
238           if (parent.compareTo(r.getParent().getFullPathName()) == 0) {
239             // initialize the rule properties
240             conflict.addRule(r);
241           }
242         }
243       }
244       if (conflict.containsAtLeastTwoRules()) {
245         possibleConflicts.add(conflict);
246       }
247
248     }
249     /*
250      * for (PossibleConflict c : possibleConflicts) { for (Rule r :
251      * c.getRules()) { System.out.println(r.getRuleMethodName());
252      * System.out.println("          " + r.getWrittenVariables());
253      * System.out.println("          " + r.getReadVariables());
254      * System.out.println("          " + r.getMayWrittenVariables());
```

Appendix C (Continued)

```
255       * System.out.println("          " + r.getMayReadVariables());
256       * System.out.println(); } }
257       */
258     return possibleConflicts;
259   }
260
261   public List<PossibleConflict> getWConflicts() {
262
263     List<PossibleConflict> possibleConflicts = new ArrayList<PossibleConflict>();
264     List<String> parents = this.parents;
265     List<Rule> rules = this.rules;
266     List<String> kinds = getWriteTriggeredKinds();
267
268     for (String parent : parents) {
269
270       PossibleConflict conflict = new PossibleConflict(parent);
271
272       for (Rule r : rules) {
273         // if it is a read Triggered Rule
274         if (kinds.contains(r.getKind())) {
275           // if it is attached to parent
276           if (parent.compareTo(r.getParent().getFullPathName()) == 0) {
277             // initialize the rule properties
278             conflict.addRule(r);
279           }
280         }
```

Appendix C (Continued)

```
281         }
282         if (conflict.containsAtLeastTwoRules()) {
283           possibleConflicts.add(conflict);
284         }
285
286       }
287       /*
288        * for (PossibleConflict c : possibleConflicts) { for (Rule r :
289        * c.getRules()) { System.out.println(r.getRuleMethodName());
290        * System.out.println("          " + r.getWrittenVariables());
291        * System.out.println("          " + r.getReadVariables());
292        * System.out.println("          " + r.getMayWrittenVariables());
293        * System.out.println("          " + r.getMayReadVariables());
294        * System.out.println(); } }
295        */
296       return possibleConflicts;
297     }
298
299     private List<String> getReadTriggeredKinds() {
300       List<String> kinds = new ArrayList<String>();
301       kinds.add("IS_VALID_KIND");
302       kinds.add("SUGGESTED_VALUE_KIND");
303       kinds.add("IS_REQUIRED_KIND");
304       kinds.add("LOWER_LIMIT_KIND");
305       kinds.add("UPPER_LIMIT_KIND");
306       kinds.add("ALLOWED_VALUE_KIND");
```

Appendix C (Continued)

```
307     return kinds;
308   }
309
310   private List<String> getWriteTriggeredKinds() {
311     List<String> kinds = new ArrayList<String>();
312     kinds.add("POST_SET_KIND");
313     kinds.add("POST_ADD_KIND");
314     kinds.add("LOWER_LIMIT_KIND");
315     kinds.add("UPPER_LIMIT_KIND");
316     kinds.add("ALLOWED_VALUE_KIND");
317     return kinds;
318   }
319
320   public Node getNodeRule(String ruleMethodName)
321       throws ParserConfigurationException, SAXException, IOException,
322       XPathExpressionException {
323
324     List<String> resultString = new ArrayList<String>();
325     XPathFactory factory = XPathFactory.newInstance();
326     XPath xpath = factory.newXPath();
327     XPathExpression expr = xpath.compile("//rule[ruleMethodName='"
328         + ruleMethodName + "']");
329
330     Object result = expr.evaluate(readWriteProperties, XPathConstants.NODE);
331     Node node = (Node) result;
332
```

Appendix C (Continued)

```
333     return node;
334
335   }
336
337   public void getDerivedValueConflicts() {
338
339     for (Rule r : rules) {
340       String parent = r.getParent().getFullPathName();
341       String kind = r.getKind();
342       if (kind.compareTo("DERIVED_VALUE_KIND") == 0) {
343         for (Rule r1 : rules) {
344           if (!r.equals(r1)
345               && r1.getWrittenVariables().contains(parent)) {
346
347             System.err.println(r.getRuleMethodName());
348             System.err.println(r1.getRuleMethodName());
349             // throw new IllegalAccessException();
350           } else if (!r.equals(r1)
351               && r1.getMayWrittenVariables().contains(parent)) {
352             System.err.println(r.getRuleMethodName());
353             System.err.println(r1.getRuleMethodName());
354             // throw new IllegalAccessException();
355           }
356         }
357       }
358     }
```

Appendix C (Continued)

```
359     for (Rule r : emptyRules) {
360       String parent = r.getParent().getFullPathName();
361       String kind = r.getKind();
362       if (kind.compareTo("DERIVED_VALUE_KIND") == 0) {
363         for (Rule r1 : rules) {
364           if (!r.equals(r1)
365               && r1.getWrittenVariables().contains(parent)) {
366             System.err.println(r.getRuleMethodName());
367             System.err.println(r1.getRuleMethodName());
368             // throw new IllegalAccessException();
369           } else if (!r.equals(r1)
370               && r1.getMayWrittenVariables().contains(parent)) {
371             System.err.println(r.getRuleMethodName());
372             System.err.println(r1.getRuleMethodName());
373             // throw new IllegalAccessException();
374           }
375         }
376       }
377     }
378
379   }
380
381   private void printToFile(String s) {
382     try {
383       FileWriter fstream = new FileWriter("RefinedAnalysisResult.xml");
384       BufferedWriter out = new BufferedWriter(fstream);
```

Appendix C (Continued)

```
385         out.write(s);
386         out.close();
387       } catch (Exception e) {
388         e.printStackTrace();
389       }
390     }
391
392     public String getAnalysisResults() {
393       String result = "<analysis>\n";
394       for (Rule r : rules) {
395         result += r.toXMLString();
396       }
397       result += "</analysis>\n";
398       return result;
399     }
400
401     List<Rule> getRules() {
402       return this.rules;
403     }
404 }
```

```
1 package conflictIdentificator;
2
3 import java.io.BufferedWriter;
4 import java.io.FileWriter;
5 import java.util.ArrayList;
```

Appendix C (Continued)

```
6  import java.util.List;
7
8  public class ConflictIdentificator {
9
10   private List<ConflictPair> conflicts;
11   private XmlQueryUtility xmlQuery;
12   private List<Rule> ruleList;
13
14   public ConflictIdentificator() {
15     conflicts = new ArrayList<ConflictPair>();
16     xmlQuery = new XmlQueryUtility();
17     this.ruleList = xmlQuery.getRules();
18     IdentifyConflicts();
19     System.out.println("\nCONFLICTS FOUND :" + conflicts.size());
20
21     String s = "";
22     for (ConflictPair c : conflicts) {
23       /*
24        * System.out.println("Conflict Pair");
25        * System.out.println("   On property: " + c.getProperty());
26        * System.out.println("   Rule1 : " + c.getActionRule1());
27        * System.out.println("   Rule2 : " + c.getActionRule2());
28        */
29       System.out.println(c.toXmlString());
30       System.out.println();
31       s += c.toXmlString();
```

Appendix C (Continued)

```
32      }
33      printToFile("<conflicts>\n" + s + "</conflicts>\n");
34    }
35
36    private void IdentifyConflicts() {
37      List<PossibleConflict> possibleReadTriggeredConflicts = null;
38      List<PossibleConflict> possibleWriteTriggeredConflicts = null;
39      int count = 0;
40
41      xmlQuery.getDerivedValueConflicts();
42
43      System.out.println("Identifying concurrent triggered rules...");
44      possibleReadTriggeredConflicts = xmlQuery.getRConflicts();
45      System.out.println("    " + possibleReadTriggeredConflicts.size()
46          + " groups of concurrent read truggered rules identified");
47      possibleWriteTriggeredConflicts = xmlQuery.getWConflicts();
48      System.out.println("    " + possibleWriteTriggeredConflicts.size()
49          + " groups of concurrent write triggered rules identified");
50
51      // concurrency Conflicts on readTriggered Rules
52      if (possibleReadTriggeredConflicts != null) {
53        System.out
54            .println("Initializing  read triggered conflict identification...");
55        for (PossibleConflict c : possibleReadTriggeredConflicts) {
56          List<Rule> rules = c.getRules();
57          for (int i = 0; i < rules.size(); i++) {
```

Appendix C (Continued)

```
58          Rule r = rules.get(i);
59          for (int j = i + 1; j < rules.size(); j++) {
60            Rule r1 = rules.get(j);
61            // check if rule r writes a property that is read by r1
62            List<Property> readVars = r.getReadVariables();
63            List<Property> readVars1 = r1.getReadVariables();
64            List<Property> writtenVars1 = r1.getWrittenVariables();
65            List<Property> mayReadVars = r.getMayReadVariables();
66            List<Property> mayReadVars1 = r1.getMayReadVariables();
67            List<Property> mayWrittenVars1 = r1
68                .getMayWrittenVariables();
69            List<Property> mayWrittenVars = r
70                .getMayWrittenVariables();
71
72            /*
73             * System.out.println("Analyzing " +
74             * r.getRuleMethodName() + " , " +
75             * r1.getRuleMethodName()); System.out
76             * .println("    r reads: " + readVars.toString());
77             * System.out.println("    r writes: " +
78             * r.getWrittenVariables());
79             * System.out.println("    r1 reads: " +
80             * readVars1.toString());
81             * System.out.println("    r1 writes: " +
82             * writtenVars1.toString());
83             * System.out.println("    r mayReads: " +
```

Appendix C (Continued)

```
84              * mayReadVars.toString());
85              * System.out.println("    r mayWrites: " +
86              * mayWrittenVars.toString());
87              * System.out.println("    r1 mayReads: " +
88              * mayReadVars1.toString());
89              * System.out.println("    r1 mayWrites: " +
90              * mayWrittenVars1.toString()); System.out.println();
91              */
92
93            for (Property p : r.getWrittenVariables()) {
94              for (Property p1 : readVars1)
95                if (p.getFullPathName().compareTo(
96                    p1.getFullPathName()) == 0) {
97                  ConflictPair conflict = new ConflictPair(r,
98                      r1);
99                  conflict.setActionRule1("write");
100                 conflict.setActionRule2("read");
101                 System.out.println(p.getVariableName()
102                     + " - " + p.getFullPathName());
103                 conflict.setProperty1(p);
104                 conflict.setProperty2(p1);
105                 conflicts.add(conflict);
106
107               }
108             for (Property p1 : writtenVars1)
109               if (p.getFullPathName().compareTo(
```

Appendix C (Continued)

```
110                     p1.getFullPathName()) == 0) {
111                 ConflictPair conflict1 = new ConflictPair(
112                     r, r1);
113                 conflict1.setActionRule1("write");
114                 conflict1.setActionRule2("write");
115                 conflict1.setProperty1(p);
116                 conflict1.setProperty2(p1);
117                 conflicts.add(conflict1);
118
119               }
120             for (Property p1 : mayReadVars1)
121               if (p.getFullPathName().compareTo(
122                   p1.getFullPathName()) == 0) {
123                 ConflictPair conflict1 = new ConflictPair(
124                     r, r1);
125                 conflict1.setActionRule1("write");
126                 conflict1.setActionRule2("mayRead");
127                 conflict1.setProperty1(p);
128                 conflict1.setProperty2(p1);
129                 conflicts.add(conflict1);
130               }
131             for (Property p1 : mayWrittenVars1)
132               if (p.getFullPathName().compareTo(
133                   p1.getFullPathName()) == 0) {
134                 ConflictPair conflict1 = new ConflictPair(
135                     r, r1);
```

Appendix C (Continued)

```
136                 conflict1.setActionRule1("write");
137                 conflict1.setActionRule2("mayWrite");
138                 conflict1.setProperty1(p);
139                 conflict1.setProperty2(p1);
140                 conflicts.add(conflict1);
141               }
142             }
143             for (Property p : writtenVars1) {
144               for (Property p1 : readVars)
145                 if (p.getFullPathName().compareTo(
146                     p1.getFullPathName()) == 0) {
147                   ConflictPair conflict2 = new ConflictPair(
148                       r, r1);
149                   conflict2.setActionRule1("read");
150                   conflict2.setActionRule2("write");
151                   conflict2.setProperty1(p);
152                   conflict2.setProperty2(p1);
153                   conflicts.add(conflict2);
154                 }
155               for (Property p1 : mayReadVars)
156                 if (p.getFullPathName().compareTo(
157                     p1.getFullPathName()) == 0) {
158                   ConflictPair conflict2 = new ConflictPair(
159                       r, r1);
160                   conflict2.setActionRule1("mayRead");
161                   conflict2.setActionRule2("write");
```

Appendix C (Continued)

```
162                 conflict2.setProperty1(p);
163                 conflict2.setProperty2(p1);
164                 conflicts.add(conflict2);
165               }
166             for (Property p1 : mayWrittenVars)
167               if (p.getFullPathName().compareTo(
168                   p1.getFullPathName()) == 0) {
169                 ConflictPair conflict2 = new ConflictPair(
170                     r, r1);
171                 conflict2.setActionRule1("mayWrite");
172                 conflict2.setActionRule2("write");
173                 conflict2.setProperty1(p);
174                 conflict2.setProperty2(p1);
175                 conflicts.add(conflict2);
176               }
177
178           }
179           for (Property p : mayWrittenVars) {
180             for (Property p1 : mayWrittenVars1)
181               if (p.getFullPathName().compareTo(
182                   p1.getFullPathName()) == 0) {
183                 ConflictPair conflict2 = new ConflictPair(
184                     r, r1);
185                 conflict2.setActionRule1("mayWrite");
186                 conflict2.setActionRule2("mayWrite");
187                 conflict2.setProperty1(p);
```

Appendix C (Continued)

```
188                 conflict2.setProperty2(p1);
189                 conflicts.add(conflict2);
190               }
191             for (Property p1 : readVars1)
192               if (p.getFullPathName().compareTo(
193                   p1.getFullPathName()) == 0) {
194                 ConflictPair conflict2 = new ConflictPair(
195                     r, r1);
196                 conflict2.setActionRule1("mayWrite");
197                 conflict2.setActionRule2("read");
198                 conflict2.setProperty1(p);
199                 conflict2.setProperty2(p1);
200                 conflicts.add(conflict2);
201               }
202             for (Property p1 : mayReadVars1)
203               if (p.getFullPathName().compareTo(
204                   p1.getFullPathName()) == 0) {
205                 ConflictPair conflict2 = new ConflictPair(
206                     r, r1);
207                 conflict2.setActionRule1("mayWrite");
208                 conflict2.setActionRule2("mayRead");
209                 conflict2.setProperty1(p);
210                 conflict2.setProperty2(p1);
211                 conflicts.add(conflict2);
212               }
213           }
```

Appendix C (Continued)

```
214             for (Property p : mayWrittenVars1) {
215               for (Property p1 : readVars)
216                 if (p.getFullPathName().compareTo(
217                     p1.getFullPathName()) == 0) {
218                   ConflictPair conflict2 = new ConflictPair(
219                       r, r1);
220                   conflict2.setActionRule1("read");
221                   conflict2.setActionRule2("mayWrite");
222                   conflict2.setProperty1(p);
223                   conflict2.setProperty2(p1);
224                   conflicts.add(conflict2);
225                 }
226               for (Property p1 : mayReadVars)
227                 if (p.getFullPathName().compareTo(
228                     p1.getFullPathName()) == 0) {
229                   ConflictPair conflict2 = new ConflictPair(
230                       r, r1);
231                   conflict2.setActionRule1("mayRead");
232                   conflict2.setActionRule2("mayWrite");
233                   conflict2.setProperty1(p);
234                   conflict2.setProperty2(p1);
235                   conflicts.add(conflict2);
236                 }
237             }
238           }
239         }
```

Appendix C (Continued)

```
240       }
241     }
242     // concurrency Conflicts on readTriggered Rules
243     if (possibleWriteTriggeredConflicts != null) {
244       System.out
245           .println("Initializing  write triggered conflict identification...");
246       for (PossibleConflict c : possibleWriteTriggeredConflicts) {
247         List<Rule> rules = c.getRules();
248         for (int i = 0; i < rules.size(); i++) {
249           Rule r = rules.get(i);
250           for (int j = i + 1; j < rules.size(); j++) {
251             Rule r1 = rules.get(j);
252             // check if rule r writes a property that is read by r1
253             List<Property> readVars = r.getReadVariables();
254             List<Property> readVars1 = r1.getReadVariables();
255             List<Property> writtenVars1 = r1.getWrittenVariables();
256             List<Property> mayReadVars = r.getMayReadVariables();
257             List<Property> mayReadVars1 = r1.getMayReadVariables();
258             List<Property> mayWrittenVars1 = r1
259                 .getMayWrittenVariables();
260             List<Property> mayWrittenVars = r
261                 .getMayWrittenVariables();
262
263             /*
264              * System.out.println("Analyzing " +
265              * r.getRuleMethodName() + " , " +
```

Appendix C (Continued)

```
266             * r1.getRuleMethodName()); System.out
267             * .println("    r reads: " + readVars.toString());
268             * System.out.println("    r writes: " +
269             * r.getWrittenVariables());
270             * System.out.println("    r1 reads: " +
271             * readVars1.toString());
272             * System.out.println("    r1 writes: " +
273             * writtenVars1.toString());
274             * System.out.println("    r mayReads: " +
275             * mayReadVars.toString());
276             * System.out.println("    r mayWrites: " +
277             * mayWrittenVars.toString());
278             * System.out.println("    r1 mayReads: " +
279             * mayReadVars1.toString());
280             * System.out.println("    r1 mayWrites: " +
281             * mayWrittenVars1.toString()); System.out.println();
282             */
283
284           for (Property p : r.getWrittenVariables()) {
285             for (Property p1 : readVars1)
286               if (p.getFullPathName().compareTo(
287                   p1.getFullPathName()) == 0) {
288                 ConflictPair conflict = new ConflictPair(r,
289                     r1);
290                 conflict.setActionRule1("write");
291                 conflict.setActionRule2("read");
```

Appendix C (Continued)

```
292                 conflict.setProperty1(p);
293                 conflict.setProperty2(p1);
294                 conflicts.add(conflict);
295
296               }
297             for (Property p1 : writtenVars1)
298               if (p.getFullPathName().compareTo(
299                   p1.getFullPathName()) == 0) {
300                 ConflictPair conflict1 = new ConflictPair(
301                     r, r1);
302                 conflict1.setActionRule1("write");
303                 conflict1.setActionRule2("write");
304                 conflict1.setProperty1(p);
305                 conflict1.setProperty2(p1);
306                 conflicts.add(conflict1);
307
308               }
309             for (Property p1 : mayReadVars1)
310               if (p.getFullPathName().compareTo(
311                   p1.getFullPathName()) == 0) {
312                 ConflictPair conflict1 = new ConflictPair(
313                     r, r1);
314                 conflict1.setActionRule1("write");
315                 conflict1.setActionRule2("mayRead");
316                 conflict1.setProperty1(p);
317                 conflict1.setProperty2(p1);
```

Appendix C (Continued)

```
318                 conflicts.add(conflict1);
319               }
320             for (Property p1 : mayWrittenVars1)
321               if (p.getFullPathName().compareTo(
322                   p1.getFullPathName()) == 0) {
323                 ConflictPair conflict1 = new ConflictPair(
324                     r, r1);
325                 conflict1.setActionRule1("write");
326                 conflict1.setActionRule2("mayWrite");
327                 conflict1.setProperty1(p);
328                 conflict1.setProperty2(p1);
329                 conflicts.add(conflict1);
330               }
331           }
332           for (Property p : writtenVars1) {
333             for (Property p1 : readVars)
334               if (p.getFullPathName().compareTo(
335                   p1.getFullPathName()) == 0) {
336                 ConflictPair conflict2 = new ConflictPair(
337                     r, r1);
338                 conflict2.setActionRule1("read");
339                 conflict2.setActionRule2("write");
340                 conflict2.setProperty1(p);
341                 conflict2.setProperty2(p1);
342                 conflicts.add(conflict2);
343               }
```

Appendix C (Continued)

```
344             for (Property p1 : mayReadVars)
345               if (p.getFullPathName().compareTo(
346                   p1.getFullPathName()) == 0) {
347                 ConflictPair conflict2 = new ConflictPair(
348                     r, r1);
349                 conflict2.setActionRule1("mayRead");
350                 conflict2.setActionRule2("write");
351                 conflict2.setProperty1(p);
352                 conflict2.setProperty2(p1);
353                 conflicts.add(conflict2);
354               }
355             for (Property p1 : mayWrittenVars)
356               if (p.getFullPathName().compareTo(
357                   p1.getFullPathName()) == 0) {
358                 ConflictPair conflict2 = new ConflictPair(
359                     r, r1);
360                 conflict2.setActionRule1("mayWrite");
361                 conflict2.setActionRule2("write");
362                 conflict2.setProperty1(p);
363                 conflict2.setProperty2(p1);
364                 conflicts.add(conflict2);
365               }
366
367           }
368           for (Property p : mayWrittenVars) {
369             for (Property p1 : mayWrittenVars1)
```

Appendix C (Continued)

```
370                 if (p.getFullPathName().compareTo(
371                     p1.getFullPathName()) == 0) {
372                   ConflictPair conflict2 = new ConflictPair(
373                       r, r1);
374                   conflict2.setActionRule1("mayWrite");
375                   conflict2.setActionRule2("mayWrite");
376                   conflict2.setProperty1(p);
377                   conflict2.setProperty2(p1);
378                   conflicts.add(conflict2);
379                 }
380               for (Property p1 : readVars1)
381                 if (p.getFullPathName().compareTo(
382                     p1.getFullPathName()) == 0) {
383                   ConflictPair conflict2 = new ConflictPair(
384                       r, r1);
385                   conflict2.setActionRule1("mayWrite");
386                   conflict2.setActionRule2("read");
387                   conflict2.setProperty1(p);
388                   conflict2.setProperty2(p1);
389                   conflicts.add(conflict2);
390                 }
391               for (Property p1 : mayReadVars1)
392                 if (p.getFullPathName().compareTo(
393                     p1.getFullPathName()) == 0) {
394                   ConflictPair conflict2 = new ConflictPair(
395                       r, r1);
```

Appendix C (Continued)

```
396                 conflict2.setActionRule1("mayWrite");
397                 conflict2.setActionRule2("mayRead");
398                 conflict2.setProperty1(p);
399                 conflict2.setProperty2(p1);
400                 conflicts.add(conflict2);
401               }
402             }
403             for (Property p : mayWrittenVars1) {
404               for (Property p1 : readVars)
405                 if (p.getFullPathName().compareTo(
406                     p1.getFullPathName()) == 0) {
407                   ConflictPair conflict2 = new ConflictPair(
408                       r, r1);
409                   conflict2.setActionRule1("read");
410                   conflict2.setActionRule2("mayWrite");
411                   conflict2.setProperty1(p);
412                   conflict2.setProperty2(p1);
413                   conflicts.add(conflict2);
414                 }
415               for (Property p1 : mayReadVars)
416                 if (p.getFullPathName().compareTo(
417                     p1.getFullPathName()) == 0) {
418                   ConflictPair conflict2 = new ConflictPair(
419                       r, r1);
420                   conflict2.setActionRule1("mayRead");
421                   conflict2.setActionRule2("mayWrite");
```

Appendix C (Continued)

```
422                     conflict2.setProperty1(p);
423                     conflict2.setProperty2(p1);
424                     conflicts.add(conflict2);
425                   }
426               }
427             }
428           }
429         }
430       }
431     }
432
433     private void printToFile(String s) {
434       try {
435         FileWriter fstream = new FileWriter("conflicts.xml");
436         BufferedWriter out = new BufferedWriter(fstream);
437         out.write(s);
438         out.close();
439       } catch (Exception e) {
440         e.printStackTrace();
441       }
442     }
443
444     public List<Rule> getRuleList() {
445       return ruleList;
446     }
447
```

Appendix C (Continued)

```
448   public List<ConflictPair> getConflicts() {
449     return this.conflicts;
450   }
451 }
```

C.1 Data Structures

The classes that represents data structures utilized during the conflict identification part are the following:

```
 1 package conflictIdentificator;
 2 import java.util.ArrayList;
 3 import java.util.List;
 4
 5 public class PossibleConflict {
 6   private String parentProperty;
 7   private List<Rule> rules;
 8
 9   public PossibleConflict(String parentProperty) {
10     this.parentProperty = parentProperty;
11     this.rules = new ArrayList<Rule>();
12   }
13
14   public void addRule(Rule r) {
15     this.rules.add(r);
16   }
17
```

Appendix C (Continued)

```
18    public boolean containsAtLeastTwoRules() {
19      if (rules.size() < 2)
20        return false;
21      return true;
22    }
23
24    @Override
25    public String toString() {
26      String s = "";
27      s += parentProperty + "\n";
28      for (Rule r : rules)
29        s += "    " + r.getRuleMethodName() + "\n";
30
31      return s;
32    }
33
34    public int sizeRules() {
35      return rules.size();
36    }
37
38    public List<Rule> getRules() {
39      return this.rules;
40    }
41  }
```

```
1  package conflictIdentificator;
```

Appendix C (Continued)

```
2
3  public class ConflictPair {
4    private Rule rule1;
5    private Rule rule2;
6    private Property property1;
7    private Property property2;
8    private String actionRule1;
9    private String actionRule2;
10
11   public ConflictPair(Rule rule1, Rule rule2) {
12     this.rule1 = rule1;
13     this.rule2 = rule2;
14   }
15
16   public void setActionRule1(String actionRule1) {
17     this.actionRule1 = actionRule1;
18   }
19
20   public String getActionRule1() {
21     return actionRule1;
22   }
23
24   public void setActionRule2(String actionRule2) {
25     this.actionRule2 = actionRule2;
26   }
27
```

Appendix C (Continued)

```
28    public String getActionRule2() {
29      return actionRule2;
30    }
31
32    public String toXmlString() {
33      String xmlString = "<conflict>\n";
34      xmlString += "<property>" + this.property1.getFullPathName()
35          + "</property>\n";
36      xmlString += "<rule1>\n";
37      xmlString += "<ruleMethodName>\n" + this.rule1.getRuleMethodName()
38          + "</ruleMethodName>\n";
39      xmlString += "<action>" + this.actionRule1 + "</action>\n";
40      xmlString += "<pathCondition>"
41          + this.property1.getFirstPathConditions().toString()
42          + "</pathCondition>";
43      xmlString += "</rule1>\n";
44      xmlString += "<rule2>\n";
45      xmlString += "<ruleMethodName>\n" + this.rule2.getRuleMethodName()
46          + "</ruleMethodName>\n";
47      xmlString += "<action>" + this.actionRule2 + "</action>\n";
48      xmlString += "<pathCondition>"
49          + this.property2.getFirstPathConditions().toString()
50          + "</pathCondition>";
51      xmlString += "</rule2>\n";
52      xmlString += "</conflict>\n";
53
```

Appendix C (Continued)

```
54      return xmlString;
55    }
56
57    public void setProperty1(Property property1) {
58      this.property1 = property1;
59    }
60
61    public Property getProperty1() {
62      return property1;
63    }
64
65    public void setProperty2(Property property2) {
66      this.property2 = property2;
67    }
68
69    public Property getProperty2() {
70      return property2;
71    }
72
73    public Rule getRule1() {
74      return this.rule1;
75    }
76
77    public Rule getRule2() {
78      return this.rule2;
79    }
```

Appendix C (Continued)

```
80  }
```

Appendix D

TRIGGERING GRAPH CONSTRUCTION

The triggering graph construction utilizes a personalized graph formed by two kinds of nodes and two kinds of edges.

The first kind of node is the one representing rules and is implemented in the *RuleNode*; the second defines a property in the system and the implementation is in the class *PropertyNode*. This class has two subclasses that are *ReadPropertyNode* and *WritePropertyNode* that represent a read or a write on that property.

The edges from a rule to a property node are represented by the *PropertyToRuleEdge* class while the ones from property to rule nodes are *ConditionalEdge* class. The source code of the classes mentioned above is the following:

```
1  package triggeringGraph;
2
3  import conflictIdentificator.Rule;
4
5  public class RuleNode {
6    private Rule rule;
7
8    public RuleNode(Rule r) {
9      this.setRule(r);
10   }
```

Appendix D (Continued)

```
11
12    public void setRule(Rule rule) {
13      this.rule = rule;
14    }
15
16    public Rule getRule() {
17      return rule;
18    }
19  }
```

```
 1  package triggeringGraph;
 2
 3  public class PropertyNode {
 4    private String propertyName;
 5
 6    public PropertyNode(String propertyName) {
 7      this.setPropertyName(propertyName);
 8    }
 9
10    public void setPropertyName(String propertyName) {
11      this.propertyName = propertyName;
12    }
13
14    public String getPropertyName() {
15      return propertyName;
16    }
```

Appendix D (Continued)

```
17 }
```

```
1  package triggeringGraph;
2
3  public class ReadPropertyNode extends PropertyNode {
4
5    private final String action;
6
7    public ReadPropertyNode(String propertyName) {
8      super(propertyName);
9      this.action = "read";
10   }
11
12 }
```

```
1  package triggeringGraph;
2
3  public class WritePropertyNode extends PropertyNode {
4    private final String action;
5
6    public WritePropertyNode(String propertyName) {
7      super(propertyName);
8      this.action = "write";
9    }
10 }
```

```
1  package triggeringGraph;
```

Appendix D (Continued)

```
2
3
4  public class PropertyToRuleEdge {
5    private RuleNode dest;
6    private PropertyNode source;
7
8    public PropertyToRuleEdge(PropertyNode source, RuleNode dest) {
9      this.setSource(source);
10     this.setDest(dest);
11   }
12
13   public void setDest(RuleNode dest) {
14     this.dest = dest;
15   }
16
17   public RuleNode getDest() {
18     return dest;
19   }
20
21   public void setSource(PropertyNode source) {
22     this.source = source;
23   }
24
25   public PropertyNode getSource() {
26     return source;
27   }
```

Appendix D (Continued)

```
28 }
```

```
1  package triggeringGraph;
2
3  import java.util.List;
4
5  public class ConditionalEdge {
6    private RuleNode source;
7    private PropertyNode dest;
8    private List<String> conditions;
9
10   public ConditionalEdge(RuleNode source, PropertyNode dest,
11       List<String> conditions) {
12     this.setSource(source);
13     this.setDest(dest);
14     this.setConditions(conditions);
15   }
16
17   public void setSource(RuleNode source) {
18     this.source = source;
19   }
20
21   public RuleNode getSource() {
22     return source;
23   }
24
```

Appendix D (Continued)

```
25    public void setDest(PropertyNode dest) {
26      this.dest = dest;
27    }
28
29    public PropertyNode getDest() {
30      return dest;
31    }
32
33    public void setConditions(List<String> conditions) {
34      this.conditions = conditions;
35    }
36
37    public List<String> getConditions() {
38      return conditions;
39    }
40
41  }
```

```
1  package triggeringGraph;
2
3  import java.util.ArrayList;
4  import java.util.HashSet;
5  import java.util.LinkedList;
6  import java.util.List;
7  import java.util.Set;
8
```

Appendix D (Continued)

```
 9 import conflictIdentificator.ConflictPair;
10 import conflictIdentificator.Property;
11 import conflictIdentificator.Rule;
12
13 public class TriggeringGraph {
14   protected Set<RuleNode> rules;
15   protected Set<PropertyNode> properties;
16
17   protected Set<ConditionalEdge> ruleTopropertiesEdges;
18   // protected Map<PropertyNode, Set<RuleNode>> propertyToRulesEdges;
19   protected Set<PropertyToRuleEdge> propertyToRulesEdges;
20
21   public TriggeringGraph(List<Rule> rules) {
22     this.rules = new HashSet<RuleNode>();
23     this.properties = new HashSet<PropertyNode>();
24     // this.propertyToRulesEdges = new HashMap<PropertyNode,
25     // Set<RuleNode>>();
26     this.propertyToRulesEdges = new HashSet<PropertyToRuleEdge>();
27     this.ruleTopropertiesEdges = new HashSet<ConditionalEdge>();
28
29     for (Rule rule : rules) {
30       RuleNode ruleNode = new RuleNode(rule);
31       this.addRuleNode(ruleNode);
32
33       // add the parent property edge for readTriggered rules
34       if (this.getReadTriggeredKinds().contains(rule.getKind())) {
```

Appendix D (Continued)

```
35          PropertyNode parentProperty = this.getPropertyNodeByNameAction(
36              rule.getParent().getFullPathName(), "r");
37          if (parentProperty != null) {
38            this.addPropToRuleEdge(parentProperty, ruleNode);
39          } else {
40            parentProperty = new ReadPropertyNode(rule.getParent()
41                .getFullPathName());
42            this.properties.add(parentProperty);
43            this.addPropToRuleEdge(parentProperty, ruleNode);
44          }
45        }
46        // add the parent property edge for writeTriggered rules
47        if (this.getWriteTriggeredKinds().contains(rule.getKind())) {
48          PropertyNode parentProperty = this.getPropertyNodeByNameAction(
49              rule.getParent().getFullPathName(), "w");
50          if (parentProperty != null) {
51            this.addPropToRuleEdge(parentProperty, ruleNode);
52          } else {
53            parentProperty = new WritePropertyNode(rule.getParent()
54                .getFullPathName());
55            this.properties.add(parentProperty);
56            this.addPropToRuleEdge(parentProperty, ruleNode);
57          }
58        }
59        // any other rule is not triggered by a read or write on a property
60        else {
```

Appendix D (Continued)

```
61
62        }
63        // add link for written variables
64        for (Property p : rule.getWrittenVariables()) {
65          PropertyNode property = this.getPropertyNodeByNameAction(
66              p.getFullPathName(), "w");
67          if (property != null) {
68            this.addRuleToPropEdge(ruleNode, property,
69                p.getFirstPathConditions());
70          } else {
71            property = new WritePropertyNode(p.getFullPathName());
72            this.properties.add(property);
73            this.addRuleToPropEdge(ruleNode, property,
74                p.getFirstPathConditions());
75          }
76        }
77
78        // add link for may written variables
79        for (Property p : rule.getMayWrittenVariables()) {
80          PropertyNode property = this.getPropertyNodeByNameAction(
81              p.getFullPathName(), "w");
82          if (property != null) {
83            this.addRuleToPropEdge(ruleNode, property,
84                p.getFirstPathConditions());
85          } else {
86            property = new WritePropertyNode(p.getFullPathName());
```

Appendix D (Continued)

```
87            this.properties.add(property);
88            this.addRuleToPropEdge(ruleNode, property,
89                p.getFirstPathConditions());
90          }
91        }
92
93        // add link for read variables
94        for (Property p : rule.getReadVariables()) {
95          PropertyNode property = this.getPropertyNodeByNameAction(
96              p.getFullPathName(), "r");
97          if (property != null) {
98            this.addRuleToPropEdge(ruleNode, property,
99                p.getFirstPathConditions());
100         } else {
101           property = new ReadPropertyNode(p.getFullPathName());
102           this.properties.add(property);
103           this.addRuleToPropEdge(ruleNode, property,
104               p.getFirstPathConditions());
105         }
106       }
107
108       // add link for may read variables
109       for (Property p : rule.getMayReadVariables()) {
110         PropertyNode property = this.getPropertyNodeByNameAction(
111             p.getFullPathName(), "r");
112         if (property != null) {
```

Appendix D (Continued)

```
113             this.addRuleToPropEdge(ruleNode, property,
114                 p.getFirstPathConditions());
115           } else {
116             property = new ReadPropertyNode(p.getFullPathName());
117             this.properties.add(property);
118             this.addRuleToPropEdge(ruleNode, property,
119                 p.getFirstPathConditions());
120           }
121         }
122       }
123     }
124
125     // given a property returns the rules linked to it( that fire because of the
126     // property)
127     public List<RuleNode> getLinkedRules(PropertyNode property) {
128       List<RuleNode> rules = new LinkedList<RuleNode>();
129       for (PropertyToRuleEdge c : propertyToRulesEdges) {
130         if (c.getSource().equals(property))
131           rules.add(c.getDest());
132       }
133       return rules;
134     }
135
136     // given a rule returns the property nodes linked to it (that the rule
137     // performs an action on)
138     public List<PropertyNode> getLinkedPropertyNodes(RuleNode rule) {
```

Appendix D (Continued)

```
139     List<PropertyNode> properties = new LinkedList<PropertyNode>();
140     for (ConditionalEdge c : ruleTopropertiesEdges) {
141       if (c.getSource().equals(rule))
142         properties.add(c.getDest());
143     }
144     return properties;
145   }
146
147   public void addRuleNode(RuleNode node) {
148     this.rules.add(node);
149   }
150
151   public void addPropertyNode(PropertyNode node) {
152
153     this.properties.add(node);
154
155   }
156
157   /*
158    * public void addPropToRuleEdge(PropertyNode source, RuleNode dest) { if
159    * (propertyToRulesEdges.containsKey(source)) { if
160    * (!propertyToRulesEdges.get(source).contains(dest))
161    * propertyToRulesEdges.get(source).add(dest); } else { Set<RuleNode> rules
162    * = new HashSet<RuleNode>(); rules.add(dest);
163    * this.propertyToRulesEdges.put(source, rules); } }
164    */
```

Appendix D (Continued)

```
165
166   public void addPropToRuleEdge(PropertyNode source, RuleNode dest) {
167     if (!isPropertyEdgePresent(source, dest)) {
168       PropertyToRuleEdge edge = new PropertyToRuleEdge(source, dest);
169       this.propertyToRulesEdges.add(edge);
170     }
171   }
172
173   private boolean isPropertyEdgePresent(PropertyNode source, RuleNode dest) {
174     for (PropertyToRuleEdge c : propertyToRulesEdges) {
175       if (c.getSource().equals(source) && c.getDest().equals(dest))
176         return true;
177     }
178     return false;
179   }
180
181   public void addRuleToPropEdge(RuleNode source, PropertyNode dest,
182       List<String> conditions) {
183     if (!isEdgePresent(source, dest)) {
184       ConditionalEdge edge = new ConditionalEdge(source, dest, conditions);
185       this.ruleTopropertiesEdges.add(edge);
186     }
187   }
188
189   private boolean isEdgePresent(RuleNode source, PropertyNode dest) {
190     for (ConditionalEdge c : ruleTopropertiesEdges) {
```

Appendix D (Continued)

```
191         if (c.getSource().equals(source) && c.getDest().equals(dest))
192           return true;
193       }
194       return false;
195     }
196
197     public RuleNode getRuleNodeByName(String methodName) {
198       for (RuleNode r : this.rules) {
199         if (r.getRule().getRuleMethodName().compareTo(methodName) == 0)
200           return r;
201       }
202       return null;
203     }
204
205     public PropertyNode getPropertyNodeByNameAction(String fullName,
206         String action) {
207       for (PropertyNode p : this.properties) {
208         if (action.compareTo("w") == 0) {
209           if (p instanceof WritePropertyNode
210               && p.getPropertyName().compareTo(fullName) == 0)
211             return p;
212         } else {
213           if (p instanceof ReadPropertyNode
214               && p.getPropertyName().compareTo(fullName) == 0)
215             return p;
216         }
```

Appendix D (Continued)

```
217       }
218       return null;
219     }
220
221     private List<String> getReadTriggeredKinds() {
222       List<String> kinds = new ArrayList<String>();
223       kinds.add("IS_VALID_KIND");
224       kinds.add("SUGGESTED_VALUE_KIND");
225       kinds.add("IS_REQUIRED_KIND");
226       kinds.add("LOWER_LIMIT_KIND");
227       kinds.add("UPPER_LIMIT_KIND");
228       kinds.add("ALLOWED_VALUE_KIND");
229       return kinds;
230     }
231
232     private List<String> getWriteTriggeredKinds() {
233       List<String> kinds = new ArrayList<String>();
234       kinds.add("POST_SET_KIND");
235       kinds.add("POST_ADD_KIND");
236       kinds.add("LOWER_LIMIT_KIND");
237       kinds.add("UPPER_LIMIT_KIND");
238       kinds.add("ALLOWED_VALUE_KIND");
239       return kinds;
240     }
241
242     public String toString() {
```

Appendix D (Continued)

```
243     String s = "";
244     s += "Rules:\n";
245     for (RuleNode n : this.rules) {
246       s += "  " + n.getRule().getRuleMethodName() + "\n";
247     }
248     s += "Properties:\n";
249     for (PropertyNode n : this.properties) {
250       s += "  " + n.getPropertyName() + "\n";
251     }
252     s += "\n";
253     for (RuleNode n : this.rules) {
254       s += "Rule:\n";
255       s += "  " + n.getRule().getRuleMethodName() + "\n";
256       for (PropertyNode p : this.getLinkedPropertyNodes(n)) {
257         if (p instanceof ReadPropertyNode)
258           s += "    read - " + p.getPropertyName() + "\n";
259         if (p instanceof WritePropertyNode)
260           s += "    write - " + p.getPropertyName() + "\n";
261       }
262     }
263     return s;
264   }
265
266   public String toGraphMLString() {
267     String xml = "<?xml version=\"1.0\" encoding=\"UTF-8\" standalone=\"no\"?>\n"
```

Appendix D (Continued)

```
268         + "<graphml xmlns=\"http://graphml.graphdrawing.org/xmlns\" xmlns:xsi=\"
                http://www.w3.org/2001/XMLSchema-instance\" xmlns:y=\"http://www.
                yworks.com/xml/graphml\" xmlns:yed=\"http://www.yworks.com/xml/yed
                /3\" xsi:schemaLocation=\"http://graphml.graphdrawing.org/xmlns http
                ://www.yworks.com/xml/schema/graphml/1.1/ygraphml.xsd\">\n"
269         + "  <!--Created by yFiles for Java 2.7-->\n"
270         + "  <key for=\"graphml\" id=\"d0\" yfiles.type=\"resources\"/>\n"
271         + "  <key attr.name=\"url\" attr.type=\"string\" for=\"node\" id=\"d1
                \"/>\n"
272         + "  <key attr.name=\"description\" attr.type=\"string\" for=\"node\" id
                =\"d2\"/>\n"
273         + "  <key for=\"node\" id=\"d3\" yfiles.type=\"nodegraphics\"/>\n"
274         + "  <key attr.name=\"Description\" attr.type=\"string\" for=\"graph\" id
                =\"d4\">\n"
275         + "    <default/>\n"
276         + "  </key>\n"
277         + "  <key attr.name=\"url\" attr.type=\"string\" for=\"edge\" id=\"d5
                \"/>\n"
278         + "  <key attr.name=\"description\" attr.type=\"string\" for=\"edge\" id
                =\"d6\"/>\n"
279         + "  <key for=\"edge\" id=\"d7\" yfiles.type=\"edgegraphics\"/>\n"
280         + "  <graph edgedefault=\"directed\" id=\"G\">";
281     for (RuleNode n : this.rules) {
282       xml += "<node id=\"" + n.getRule().getRuleMethodName() + "\">";
283       xml += "<data key=\"d3\">\n"
284           + "          <y:ShapeNode>\n"
```

Appendix D (Continued)

```
285         + "             <y:Geometry height=\"30.0\" width=\"30.0\" x=\"69.0\" y
                =\"45.0\"/>\n"
286         + "             <y:Fill color=\"#FFCC00\" transparent=\"false\"/>\n"
287         + "             <y:BorderStyle color=\"#000000\" type=\"line\" width
                =\"1.0\"/>\n"
288         + "             <y:NodeLabel alignment=\"center\" autoSizePolicy=\"
                content\" borderDistance=\"0.0\" fontFamily=\"Dialog\" fontSize
                =\"13\" fontStyle=\"plain\" hasBackgroundColor=\"false\"
                hasLineColor=\"false\" height=\"19.310546875\" modelName=\"internal
                \" modelPosition=\"c\" textColor=\"#000000\" visible=\"true\" width
                =\"23.97607421875\" x=\"3.011962890625\" y=\"5.3447265625\">"
289         + n.getRule().getRuleMethodName().hashCode() + "\n"
290         + " </y:NodeLabel>\n"
291         + "             <y:Shape type=\"ellipse\"/>\n"
292         + "         </y:ShapeNode>\n" + "       </data>\n"
293         + "</node>";
294     }
295     for (PropertyNode n : this.properties) {
296       String s = "";
297       if (n instanceof ReadPropertyNode)
298         s = "Read";
299       else
300         s = "Write";
301       xml += "<node id=\"" + s + n.getPropertyName() + "\">";
302       xml += "<data key=\"d3\">\n"
303           + "         <y:ShapeNode>\n"
```

Appendix D (Continued)

```
304         + "            <y:Geometry height=\"30.0\" width=\"30.0\" x=\"69.0\" y
                =\"45.0\"/>\n"
305         + "            <y:Fill color=\"#FFCC00\" transparent=\"false\"/>\n"
306         + "            <y:BorderStyle color=\"#000000\" type=\"line\" width
                =\"1.0\"/>\n"
307         + "            <y:NodeLabel alignment=\"center\" autoSizePolicy=\"
                content\" borderDistance=\"0.0\" fontFamily=\"Dialog\" fontSize
                =\"13\" fontStyle=\"plain\" hasBackgroundColor=\"false\"
                hasLineColor=\"false\" height=\"19.310546875\" modelName=\"internal
                \" modelPosition=\"c\" textColor=\"#000000\" visible=\"true\" width
                =\"23.97607421875\" x=\"3.011962890625\" y=\"5.3447265625\">"
308         + n.getPropertyName() + s + "\n" + "  </y:NodeLabel>\n"
309         + "            <y:Shape type=\"rectangle\"/>\n"
310         + "          </y:ShapeNode>\n" + "        </data>\n"
311         + "</node>";
312     }
313
314     for (ConditionalEdge c : this.ruleTopropertiesEdges) {
315       String s = "";
316       if (c.getDest() instanceof ReadPropertyNode) {
317         s = "Read";
318       } else
319         s = "Write";
320       xml += "<edge id=\"" + c.hashCode() + "\" source=\""
321           + c.getSource().getRule().getRuleMethodName()
322           + "\" target=\"" + s + c.getDest().getPropertyName()
```

Appendix D (Continued)

```
323             + "\"></edge>";
324         }
325
326         /*
327          * for (Iterator iter = this.propertyToRulesEdges.entrySet().iterator();
328          * iter .hasNext();) { Map.Entry entry = (Map.Entry) iter.next();
329          * PropertyNode key = (PropertyNode) entry.getKey(); String s = ""; if
330          * (key instanceof ReadPropertyNode) s = "Read"; else s = "Write";
331          * Set<RuleNode> rules = (Set<RuleNode>) entry.getValue(); for (RuleNode
332          * n : rules) { xml += "<edge id=\"" + entry.hashCode() + "\" source=\""
333          * + s + key.getPropertyName() + "\" target=\"" +
334          * n.getRule().getRuleMethodName() + "\"></edge>"; } }
335          */
336
337         for (PropertyToRuleEdge c : this.propertyToRulesEdges) {
338           String s = "";
339           if (c.getSource() instanceof ReadPropertyNode) {
340             s = "Read";
341           } else
342             s = "Write";
343           xml += "<edge id=\"" + c.hashCode() + "\" source=\"" + s
344               + c.getSource().getPropertyName() + "\" target=\""
345               + c.getDest().getRule().getRuleMethodName() + "\"></edge>";
346         }
347
348         xml += " </graph>\n" + "  <data key=\"d0\">\n" + "    <y:Resources/>\n"
```

Appendix D (Continued)

```
349           + "  </data>\n" + "</graphml>";
350       return xml;
351     }
352
353     public List<PropertyNode> getPredProperties(RuleNode rule) {
354       List<PropertyNode> prec = new LinkedList<PropertyNode>();
355       for (PropertyToRuleEdge e : this.propertyToRulesEdges) {
356         if (e.getDest().equals(rule))
357           prec.add(e.getSource());
358       }
359       return prec;
360     }
361
362     public List<RuleNode> getPredRules(PropertyNode property) {
363       List<RuleNode> prec = new LinkedList<RuleNode>();
364       for (ConditionalEdge e : this.ruleTopropertiesEdges) {
365         if (e.getDest().equals(property))
366           prec.add(e.getSource());
367       }
368       return prec;
369     }
370 }
```

CITED LITERATURE

1. Buchanan, B. G. and Duda, R. O.: Principles of rule-based expert systems. Technical report, Stanford University, Stanford, CA, USA, 1982.

2. Giarratano, J. C. and Riley, G.: Expert Systems: Principles and Programming. Pacific Grove, CA, USA, Brooks/Cole Publishing Co., 1989.

3. Hendrick, S. D.: Worldwide business rules management systems 2009-2013 forecast update and 2008 vendor shares. IDC, 10 2009.

4. Sinur, J.: The art and science of rules vs. process flows. Gartner, 03 2009.

5. Smith, S. and Kandel, A.: Verification and Validation of Rule-Based Expert Systems. Boca Raton, FL, USA, CRC Press, Inc., 1994.

6. Avritzer, A., Ros, J. P., and Weyuker, E. J.: Reliability testing of rule-based systems. IEEE Software, 13(5):76–82, 1996.

7. Waldinger, R. J. and Stickel, M. E.: Proving properties of rule-based systems. International Journal of Software Engineering and Knowledge Engineering, 2(1):121–144, 1992.

8. Schulte, W. and Sinur, J.: Rule engines and event processing. Gartner, 03 2009.

9. McCoy, D. W.: Taking the mystery out of business rule representation. Gartner, 03 2009.

10. Harris-Ferrante, K. and Forte, S.: Hype cycle for p&c insurance. Gartner, 07 2009.

11. Wikipedia: http://en.wikipedia.org/wiki/expert_system, 2010.

12. Russel, S. J. and Norvig, P.: Artificial Intelligence A Modern Approach. Prentice Hall, second edition edition, 2003.

13. Friedman, E. and Hill: Jess in Action: Rule-Based Systems in Java. Manning, 2003.

14. Abraham, A.: Rule-based expert systems. P. H. Sydenham & R. Thorn (Eds.), Handbook of measuring system design, pages 909–919, 2005.

15. Mohan, C. K.: Frontiers of expert systems. Kluwer Academic Publishers, 2000.

16. Jackson, P.: Introduction to Expert Systems, 3rd Edition. Addison-Wesley, 1999.

17. Bundy, A.: How to improve the reliability of expert systems. In Proceedings of Expert Systems '87 on Research and Development in Expert Systems IV, pages 3–17, New York, NY, USA, 1987. Cambridge University Press.

18. ed. E. Hollnagel The reliability of expert systems. Upper Saddle River, NJ, USA, Prentice-Hall, Inc., 1989.

19. Horwitz, S., Reps, T., and Binkley, D.: Interprocedural slicing using dependence graphs. SIGPLAN Not., 23:35–46, June 1988.

20. King, J. C.: A program verifier. In IFIP Congress (1), pages 234–249, 1971.

21. King, J. C.: Symbolic execution and program testing. Commun. ACM, 19(7):385–394, 1976.

22. eds. L. A. Clarke and D. J. Richardson Symbolic evaluation methods for program analysis.. Prentice-Hall, 1981.

23. http://www.antlr.org/wiki/display/antlr3/antlr+v3+documentation.

24. Parr, T.: The Definitive ANTLR Reference. The Pragmatic Programmer, 2007.

25. Wikipedia: http://en.wikipedia.org/wiki/symbol_table, 2010.

26. rni Einarsson, A. and Nielsen, J. D.: A survivor's guide to java program analysis with soot. Technical report, University of Aarhus, 2008.

27. Wikipedia: http://en.wikipedia.org/wiki/symbolic_execution, 2010.

28. King, J. C.: Symbolic execution and programtesting. Technical report, IBM Thomas J. Watson Research Center, 1976.

29. http://graphml.graphdrawing.org/, 2002.