

**GENERATING  
RUNTIME VERIFICATION SPECIFICATIONS  
BASED ON STATIC CODE ANALYSIS ALERTS**



A Thesis

by

Yunus Kılıç

Submitted to the  
Graduate School of Sciences and Engineering  
In Partial Fulfillment of the Requirements for  
the Degree of

Master of Science

in the  
Department of Computer Science

Özyeğin University  
December 2017

Copyright © 2017 by Yunus Kılıç

**GENERATING  
RUNTIME VERIFICATION SPECIFICATIONS  
BASED ON STATIC CODE ANALYSIS ALERTS**

Approved by:

---

Assoc. Prof. Hasan Sözer (Advisor)  
Department of Computer Science  
*Özyeğin University*

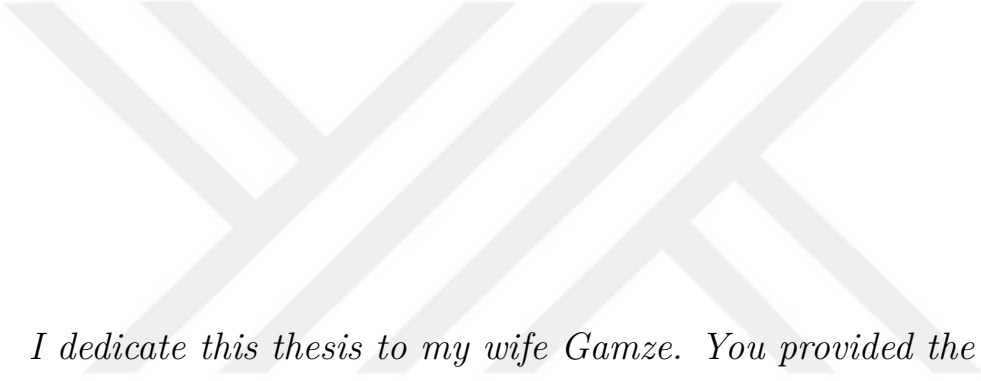
---

Asst. Prof. Mehmet Aktaş  
Department of Computer Engineering  
*Yıldız Technical University*

---

Asst. Prof. Barış Aktemur  
Department of Computer Science  
*Özyeğin University*

Date Approved: ... ..... 2017



*I dedicate this thesis to my wife Gamze. You provided the inspiration  
necessary for me to complete this process.*

## ABSTRACT

There are various approaches in order to find bugs in a software system. One of these approaches is static code analysis, which tries to achieve this goal by analyzing code without executing it. Another complementary approach is runtime verification, which is employed to verify dynamic system behavior with respect to a set of specifications at runtime. These specifications are often created manually based on system requirements and constraints. In this thesis, we propose a novel methodology and tool support for automatically generating runtime verification specifications based on alerts that are reported by static code analysis tools. We introduce a domain specific language for defining a set of rules to be checked for an alert type. Violations of the rules indicate either the absence or existence of an actual bug designated by the instances of that alert type. Formal verification specifications are automatically generated for each reported alert instance based on the defined rules. Then, runtime monitors are automatically synthesized and integrated into the system. These monitors report detected errors or false positive alerts during software execution. The set of rules can be reused across different projects. We performed case studies with two open source software systems to illustrate this. Our tool currently supports the use of two different static code analysis tools for generating runtime monitors in Java language. It is designed to be extendible for supporting other tools as well.

## ÖZETÇE

Yazılım hatalarının bulunması amacıyla kullanılan birçok yöntem bulunmaktadır. Bu yöntemlerden biri olan statik kod analizi koddaki hataların, kodun çalıştırılmadan ortaya çıkarılmasını sağlamaktadır. Bunu tamamlayıcı nitelikte olan çalışma zamanı doğrulama ise dinamik sistem davranışlarını yazılmış olan kurallara göre kontrol etmek için kullanılmaktadır. Bu kural listesi genellikle sistem gereksinimleri ve kısıtlarına göre manuel olarak oluşturulmaktadır. Bu tezde, statik kod analizi araçlarının oluşturduğu uyarılardan çalışma zamanı doğrulama kurallarını otomatik olarak oluşturan yeni bir yöntem ve araç geliştirilmiştir. Alana özgü bir dil geliştirilerek, uyarı tiplerine özgü kurallar tanımlanması sağlanmıştır. Oluşturulan bu kuralların ihlal edilip edilmediğine göre hatanın gerçekleşip gerçekleşmediğine karar verilmektedir. Çalışma zamanı doğrulama kuralları her bir uyarı için daha önceden tanımlanmış kurallara göre otomatik oluşturulmaktadır. Daha sonra ise, oluşan bu kurallara ilişkin çalışma izleyiciler otomatik sentezlenerek, sisteme entegre edilmektedir. Bu izleyiciler yazılımla birlikte çalışarak tespit edilen hataları ve yanlış üretilmiş uyarıları raporlamaktadır. Bir kere oluşturulan kurallar farklı projelerde kullanılabilir. Bu durumu gösterebilmek için iki farklı açık kaynak kod üzerinde vaka çalışmaları gerçekleştirilmiştir. Aracımız şu anda Java programlama dili için uyarı üreten iki farklı statik kod analiz aracını desteklemektedir. Ayrıca bu araç, yeni statik kod araçlarını destekleyebilecek şekilde tasarlanmıştır.

## ACKNOWLEDGMENTS

Firstly, I would like to thank my advisor, Dr. Hasan Sözer for all his help and guidance he has provided over the past two years. I would also like to thank Dr. Tankut Barış Aktemur and Dr. Mehmet Aktaş for accepting to be part of the evaluation committee for this thesis.

Secondly, I deeply thank my wife, Gamze Kılıç, my mother, Hanife Kılıç and my father, Yalçın Kılıç for their unconditional trust, timely encouragement, and endless patience. It was their love that raised me up again when I got weary.

This work is supported by The Scientific and Research Council of Turkey (TUBITAK), grant number 113E548.

# TABLE OF CONTENTS

<b>DEDICATION</b> . . . . .	<b>iii</b>
<b>ABSTRACT</b> . . . . .	<b>iv</b>
<b>ÖZETÇE</b> . . . . .	<b>v</b>
<b>ACKNOWLEDGMENTS</b> . . . . .	<b>vi</b>
<b>LIST OF TABLES</b> . . . . .	<b>ix</b>
<b>LIST OF FIGURES</b> . . . . .	<b>x</b>
<b>I INTRODUCTION</b> . . . . .	<b>1</b>
<b>II BACKGROUND</b> . . . . .	<b>4</b>
2.1 Static Code Analysis . . . . .	4
2.2 Runtime Verification . . . . .	10
<b>III RELATED WORK</b> . . . . .	<b>16</b>
<b>IV OVERALL APPROACH</b> . . . . .	<b>18</b>
4.1 Rule Specification for Alerts . . . . .	19
4.2 Generation of Runtime Verification Specifications . . . . .	22
4.3 Generation of Runtime Monitors . . . . .	24
<b>V EVALUATION</b> . . . . .	<b>25</b>
5.1 Subject Systems and Tools . . . . .	25
5.2 Rule Specification . . . . .	27
5.3 Runtime Verification Specification Generation . . . . .	28
5.4 Runtime Verification Monitor Generation . . . . .	29
5.5 Reuse of Rules Across Projects . . . . .	29
5.6 Reuse of Rules Across Tools . . . . .	32
<b>VI RESULTS AND DISCUSSION</b> . . . . .	<b>33</b>
<b>VII CONCLUSIONS AND FUTURE WORK</b> . . . . .	<b>36</b>

**APPENDIX A — JAVALTl SYNTAX . . . . . 37**  
**APPENDIX B — SCAT2RV TOOL PLUGIN INTERFACE . . . . . 39**  
**APPENDIX C — SCAT2RV TOOL USER MANUAL . . . . . 41**  
**REFERENCES . . . . . 43**





## LIST OF TABLES

1	Common types of static code analysis alerts. . . . .	4
2	Supported platforms, languages and formalisms by MOP (NA: Not Available). . . . .	11
3	A sample list of alert types supported by FindBugs and PMD together with the corresponding rule specifications. . . . .	23
4	Subject software systems. . . . .	25
5	Ratio of alerts for which RV specifications can be automatically generated. . . . .	34
6	The list of SCAT2RV command line arguments. . . . .	42

## LIST OF FIGURES

1	Taxonomy of static code analysis tools [14]. . . . .	6
2	A snapshot of the FindBugs GUI [15]. . . . .	7
3	A snapshot of the PMD XPath GUI [17]. . . . .	9
4	The overall approach. . . . .	18



# CHAPTER I

## INTRODUCTION

Static code analysis is a type of technique that involves the process of analyzing source code without executing it [1]. This technique is mainly used to find potential issues like bugs and bad programming practices in programs [2]. The detected issues are reported in the form of a list of alerts. These alerts can point to, for instance, uninitialized variables, unreachable code, resource leaks, unused variables and security vulnerabilities [3]. Static analysis tools [4] can generate alerts by checking the violation of predefined patterns throughout control flow and data flow paths in the program. Although these tools have been successfully utilized and used in the software industry [5, 6, 7], they have limitations as well. One of the major challenges is to filter out *false positive alerts*. These are the alerts that do not actually point to any real problem with the software. They are also called as *unactionable alerts* [8] since no action has to be taken concerning these alerts. It takes a significant amount of time to manually review each reported alert and obtain the list of actionable alerts [5]. It was empirically shown that existing static code analysis tools (SCATs) have false positive rates ranging from 30% up to 100% [9].

Runtime verification (RV) is a complementary approach to static code analysis. RV [10] is the process of checking if the software behavior at runtime satisfies or violates a given correctness property. Unlike other formal verification techniques such as model checking, the verification scope is narrowed down to a single execution trace of the software during its actual execution at the operational phase. As a result, the verification does not necessarily cover all possible execution traces; however, the verification is performed with a higher precision. The correctness property that is

checked at runtime can be specified with different formalisms such as finite state machines, regular expressions and temporal logic [11, 12]. There exist techniques and tools [11, 13], which can take such formal specifications as input and automatically synthesize monitors for the target system. However, the creation of formal verification specifications is often performed manually based on system requirements and constraints. This process requires effort and expertise. Moreover, system requirements and constraints are often not documented explicitly or formally. As a result, the focus of RV (what should be checked at runtime?) is not always clear.

In this thesis, we present a novel approach and a tool, named SCAT2RV, to generate runtime verification monitors automatically from static code analysis alerts. We introduce a simple domain specific language (DSL) for defining one or more rules to be checked for an alert type. There are two types of rules: *i)* a rule for *falsifying* an alert. The violation of this rule represents a counter-example that falsifies the alert. For example, the alert can indicate that a variable is never initialized. In fact, the variable might be initialized via some mechanism that can not be detected by static analysis. If a rule is defined as the corresponding variable must never be initialized, its violation at runtime represents a counter-example that falsifies the alert. *ii)* a rule for *detecting* the error designated by an alert. For example, the alert can indicate that an object can be left null and its value is not checked before use. One can specify that the corresponding object must not be subject to a *null pointer* exception. The violation of this rule detects the error pointed out by the alert, which can be reported together with the associated alert explanation as diagnostic information. SCAT2RV automatically generates formal verification specifications for each reported alert instance based on the defined rules for the corresponding alert type. Then, runtime monitors are automatically synthesized and integrated into the system. These monitors report detected errors or falsified alerts during software execution.

We illustrate our approach for the RV of two open source software systems. We

define a rule for a particular alert type for one of these systems. RV monitors are automatically generated for each instance of the alert type based on the defined rule. Then, we apply the approach to the other system. We reuse the same rule defined before. RV monitors are automatically generated from alerts regarding the second system without any manual effort. Hence, we show that rules can be used for various instances of an alert type as well as various instances across projects.

SCAT2RV currently supports the use of two different SCATs for generating runtime monitors in Java language. It is designed to be extendible for supporting other tools as well. A new tool can be integrated by only providing a plugin module in order to parse the particular alert format used by the SCAT.

The remainder of this thesis is organized as follows. In the following chapter, we provide background information on static code analysis in general and particular SCATs that are employed in our case studies. We summarize the related studies in Chapter 3. We present the overall approach in Chapter 4. The approach is illustrated and evaluated in Chapter 5, in the context of two case studies. We present and discuss the results in Chapter 6. Finally, in Chapter 7, we provide the conclusions and discuss possible future directions.

## CHAPTER II

### BACKGROUND

In this chapter, we provide brief background information on static code analysis and runtime verification. We also explain the set of tools used in our case studies.

#### *2.1 Static Code Analysis*

A Static Code Analysis Tool (SCAT) creates a list of possibly hundreds of alerts without executing code [3]. The analysis is usually performed by analyzing the source code of the program only. There exists a static code analysis tool for almost all of the modern programming languages. Some of these tools can be integrated with Integrated Development Environments (IDEs) to make them an integral part of the software development lifecycle. As a result, developers can use SCATs across different projects. The alert list basically points out potential errors and it is very useful in order to identify problems early. Common list of alert types and sample alert descriptions are listed in Table 1.

**Table 1:** Common types of static code analysis alerts.

<b>Alert Category</b>	<b>Example Alert Description</b>
Bad practice	Confusing method names
Correctness	Impossible cast, instanceof will always return false
Malicious code	Field should be both final and package protected
Multithreaded	Possible double check of field
Performance	Boxed value is unboxed and then immediately reboxed
Security	Empty database password

SCATS have some drawbacks as well. One of the major drawbacks is the existence of false positive alerts. The term false positive means that the existence of some property or situation is pointed as positive but actually it does not exist. In our case, false positive means that a generated alert does not really point to a bug. This circumstance might sound insignificant. However, false positive alerts lead to a huge problem for large-scale systems, where a SCAT might report thousands of alerts. Developers have to scrutinize every alert and check each of them if it points to a real bug or not. So false positive alerts cause a significant loss of time and effort.

A taxonomy of SCATs is provided [14] as depicted in Figure 1. Hereby, SCATs are categorized with respect to 10 properties. *Input* defines what types of files can be loaded into the tool. *Releases* mean how many releases take place per year. *Supported languages* are defined to clarify the programming languages supported. *Technology* is related to particular technologies are used for searching errors in code. *Rules* define the types of rules that are checked on the source code. *Configurability* is regarding the ability and mechanisms to customize tool. *Extensibility* is a boolean property that defined if the tool can be extended with custom rules or not. *Availability* differentiates among free, commercial and open source tools. *User Experience* lists categories relevant for usability of the tool. *Output* defines the possible output formats supported by the tool.

In the scope of this thesis, we used two SCATs, which are named FindBugs<sup>1</sup> and PMD<sup>2</sup>. In the following, we explain these tools in more detail from the viewpoint of the SCAT taxonomy [14] depicted in Figure 1.

---

<sup>1</sup><http://findbugs.sourceforge.net/>

<sup>2</sup><https://pmd.github.io/>

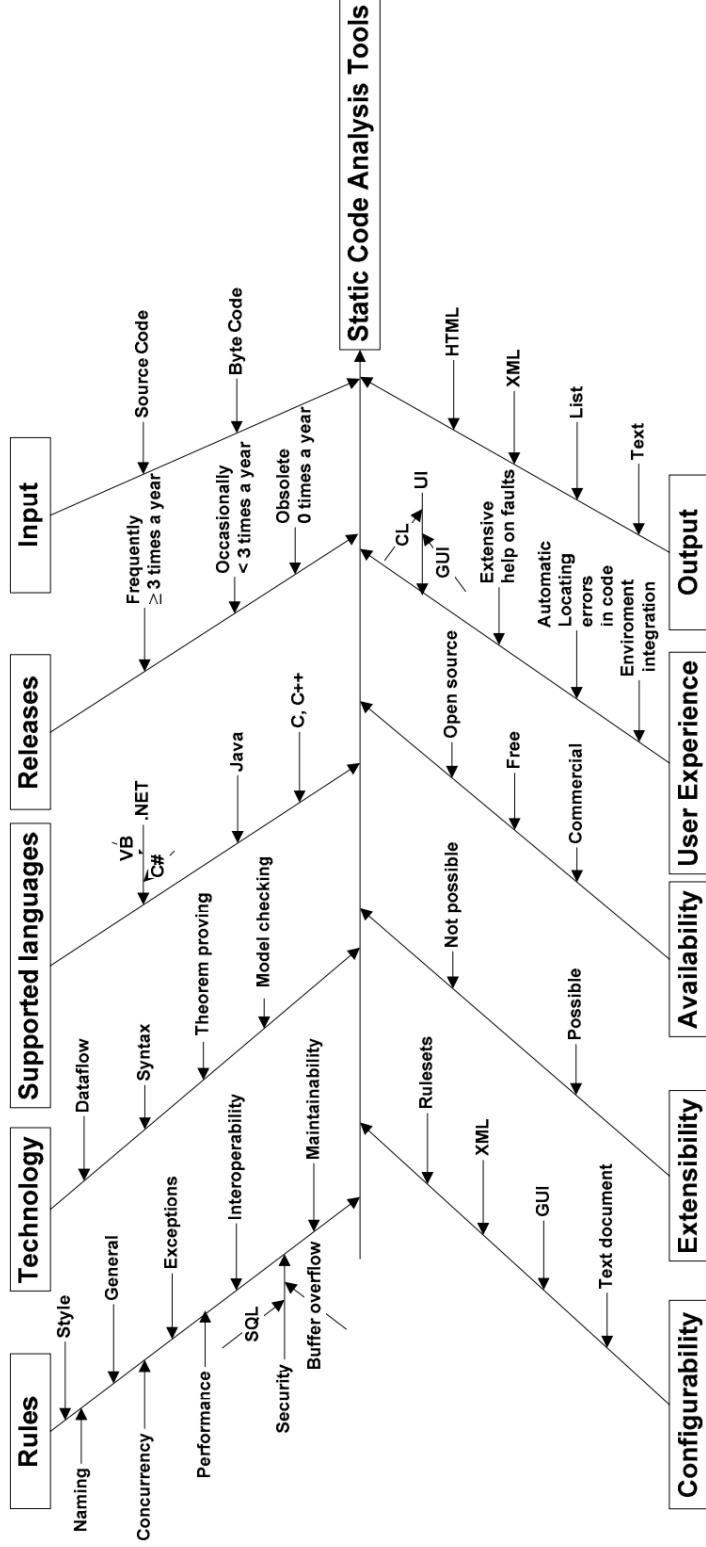


Figure 1: Taxonomy of static code analysis tools [14].



### 2.1.1 FindBugs

FindBugs takes bytecode as input. It is released occasionally, which means less frequent than 3 releases in a year. It supports only the Java language. It employs syntax rules and data flow to search for errors. General, style, concurrency, and performance rules can be checked with FindBugs. It is a configurable tool, where the user can configure rulesets. Extensibility with custom rules is supported. It is an open source software, distributed under the terms of the Lesser GNU Public License [15]. FindBugs can be used stand-alone also it supports integration with lots of various IDEs such as Eclipse<sup>3</sup>, IntelliJ<sup>4</sup>, NetBeans<sup>5</sup>, etc. FindBugs has a Graphical User Interface (GUI) for stand-alone use. Figure 2 depicts a snapshot of this GUI.

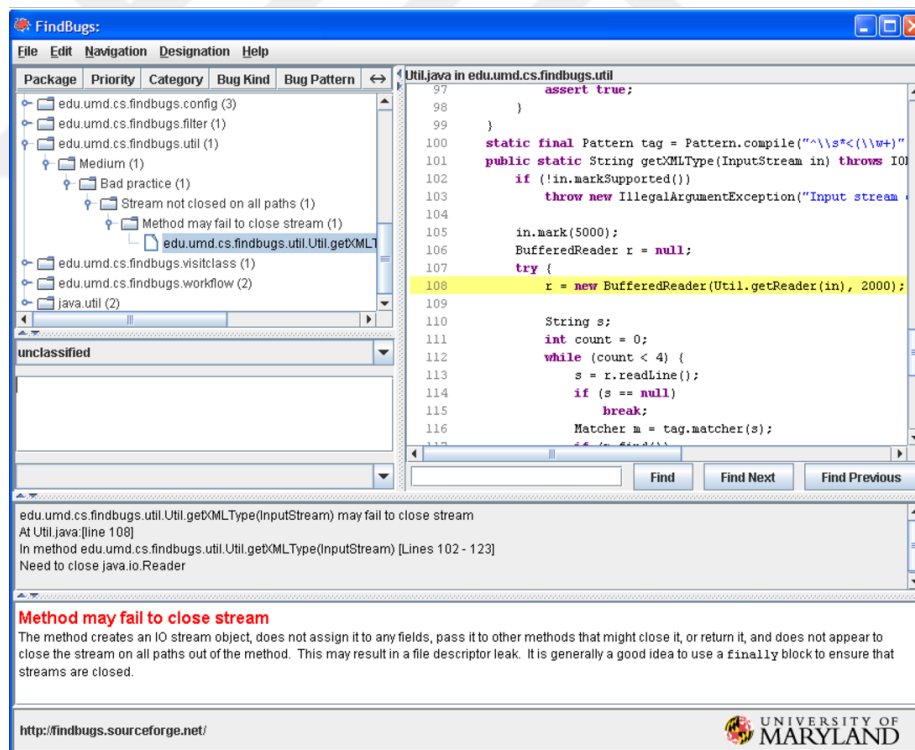


Figure 2: A snapshot of the FindBugs GUI [15].

<sup>3</sup><https://www.eclipse.org/>

<sup>4</sup><https://www.jetbrains.com/idea/>

<sup>5</sup><https://netbeans.org/>

The top left part of the GUI is used for providing source code information. Hereby, you can find package and class names with matched alert types. You can see the corresponding line of code highlighted at the top right part of the panel. The bottom part is used for presenting the definition and description of the selected alert. The list of alerts can be exported in the form of an XML file. The spectrum of types of alert is very wide in Findbugs, ranging from security issues to bad style [15]. Some of the bug categories are listed below:

- Single-threaded correctness issues
- Thread/synchronization correctness issues
- Performance issues
- Security and vulnerability issues in malicious, untrusted code

In the background, FindBug employs BCEL<sup>6</sup>, which is an open source tool used for manipulating and analyzing Java bytecode. FindBugs checkers are designed to use the Visitor design pattern [16]. Each class in the source code is visited by a checker. These checkers traverse the control flow graph derived from the source code for analysis.

### **2.1.2 PMD**

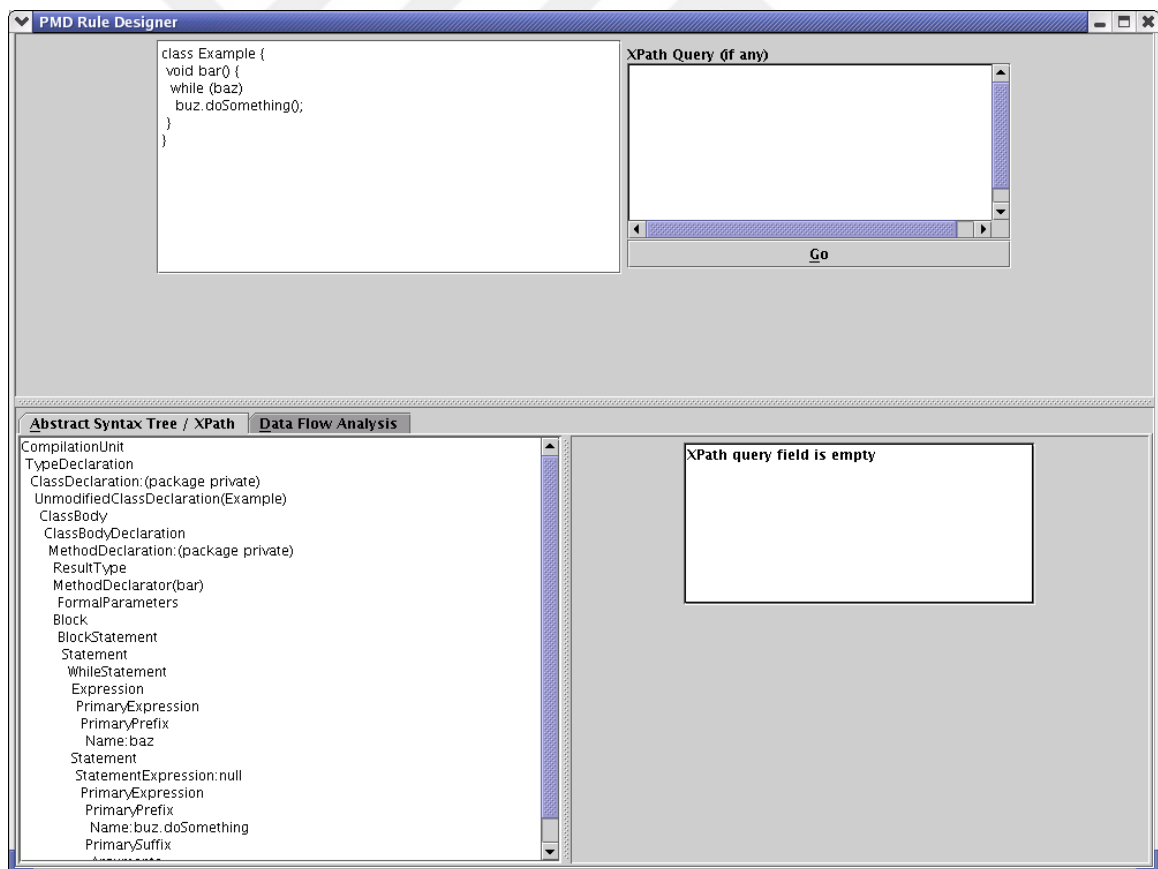
PMD is another SCAT. It takes source code instead of bytecode as input. It is released frequently, i.e., more than 3 releases in a year. It supports various languages including Java, JavaScript, Apex, PLSQL, Apache Velocity, XML, and XSL [17]. PMD employs syntax rules and data flow information to detect errors. It is a configurable and extendable tool like Findbugs. PMD finds common programming flaws like unused variables, empty catch blocks, and unnecessary object creation. It mainly focuses on

---

<sup>6</sup><https://commons.apache.org/proper/commons-bcel/>

code quality and security issues [18]. PMD can be used as a stand-alone application as well as part of various IDEs.

In this thesis work, we used PMD Java because SCAT2RV supports the generation of runtime monitors for Java programs. PMD uses JavaCC<sup>7</sup> to parse Java source code and generate its Abstract Syntax Tree (AST). It provides an API to traverse this AST and define specialized checkers for custom rules. These checkers can be implemented in the form of either Java code or XPath expressions. Checkers in Java are implemented as Java classes that extend *net.sourceforge.pmd.AbstractRule*. Checkers in the form of XPath queries can be defined via the PMD XPath GUI as depicted in Figure 3.



**Figure 3:** A snapshot of the PMD XPath GUI [17].

---

<sup>7</sup><https://javacc.org/>

As it can be seen in the snapshot of the PMD XPath GUI (Figure 3), the source code is listed at the top left part of the panel. The corresponding AST is presented in the left bottom. The right part of the panel is used for specifying the XPath query. XPath<sup>8</sup> is a language for addressing parts of an XML document, designed to be used by both XSLT and XPointer. One needs to write an XPath expression that matches the violation of the custom rule to be introduced.

## 2.2 Runtime Verification

In this section, we provide background information on Runtime Verification (RV), for which we adopt the following definition.

**Definition:** *"RV is the discipline of computer science that deals with the study, development, and application of those verification techniques that allow checking whether a run of a system under scrutiny satisfies or violates a given correctness property"* [19].

The correctness property can be considered as the expected behavior. The difference between the expected behavior and the observed behavior is interpreted as a software failure. RV monitors system with respect to written specifications in order to be sure that actual and expected behaviors are the same. However, RV can only check given specifications that define the correctness property or expected behavior. These specifications have to be written manually.

Monitoring-Oriented Programming, abbreviated as MOP<sup>9</sup>, is a framework for RV, where runtime monitors are automatically synthesized from formal specifications. Then monitors are integrated into source code in order to track dynamic behavior during execution. A specification is either violated or validated at runtime. Then a user-defined action will be triggered automatically. By using this action, for instance, runtime recovery can be provided. On the other hand, if specification is

---

<sup>8</sup><https://www.w3.org/TR/xpath/>

<sup>9</sup><http://fsl.cs.illinois.edu/index.php/MOP>

**Table 2:** Supported platforms, languages and formalisms by MOP (NA: Not Available).

<i>Formalism/Lang.</i>	<b>FSM</b>	<b>ERE</b>	<b>CFG</b>	<b>PTLTL</b>	<b>LTL</b>	<b>PTCaRet</b>	<b>SRS</b>
<b>JavaMOP</b>	JavaFSM	JavaERE	JavaCFG	JavaPTLTL	JavaLTL	JavaPTCaRet	JavaSRS
<b>BusMOP</b>	BusFSM	BusERE	NA	BusLTL	NA	NA	NA
<b>ROSMOP</b>	ROSMOP	NA	ROSCFG	NA	NA	NA	NA

validated, logging can be done. Basically, MOP can be taught as an extension of programming languages with logic, whose benefits are improving safety, reliability, and dependability of a system by monitoring its requirements against its implementation at runtime [20]. MOP is generic both with respect to the underlying programming language and the requirements specification formalism in which properties are expressed. The list of platforms and formalisms currently supported by MOP<sup>10</sup> is listed in Table 2. Hereby, Y-axis lists the supported platforms. *JavaMOP* is a MOP tool for Java. *BusMOP* is a MOP tool for monitoring consumer off-the-shelf components over the PCI bus. *ROSMOP* is a MOP tool for the Robot Operating System (ROS). X axis represents Logic Plugins, each of which support a formalism as listed below.

- FSM: Finite State Machines
- ERE: Extended Regular Expressions
- CFG: Context Free Grammars
- PTLTL: Past Time Linear Temporal Logic
- LTL: Linear Temporal Logic
- PTCaRet: Past Time LTL with Calls and Returns
- SRS: String Rewriting Systems

In our approach, we used JavaLTL for rule specification. Linear Temporal Logic, abbreviated as LTL, is used for defining a proposition on *"an infinite sequence of states where each point in time has a unique successor, based on a linear-time perspective"* [21]. LTL plugin for MOP allows one to synthesize monitors for properties that are defined in the form of linear temporal logic (LTL) descriptions. The LTL

---

<sup>10</sup><http://fsl.cs.illinois.edu/index.php/MOP>

plugin supports both past and future time LTL operators. LTL syntax can be found in Appendix A.

```
1 import java.util.*;
2 public class HasNext {
3     public static void main(String[] args){
4         Vector<Integer> v = new Vector<Integer>();
5         v.add(1);
6         v.add(2);
7         v.add(4);
8         v.add(8);
9
10        Iterator i = v.iterator();
11        int sum = 0;
12
13        if(i.hasNext()){
14            sum += (Integer)i.next();
15            sum += (Integer)i.next();
16            sum += (Integer)i.next();
17            sum += (Integer)i.next();
18        }
19
20        System.out.println("sum: " + sum);
21    }
22 }
```

**Listing 2.1:** Example source code that is monitored with JavaLTL.

In the following, we present an example to illustrate and clarify the use of MOP. The program to be monitored is provided in Listing 2.1. This program works correctly without any errors. This is because 4 elements are added to a vector after which the *next* method is called 4 times. Existence of elements is checked with the *hasNext* method but this check is performed only once. What if 3 elements were added instead of 4? Then, there would be an error. So, this check should have been performed before every access to the data structure. One should ensure that an iterator has further elements before retrieving these. To validate this rule/property, one can write a JavaLTL specification as listed in Listing 2.2. The specification is called *HasNext* and it takes an *Iterator* object for which the property is checked. There are 3 events (Lines 6, 9 and 12) in the specification to be monitored at runtime. AspectJ [22, 23] the syntax is used for specifying these events. The first event (Line 9) captures all the calls to the *hasNext* method of the *Iterator* object that returns true. The second event (Line 9) captures all the calls to the *hasNext* method of the *Iterator* object that returns false. The last event (Line 12) captures all the calls to the *next* method of the *Iterator* object. A property specification follows these event specifications (Line 16), which starts with the *ltl* keyword. It defines a temporal property among the predefined set of events. In this example, a property is defined regarding the first and last events. Square brackets mean *always* and  $next \implies (*) hasnexttrue$  means that a *hasnexttrue* event must precede the *next* event. The last part of the specification (Line 18) defines the set of actions when this property is violated. In this example, just a simple message is printed on the console.



```

1 import java.io.*;
2 import java.util.*;
3
4 hasNext(Iterator i) {
5
6     event hasnexttrue after(Iterator i) returning(boolean b) :
7         call(* Iterator.hasNext())
8         && target(i) && condition(b) { }
9     event hasnextfalse after(Iterator i) returning(boolean b) :
10        call(* Iterator.hasNext())
11        && target(i) && condition(!b) { }
12    event next before(Iterator i) :
13        call(* Iterator.next())
14        && target(i) { }
15
16    ltl: [] (next => (*) hasnexttrue)
17
18    @violation { System.out.println("ltl violated!"); }
19
20 }

```

**Listing 2.2:** A sample JavaLTL Specification.

## CHAPTER III

### RELATED WORK

There exist several studies in the literature on automated generation of RV specifications [24, 25, 26, 27]. These studies utilize different types of artifacts as information sources for deriving and specifying system properties to be checked at runtime. In one of these studies [24], RV specifications are generated by analyzing software repositories. First, the use of coupled function calls is identified by mining these repositories. These functions, for example, can be related to the allocation and deallocation of system resources. Then, RV specifications are automatically generated to check the rules regarding the use of such functions. Our approach is not based on repository mining. We only use the latest version of the source code to apply static analysis.

In a recent study, UML models have been utilized to generate RV specifications in the form of finite state machines [25]. We do not rely on design models to generate RV specifications. We only make use of alerts that are generated by a SCAT and a library of predefined rules for alert types. Unlike UML models which are application specific, these rules are generic and they can be reused across different projects.

In another recent study, RV specifications were generated to complement theorem provers [26]. First, a deductive verification tool is used for analyzing the system. Then, RV specifications are automatically generated for cases that cannot be verified statically. Theorem provers are out of the scope of this thesis. We also do not aim at complementing static analysis. On the contrary, we especially focus on potential faults that are designated by SCAT alerts. Our goal is to either detect errors caused by these faults or prove the absence of them by RV.

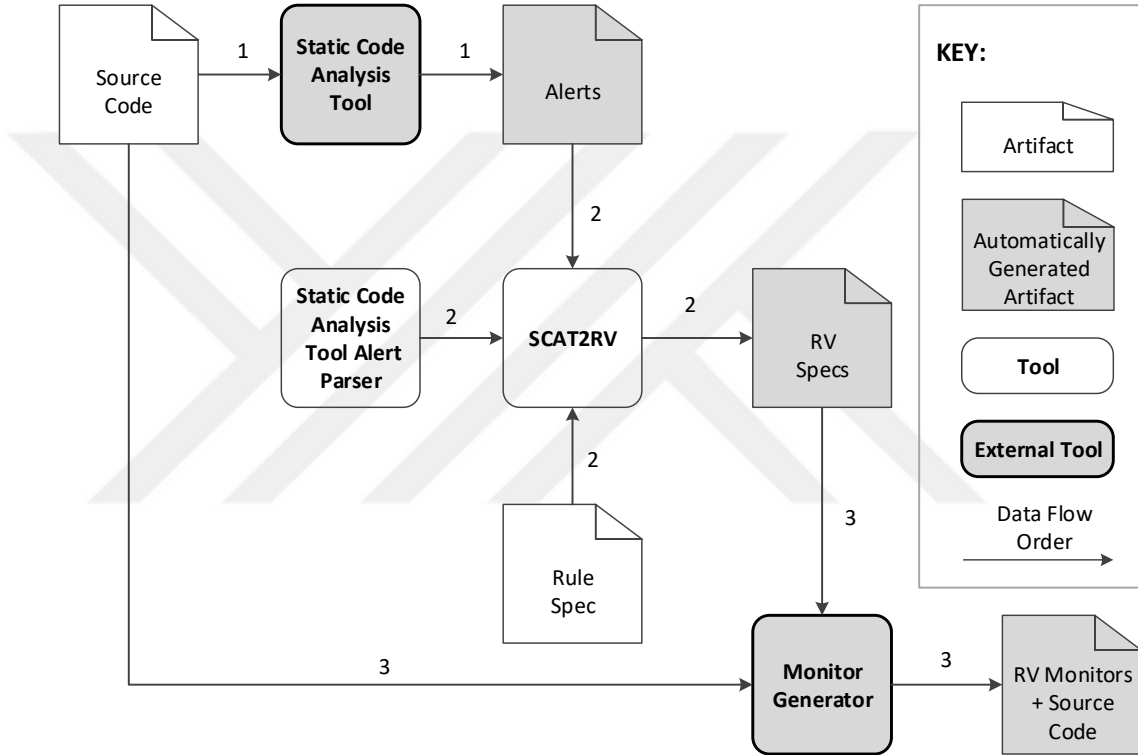
JNUKE VM [28] is a special virtual machine that enables the application of the

same analysis algorithms both statically and dynamically. The only difference between algorithms is that static analysis relies on abstract states instead of concrete states, which are available only at runtime. This enables the verification of static analysis results and elimination of false positives. In that sense, we share the same goal with JNUKE VM. However, we are not introducing a specialized environment to reach this goal. Our approach relies on automated transformations of input/output artifacts of existing tools. Essentially, we introduce a toolchain without changing these tools.

We have previously introduced a toolchain for generating RV specifications based on SCAT [2]. In that approach, a template specification for each alert type is defined once. Then, concrete specifications can be instantiated from this template for different instances of the same alert type [29]. In this work, we introduce a simple DSL for defining rules to be checked for an alert type. This contribution brings in two advantages. First, multiple rules can be defined for each alert type. Second, rule specification hides the underlying formalisms from the user, whereas template specification requires expertise on Linear Temporal Logic (LTL) [30] and the AspectJ language [23].

# CHAPTER IV

## OVERALL APPROACH



**Figure 4:** The overall approach.

The overall approach is depicted in Figure 4, which involves 3 main steps [31]. First, *Static Code Analysis Tool* takes the source code of a program as input, analyzes the source code and generates an alert list as output (1). Then, this alert list is provided to regarding SCAT2RV as input. SCAT2RV employs two more inputs. First, it uses a *Static Code Tool Alert Parser* to parse the alert list. This input is actually a plugin for SCAT2RV to be able to utilize alerts generated in various formats by various SCATs (See Appendix B for details.). The second input is *Rule Specifications*. This input

defines a set of rules to be checked for each alert type. *SCAT2RV* generates *Runtime Verification Specifications* according to these rules and the provided list of alerts (2). Finally, *Monitor Generator* tool takes these specifications and creates *Runtime Monitors* which are injected into the source code to track the runtime behavior of software system (3).

As depicted in Figure 4, the approach is fully automated. However, our approach relies on a set of predefined rule specifications, which is used for creating RV specifications. We introduced a simple DSL for this purpose, which will be explained in the next subsection. Our approach is extendable to support various SCATs. However, one needs to write a dedicated *Static Code Tool Alert Parser* as plugin to *SCAT2RV* in order to utilize a particular SCAT. Currently, we provide two such parser plugins, which are used for parsing alerts provided by FindBugs and PMD tools. We use JavaMOP as *Monitor Generator*.

In the following subsections, we explain each step of our approach in more detail. After that, in Chapter 5, we demonstrate the application of the approach in the context of two case studies. We developed the necessary tools in Java in order to integrate each step. However, our approach could also be implemented with different programming languages and environments.

#### ***4.1 Rule Specification for Alerts***

The first step of our approach involves the definition of rule specifications for alerts. We developed a DSL for this purpose. The grammar of this DSL is provided in Listing 4.1.

```

1 Rule ::= <ViolationResult>.<TemporalOp>.<Event>;
2 ViolationResult ::= Error | FalsePositive
3 TemporalOp ::= Never | Always | Eventually
4 Event ::= <BasicEvent>
5           | <BasicEvent>.<TemporalOrder>.<BasicEvent>
6 BasicEvent ::= FieldGet | FieldSet | MethodCall | <Exception>
7 TemporalOrder ::= Before | After
8 Exception ::= NullPointerException | SecurityException

```

**Listing 4.1:** Grammar of the DSL for rule specification.

Our DSL basically supports LTL to define one or more rules per alert type in the form of a temporal order among events. Hereby, a rule is composed of 3 parts separated by dots and finalized with a semi-colon. The first part identifies the type of rule depending on the result of its violation. There are two possible results reflected by the corresponding terminals (Line 2). The *Error* means the violation of the rule will indicate that an error is detected and this error is triggered by a fault reported on the particular alert. *FalsePositive* means the violation of the rule will indicate that a counter-example case is encountered at runtime, proving the particular alert to be false positive. For example, we can confirm the validity of a *null pointer* alert if a null pointer access occurs at runtime. If such a case does not take place, we can not claim that it will never happen. We can only confirm the existence of an error associated with this alert type, not the lack of it. On the other hand, *field never set* alerts can only be falsified. If the related field is ever set, the monitor can mark this alert as a false positive. If not, this time we can not conclude that alert points out an error. The field can be set during another execution of the program.

The second part of the rule specifies a *temporal operator* (i.e., TemporalOp), which can be the either one of *Never*, *Always*, or *Eventually* (Line 3). As implied

by their names, these terminals specify when an event is expected to occur. This event is specified as the last part of the rule. There are two kinds of *events* in our approach. First one is the basic event which is demonstrated with *BasicEvent* (Line 7). Another type of event is the composition of two basic events separated by dots and a *TemporalOrder*, which specifies a particular order between them (Line 6). Temporal Order is defined by terminals *Before or After* (Line 8). Temporal Order is used to establish temporal relativism between events. The set of possible basic events is defined as *FieldGet*, *FieldSet*, *MethodCall*, and *Exception* (Lines 6-7). *FieldGet* and *FieldSet* events take place when the field designated by the alert is accessed or modified, respectively. *MethodCall* event occurs when the method designated by the alert is called. Finally, an *Exception* event takes place when an exception occurs at runtime. The corresponding non-terminal can be replaced with the name of any subclass of the *java.lang.RuntimeException* class as the terminal value. Only two examples are shown in Listing 3 (Line 9), not to clutter the specification. Based on this grammar, for instance, one can specify a rule for the alert type *UwF: Unwritten field* as

```
FalsePositive.Never.FieldSet;
```

This alert type basically indicates that this field is never written. An instance of this alert type can be falsified if the corresponding field is ever set during execution. For this reason, we use *FalsePositive* as *ViolationResult*. Setting a field is a basic event because it does not have any dependency with respect to other events. Therefore, a *basic event* is used for this case. The event is setting a field. Overall, this rule will lead to a RV specification, which indicates that the corresponding field must never be set. In case a counterexample is observed at runtime, where the field is set, the alert is deemed to be false positive.

SCAT2RV generates RV specifications based on such rules as explained in the following section.

## 4.2 *Generation of Runtime Verification Specifications*

Once the list of rules is defined, it can be used by SCAT2RV<sup>1</sup> to automatically generate RV specifications for a given list of SCAT alerts. Our approach is agnostic to the SCAT that generates alerts. We developed a plugin interface for SCAT2RV, which includes basic operations to parse the alert output (See Appendix B for details). We already developed parsers for FindBugs and PMD tools. Developers can further extend this scope by writing new parsers, which implement our interface. The set of defined rules must be associated with types of alerts for various SCATs. Table 3 lists the rules that we defined for some of the alert types generated by FindBugs and PMD.

In the background, SCAT2RV employs templates for possible events, which are composed and instantiated to generate a RV specification based on an alert instance. For instance, *field get* and *call* events are predefined in the following format for event specification.

```
get(@FIELDINFO); call(@METHODINFO);
```

Hereby, parameters such as *@FIELDINFO* and *@METHODINFO* are replaced with actual field and method names, which are obtained by parsing the given alert instance. The part of the RV specification that defines a temporal ordering among the events is generated according to the rule definition that is associated with the type of the given alert. SCAT2RV outputs an error message in case the type of the alert cannot be associated with a rule specification. In the following, we explain the generation of runtime monitors based on the generated RV specifications.

---

<sup>1</sup>The source code of the tool is available at <https://github.com/yunuskilicdev/Saida>



**Table 3:** A sample list of alert types supported by FindBugs and PMD together with the corresponding rule specifications.

<b>Tool</b>	<b>Alert</b>	<b>Rule Specification</b>
FindBugs	BC: Impossible cast	Error.Always.ClassCastException
FindBugs	BC: Impossible downcast	Error.Always.ClassCastException
FindBugs	NP: Null pointer dereference	Error.Always.FieldGet.Before.FieldSet
FindBugs	UwF: Unwritten public or protected field	FalsePositive.Never.FieldSet
FindBugs	NP: Method does not check for null argument	Error.Always.NullPointerException
PMD	ImmutableField	FalsePositive.Never.FieldSet.After.MethodCall
PMD	UnusedPrivateField	Error.Never.FieldGet
PMD	UnusedLocalVariable	Error.Never.FieldGet
PMD	UnusedPrivateMethod	Error.Never.MethodCall

### ***4.3 Generation of Runtime Monitors***

SCAT2RV generates RV specifications that conform to JavaLTL as explained in Chapter 2. Then, it employs the JavaMOP tool in the background for converting this specification to AspectJ code. The final step is to introduce the generated AspectJ code on top of the base code of the subject system. Then, aspects will start the monitoring tasks when the system is executed and they will output notifications regarding detected errors or identified false positive alerts.



# CHAPTER V

## EVALUATION

In this chapter, we present an evaluation of our approach on two subject systems, which are introduced in the next section. Then, a sample SCAT alert for one of these systems will be used for illustrating the end-to-end application of the whole approach. After that, the same rule will be used for applying RV on the other subject system to illustrate the reuse of rules across various projects. Finally, another SCAT will be used on the same subject system to illustrate the reuse of rules across various tools.

### *5.1 Subject Systems and Tools*

An overview of the subject systems is provided in Table 4. The first subject system is JBook<sup>1</sup>, which is an open source project developed with Java. JBook lets users retrieve, read, and bookmark electronic texts. We used version 1.4 of this tool, which has 1.2K lines of code. The second subject system is JDom<sup>2</sup>, which is used for accessing, manipulating, and outputting XML data from Java code. It contains around 8.4K lines of Java code. Both of these systems are part of a benchmark suite that is used for evaluating SCATs [8]. In our evaluation, we used FindBugs and PMD as SCATs that are explained in Section 2.1.

---

<sup>1</sup><http://jbook.sourceforge.net/>

<sup>2</sup><http://www.jdom.org/>

**Table 4:** Subject software systems.

	<b>Version</b>	<b>License</b>	<b>Lines of Code</b>
<b>JBook</b>	1.4	GNU GPL	1276
<b>Jdom</b>	1.1	Apache-style	8422

```

1 public class Display extends ... {
2   ...
3   String strFontName;
4   public Display(...) {
5       this.state = state;
6       //this.strFontName = ...;
7       ...
8   }
9 }
10
11 public class JBook extends JFrame {
12   ...
13   public JBook() {
14       ...
15       initialize(display);
16   }
17
18   private static initialize(Display display) {
19       ...
20       Field strFontNameField = display.getClass().getDeclaredField
21           ("strFontName");
22       strFontNameField.set(display, "False Positive");
23   }

```

**Listing 5.1:** A code snippet from the *Display* class in *JBook*.

## 5.2 Rule Specification

In this section, the application of the rule specification step is explained. Consider a code snippet from the JBook source code as shown in Listing 5.1. Hereby, there is a member variable, *strFontName* of the *Display* class, of type *String*. This member variable is normally initialized in the constructor (Line 6). We commented out this line and initialized the variable externally, within the *JBook* class (Lines 20-21) by means of reflection. Findbugs currently cannot detect such indirect dependencies although there have been recently proposed approaches [32, 33] to address this problem. It reports an alert for the *strFontName* variable. The type of the alert is *UwF: Unwritten field*, which is described as “*This field is never written. All reads of it will return the default value*”<sup>3</sup>. Such alerts can be exported from Findbugs in an XML format as shown in Listing 5.2.

```
1 <BugInstance type="UWF_UNWRITTEN_FIELD" priority="2" category
   ="CORRECTNESS" ...>
2 ...
3 <Field classname="org.jbook.source.Display"
4     name="strFontName" signature="Ljava/lang/String;"
5     isStatic="false">
6     ...
7 </Field>
8 </BugInstance>
```

**Listing 5.2:** The exported alert information for the example case.

In the previous chapter, we introduced a rule for this alert type as “*FalsePositive.Never.FieldSet;*”. If the program execution is monitored at runtime according to this rule, a violation of the rule can be detected. This violation would indicate that

---

<sup>3</sup><http://findbugs.sourceforge.net/bugDescriptions.html>

the field is indeed written and the alert is a false positive. In the following, we discuss the generation of RV specifications based on such rule definitions.

### 5.3 *Runtime Verification Specification Generation*

SCAT2RV converts SCAT alerts to RV specifications according to the rule definitions for the corresponding alert type. Listing 5.3 shows the RV specification that is generated for the alert in Listing 5.2 according to the rule introduced as an example. The specification is composed of 3 parts. The first part (Lines 4-5) specifies the set of events that should be monitored. The specification is generated according to the AspectJ pointcut syntax [23] and it captures points of execution after a field is modified. The name and the location of the particular field to be monitored is obtained from the reported alert. The second part (Line 6) specifies the temporal rule in JavaMop LTL Syntax [13]. This rule indicates that the event defined in the first part of the specification must never take place. The last part (Line 7) specifies what to perform when a violation of the rule is detected at runtime. Hereby, the corresponding alert is reported as a false positive.

```
1 UWF_UNWRITTEN_FIELD_R1_A1(){
2     event UWF_UNWRITTEN_FIELD after() :
3         set(String org.jbook.source.Display.strFontName){}
4     ltl: []!UWF_UNWRITTEN_FIELD
5     @violation{System.out.println("UWF_UNWRITTEN_FIELD bug
6         reported for org.jbook.source.Display.strFontName is
7         false positive!");}
8 }
```

**Listing 5.3:** Automatically generated JavaMOP specification based on the example rule specification for falsifying an alert of type *UwF: Unwritten field*.

In the following, we discuss the synthesis and execution of runtime monitors based on the generated RV specification.

#### ***5.4 Runtime Verification Monitor Generation***

We rely on the JavaMOP tool [13] for the synthesis of runtime monitors and the integration of these monitors with the system. JavaMOP automatically creates the AspectJ code based on an RV specification that SCAT2RV generated as listed in Listing 5.3.

The generated AspectJ code facilitates the weaving of online monitoring code into the system at compile time. For example, we have included the generated code as part of the JBook project files, recompiled the system and run it. The monitored variable was initialized and this event was captured by the monitor. As a result, we observed the console output as specified in Line 5 of Listing 5.3.

Our approach enables a transparent integration of SCATs and RV tools. One can specify just a line of a simple rule regarding an alert type to enable the generation and execution of monitors for all the alerts of this type. One can also specify multiple rules per alert type. Moreover, these rules are generic and they can be reused across projects. In the following section, we show the reuse of the rule defined in this section for another software system to generate RV monitors.

#### ***5.5 Reuse of Rules Across Projects***

In this section, we replicate our study for the second subject system, JDom. We focus on the same case, where a variable is claimed to be never written. A runtime monitor is generated by reusing the same rule defined in the previous section. Hence, we show that rules can be used across projects and monitors for the corresponding alert types can be generated without any manual effort.

```

1 public class ProcessingInstruction extends Content ... {
2     ...
3     public ProcessingInstruction setData(String data) {
4         String reason = Verifier.checkProcessingInstructionData(
5             data);
6         if (reason != null) {
7             throw new IllegalArgumentException(data, reason);
8         }
9         //this.rawData = data;
10        this.mapData = parseData(data);
11        return this;
12    }
13    ...
14    Field pInsField = processingInstruction.getClass().
15        getDeclaredField("rawData");
16    pInsField.set(processingInstruction, "Setted");
17    ...
18 }

```

**Listing 5.4:** A code snippet from the *ProcessingInstruction* class in *JDom*.

A code snippet from the *JDom* source code is shown in Listing 5.4. Hereby, there is a member variable, *rawData* of the *ProcessingInstruction* class, of type *String*. This variable is initialized by means of reflection. However, this can not be detected with static code analysis. Hence, Findbugs reports an alert of type *UwF: Unwritten field* for the variable. The reported alert is shown in Listing 5.5. We did not specify any rule for this case study. We utilized the same rule that was defined for the alert type *UwF: Unwritten field* as “*FalsePositive.Never.FieldSet;*”.



```

1 <BugInstance type="UWF_UNWRITTEN_FIELD" priority="2" category
   = "CORRECTNESS" ...>
2 ...
3 <Field classname="org.jdom.ProcessingInstruction"
4   name="rawData" signature="Ljava/lang/String;"
5   isStatic="false">
6   ...
7 </Field>
8 </BugInstance>

```

**Listing 5.5:** The exported alert information for the *rawData* variable.

Although the rule is the same for the alert type, the generated monitors are specific to the subject system and the particular instances of the alert type. SCAT2RV derives the relevant context of the reported alert (See Listing 5.5) and generates a RV specification accordingly. Listing 5.6 shows the specification generated for the alert reported for the JDom system. We can see that the structure of the specification is the same as the one generated for JBook system (See Listing 5.3). However, the monitored objects and variables are different.

```

1 UWF_UNWRITTEN_FIELD_R1_A1(){
2   event UWF_UNWRITTEN_FIELD after() :
3   set(String org.jdom.ProcessingInstruction.rawData){}
4   lt1: []!UWF_UNWRITTEN_FIELD
5   @violation{System.out.println("UWF_UNWRITTEN_FIELD bug
   reported for org.jdom.ProcessingInstruction.rawData is
   false positive!");}
6 }

```

**Listing 5.6:** Automatically generated JavaMOP specification for the JDom system.

We have included the generated AspectJ code by JavaMOP as part of the JDom project files, recompiled the system and run it. As a result, we have observed the console output as specified in Line 5 of Listing 5.6.

## 5.6 Reuse of Rules Across Tools

In this section, we discuss the use of our approach with PMD, as an alternative SCAT for FindBugs. Some of the alert types in these SCATs are representatives of the same bug types. We used the alert type *UwF: Unwritten field* to demonstrate our approach with FindBugs. The alert type named *UnusedLocalVariable* is concerned with the same issue in PMD. An instance of this alert type is shown in Listing 5.7 as reported by PMD for the JBook system. A list of such alerts can be obtained in the form of an XML document, which can be parsed with the parser plugin we developed for PMD.

```
1 <?xml version="1.0"?>
2   <pmd>
3     <file name="c:\data\pmd\pmd\org\jbook\source\Display.
4       java">
5       <violation line="5" rule="UnusedLocalVariable">
6         Avoid unused local variables such as 'strFontName'
7       </violation>
8     </file>
9   </pmd>
```

**Listing 5.7:** A sample alert generated by PMD.

We only needed to associate the same rule with the alert type in PMD to replicate the approach with this SCAT. We observed the same results.

## CHAPTER VI

### RESULTS AND DISCUSSION

Not all the static code analysis alerts are relevant for RV. For instance, many types of alerts that are reported by FindBugs are categorized as *bad practice*. There is nothing to check at runtime for these alert types, which might be related to styling issues. However, there also exist alert types that are highly relevant for RV. For instance, concurrency bugs lead to failures depending on the runtime context and the scheduling performed by the operating system. Hence, alerts that are categorized as *multithreaded correctness* point at potential errors that can be monitored at runtime. On the other hand, instances of some of the alert types can be deemed false positive by RV. The list of relevant alert types that are processed by SCAT2RV can be seen at Table 3.

The subject systems JBook and JDom that we used in our evaluation are part of a benchmark suite [34], which contains 6 software systems in total. We collected and analyzed all the alerts reported for these systems. We selected those that are pointing at potential faults rather than styling issues or violations of coding conventions. Then, we checked which of those are relevant for RV and can be processed by SCAT2RV to automatically generate runtime monitors. The overall results are listed in Table 5. For example, JBook has 52 alerts. 19 of them are pointing at potential faults. For 53% of these 19 alerts, that is, for 10 alerts SCAT2RV can generate runtime monitors automatically.

One may utilize exception handling to detect and tolerate some of the reported bug instances. For instance, a specific check or exception handler can be simply added to the source code for eliminating accesses to null references as pointed out by an alert.

**Table 5:** Ratio of alerts for which RV specifications can be automatically generated.

	Total # of alerts	# of relevant alerts pointing at potential faults	# of alerts applicable for <b>SCAT2RV</b>	Ratio of relevant alerts applicable for <b>SCAT2RV</b>
<b>JBook</b>	52	19	10	53%
<b>JDom</b>	55	18	8	44%
<b>CsvObject</b>	7	0	Not Defined	Not Defined
<b>ImportScrubber</b>	35	12	3	25%
<b>iTrust</b>	110	29	1	3%
<b>org.eclipse.core.runtime</b>	98	28	7	25%
<b>Overall</b>	357	106	29	27%

This approach has 3 drawbacks. First, the types of checks are limited by the set of defined exception types. Second, it requires time and effort to manually analyze all the alerts and modify the corresponding parts of the source code. Third, some of the bug types cannot be localized in a single module of the program. The corresponding erroneous scenario can involve a series of events that are related to multiple classes. Our approach addresses all of these drawbacks. A set of reusable rules can be utilized for automatically generating RV monitors, regardless of the location/distribution of the corresponding modules.

Our study is subject to a set of validity threats [35]. To mitigate *external validity* threats we performed a case study with two subject systems and analyzed 6 subject systems in total. We showed that the approach is feasible and viable for utilizing static analysis to focus RV and automatically generate RV specifications. We mitigated the *construct validity* threat by implementing an instance of the approach. This instance represents a proof-of-concept implementation for showing the viability of our approach. There exist a *reliability* threat due to the variations of runtime behavior of the subject system based on usage scenarios. The full replication of the study requires the application of the same scenarios. Considering our case studies, any scenario that leads to the modification of the monitored variables is enough for replication. *Internal validity* threats are mitigated since we created a toolchain without changing the implementation or any parameters of the externally utilized tools. However, not all the runtime factors can be controlled, in case they have an impact on the results.

## CHAPTER VII

### CONCLUSIONS AND FUTURE WORK

In this thesis, we introduced a new approach and a tool for automatically generating runtime monitors based on a list of alerts which are reported by different static code analysis tools. In order to achieve this aim, we introduced a simple domain specific language for defining rules to be checked for each alert type. Formal verification specifications are automatically generated for each reported alert instance based on the set of predefined rules for the corresponding alert type. Then runtime monitors are automatically synthesized and integrated into the system. These integrated monitors continuously report false positive alerts during software execution or they report detected errors together with the diagnosis information obtained from the corresponding alert. The approach was applied to two different open source software systems. We observed that static code analysis alerts can be proved to be false positives and/or identified errors can be checked at runtime by means of automatically generated runtime monitors. We also showed that the defined rules can be reused across projects. Furthermore, two different static code analysis tools alerts were used to demonstrate how our system works with multiple tools.

As future work, additional rules can be defined regarding various SCAT alert types that are relevant for RV. The rule specification language might also be improved if its expressiveness turns out to be insufficient for defining new rules.

## APPENDIX A

### JAVALTTL SYNTAX

The MOP LTL plugin syntax instantiates the generic  $\langle$ Logic Name $\rangle$ ,  $\langle$ Logic Syntax $\rangle$ , and  $\langle$ Logic State $\rangle$  from the Logic Repository Syntax. It is used in conjunction with the  $\langle$ Logic Repository I/O $\rangle$  syntax and defined using Backus Normal Form (BNF) [36]. LTL syntax is the base element of language. Multiple LTL syntax elements can be connected with various operators like *and*, *or*, *xor*. Another important element is the *event*. You can define events like below.

```
 $\langle$ Event $\rangle ::= ["creation"] \text{ "event" } \langle Id \rangle \langle \text{AspectJ advice} \rangle$   
 $\text{"{" } \langle \text{Java Statements} \rangle \text{"}"}$ 
```

An event is basically defined by a name and the corresponding AspectJ advice. A set of defined *Java Statements* will be executed when the defined event is triggered. The *violation* term is used to indicate that they will be committed in case of violation of the defined rules.

```

1 // BNF below is extended with {p} for zero or more and [p] for
   zero or one repetitions of p
2 // The mandatory MOP logic syntax
3 <LTL Name>      ::= "ltl"
4 <LTL Syntax>    ::= "true" | "false"
5                 | <Event Name>
6                 | <Not> <LTL Syntax>
7                 | <LTL Syntax> "and" <LTL Syntax>
8                 | <LTL Syntax> "or" <LTL Syntax>
9                 | <LTL Syntax> "xor" <LTL Syntax>
10                | <LTL Syntax> "=>" <LTL Syntax>
11                | <LTL Syntax> "<=>" <LTL Syntax>
12                | "[" <LTL Syntax>
13                | "<" <LTL Syntax>
14                | "o" <LTL Syntax>
15                | <LTL Syntax> "U" <LTL Syntax>
16                | <LTL Syntax> "~U" <LTL Syntax>
17                | <LTL Syntax> "R" <LTL Syntax>
18                | "<*" <LTL Syntax>
19                | "(*)" <LTL Syntax>
20                | <LTL Syntax> "S" <LTL Syntax>
21                | <LTL Syntax> "~S" <LTL Syntax>
22 <LTL State>    ::= "violation"

```

**Listing A.1:** JavaLTL Syntax.



## APPENDIX B

### SCAT2RV TOOL PLUGIN INTERFACE

SCAT2RV is developed with Java. An interface mechanism (See Listing B.1) was used to handle various SCATs with various output formats. A parser that conforms to this interface must be developed to parse output of a particular SCAT. The set of variables and/or methods that are related to the reported alerts must be extracted from SCAT output. A parser must return a list of *BugInstance* object. The parser might need to perform additional code analysis to populate these objects with the necessary information. For instance, output of PMD does not include name of the class/package and signature of the method that is related to the reported alert. It provides the path of the related class instead. So, our PMD parser performs an additional analysis to retrieve the necessary information based on this path.

```

1 public abstract class ToolBase {
2     private String inputPath;
3
4     public ToolBase(String inputPath){
5         setInputPath(inputPath);
6     }
7     public void Create() throws IOException {
8         List<BugInstance> alerts = ParseAlerts();
9         ...
10    }
11
12    private String FindMopPath(String alertType) {
13        String path = String.format("%s\\%s.mop", OutputFolder,
14            alertType);
15        File file = new File(path);
16        if (file.exists())
17            return path;
18        return StringUtils.EMPTY;
19    }
20
21    //Main method to parse alerts
22    protected abstract List<BugInstance> ParseAlerts();
23
24    ...
25 }

```

**Listing B.1:** SCAT parser interface.

## APPENDIX C

### SCAT2RV TOOL USER MANUAL

SCAT2RV is a command line tool, lacking a graphical user interface. Input specifications must be provided to be able to use the tool. First of all, each rule must be specified within a text file, whose name is exactly the same as the corresponding alert type. For example, a file name can be *UWF\_UNWRITTEN\_FIELD.txt*. The content of this file can be as follows.

```
FalsePositive.Never.FieldSet;
```

These files should be saved into a folder in the file system. The user needs to provide the full path of this folder to SCAT2RV because it will search for all the rule specifications here and find the relevant specification whose name matches with alert type of a given alert output.

SCAT2RV is executed with several command line arguments. These arguments are explained in Table 6. A sample run of the tool with these arguments set is shown below.

```
java -jar SCAT2RV.jar -rules C:\rules\ -tool FindBugs  
-alerts C:\exampleSCAToutput.xml -src C:\JBook\src
```

The above command generates an AspectJ file as output. This file can be accompanied with the base source code of the system to enable monitoring.

**Table 6:** The list of SCAT2RV command line arguments.

<b>Argument</b>	<b>Description</b>
-rules	Path of the folder that contains rule specifications.
-alerts	Path of the SCAT alert list file.
-tool	SCAT parser will be used (FindBugs or PMD).
-src	Path of the source code of the software.



## References

- [1] B. Chess and G. McGraw, “Static analysis for security,” *IEEE Computer Society*, vol. 2, no. 6, pp. 76–79, 2004.
- [2] H. Sozer, “Integrated static code analysis and runtime verification,” *Software: Practice and Experience*, vol. 45, no. 10, pp. 1359–1373, 2015.
- [3] T. Delev and D. Gjorgjevikj, “Static analysis of source code written by novice programmers,” in *2017 IEEE Global Engineering Education Conference (EDUCON)*, pp. 825–830, April 2017.
- [4] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, “Why don’t software developers use static analysis tools to find bugs?,” in *Proceedings of the 35th International Conference on Software Engineering*, pp. 672–681, 2013.
- [5] U. Yuksel and H. Sozer, “Automated classification of static code analysis alerts: A case study,” in *Proceedings of the 29th IEEE International Conference on Software Maintenance*, (Eindhoven, The Netherlands), pp. 532–535, 2013.
- [6] R. Krishnan, M. Nadworny, and N. Bharill, “Static analysis tools for security checking in code at motorola,” *ACM SIG Ada Letters*, vol. 28, no. 1, pp. 76–82, 2008.
- [7] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. Hudepohl, and M. Vouk, “On the value of static analysis for fault detection in software,” *IEEE Transactions on Software Engineering*, vol. 32, no. 4, pp. 240–253, 2006.
- [8] S. Heckman and L. Williams, “A systematic literature review of actionable alert identification techniques for automated static code analysis,” *Information and Software Technology*, vol. 53, no. 4, pp. 363–387, 2011.
- [9] T. Kremenek and D. Engler, *Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations*, pp. 295–315. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003.
- [10] N. Delgado, A. Q. Gates, and S. Roach, “A taxonomy and catalog of runtime software-fault monitoring tools,” *IEEE Transactions on Software Engineering*, vol. 30, pp. 859–872, Dec 2004.
- [11] M. Leucker and C. Schallhart, “A brief account of runtime verification,” *Journal of Logic and Algebraic Programming*, vol. 78, pp. 293–303, 2008.
- [12] D. Jin, P. O. Meredith, C. Lee, and G. RoÅşu, “Javamop: Efficient parametric runtime monitoring framework,” in *2012 34th International Conference on Software Engineering (ICSE)*, pp. 1427–1430, June 2012.

- [13] D. Jin, P. Meredith, C. Lee, and G. Rosu, “JavaMOP: Efficient parametric runtime monitoring framework,” in *Proceedings of the 34th International Conference on Software Engineering*, (Zurich, Switzerland), pp. 1427–1430, 2012.
- [14] J. Novak, A. Krajnc, and R. Ajontar, “Taxonomy of static code analysis tools,” in *The 33rd International Convention MIPRO*, pp. 418–422, May 2010.
- [15] “FindBugs official website,” 2017. [online] <http://findbugs.sourceforge.net>.
- [16] D. Hovemeyer and W. Pugh, “Finding bugs is easy,” *SIGPLAN Not.*, vol. 39, pp. 92–106, Dec. 2004.
- [17] “PMD official website,” 2017. [online] <https://pmd.github.io/>.
- [18] “PMD official rulesets,” 2017. [online] <https://pmd.github.io/pmd-5.8.0/pmd-java/rules/index.html>.
- [19] M. Leucker and C. Schallhart, “A brief account of runtime verification,” 2008.
- [20] “Monitoring-oriented programming,” 2017. [online] <http://fsl.cs.illinois.edu/index.php/MOP>.
- [21] “Linear temporal logic,” 2017. [online] <http://www.cs.colostate.edu/france/CS614/Slides/Ch5-Summary.pdf>.
- [22] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, “Aspect-oriented programming,” in *Proceedings of the European Conference on Object-Oriented Programming*, (Paris, France), pp. 220–242, 1987.
- [23] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, “An overview of AspectJ,” in *Proceedings of the European Conference on Object-Oriented Programming*, pp. 327–353, 2001.
- [24] B. Livshits and T. Zimmerman, “Dynamine: Finding common error patterns by mining software revision histories,” *SIGSOFT Software Engineering Notes*, vol. 30, pp. 296–305, 2005.
- [25] S. Ciraci, H. Sozer, and B. Tekinerdogan, “An approach for detecting inconsistencies between behavioral models of the software architecture and the code,” in *Proceedings of the 36th International Conference on Computer Software and Applications*, (Izmir, Turkey), pp. 257–266, 2012.
- [26] W. Ahrendt, G. Pace, and G. Schneider, “A unified approach for static and runtime verification: Framework and applications,” in *Proceedings of the International Symposium on Leveraging Applications of Formal Methods*, pp. 312–326, 2012.

- [27] S. Ciraci, H. Sozer, and B. Tekinerdogan, “A runtime verification framework for smart grid applications implemented on simulation frameworks,” in *Proceedings of the Workshop on Software Engineering Challenges for the Smart Grid*, (San Francisco, CA, USA), pp. 1–8, 2013.
- [28] C. Artho and A. Bierel, “Combined static and dynamic analysis,” in *Proceedings of the International Workshop on Abstract Interpretation of Object-Oriented Languages*, (Paris, France), pp. 98–115, 2005.
- [29] K. Czarnecki and S. Helsen, “Feature-based survey of model transformation approaches,” *IBM Systems Journal*, vol. 45, pp. 621–645, 2006.
- [30] D. Gabbay, I. Hodkinson, and M. Reynolds, *Temporal Logic*. Oxford, UK: Oxford University Press, 1997.
- [31] Y. Kiliç and H. Sözer, “Generating runtime verification specifications based on static code analysis alerts,” in *Proceedings of the Symposium on Applied Computing, SAC '17*, (New York, NY, USA), pp. 1342–1347, ACM, 2017.
- [32] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini, “Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders,” in *Proceedings of the 33rd International Conference on Software Engineering*, pp. 241–250, 2011.
- [33] Y. Li, T. Tan, Y. Sui, and J. Xue, “Self-inferencing reflection resolution for java,” in *Proceedings of the 28th European Conference on Object-Oriented Programming*, pp. 27–53, 2014.
- [34] S. Heckman and L. Williams, “On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques,” in *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '08*, (New York, NY, USA), pp. 41–50, ACM, 2008.
- [35] C. Wohlin, P. Runeson, M. Host, M. Ohlsson, B. Regnell, and A. Wesslen, *Experimentation in Software Engineering*. Berlin, Heidelberg: Springer-Verlag, 2012.
- [36] J. W. Backus, “The syntax and semantics of the proposed international algebraic language of the zurich acm-gamm conference,” in *Proceedings of the International Conference on Information Processing*, 1959.