Symbolic Execution for Runtime Error Detection and Investigation of Refactoring Activities Based on a New Dataset

István Kádár

Department of Software Engineering
University of Szeged

Szeged, 2017

Supervisor:
Dr. Rudolf Ferenc

A THESIS SUBMITTED FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
OF THE UNIVERSITY OF SZEGED

University of Szeged
PhD School in Computer Science

"The greater danger for most of us is not that we aim too high and we miss it, but we aim too low and reach it."

— Michelangelo

Preface

I am writing these lines in the exact same room where I switched on my very first computer for the first time. I used it to play computer games of course. I could never have imagined that some day I would write a PhD dissertation and might become a computer scientist. The way was full of challenges. Learning the programmer profession is not easy at all, but I was all in. I truly wanted to become an excellent programmer. I started working at the Department of Software Engineering, University of Szeged as a developer in 2010. Soon, my mentors, Dr. Tibor Bakota and Dr. Rudolf Ferenc gave me the opportunity to work on a research related project. I was very excited, but I still did not think that I would become a researcher at some point. As time passed, I set bigger and bigger goals in research. One of them was the PhD degree. Looking back on the last seven years, I have learnt that it is worth setting big goals and Michelangelo's philosophy still holds true: *"The greater danger for most of us is not that we aim too high and we miss it, but we aim too low and reach it."*

Of course, none of the presented research work could have been realized without the help of others. Therefore I would like to thank all of my colleagues, teachers, mentors and friends for their help in getting me here. First of all, I would like to thank my supervisor, Dr. Rudolf Ferenc for guiding me through my studies. He motivated me with his positive attitude to programming and research-oriented thinking. Without him I would probably have never even got involved in scientific research. I could not have asked for a better mentor. My special thanks goes to Dr. Péter Hegedűs, whom I consider my second mentor and supervisor. He always helped me when I turned to him. I am grateful for his continuous support, guidance and encouragement. I would also like to express my sincerest gratitude to Dr. Tibor Gyimóthy, the head of the Software Engineering Department, for supporting my research work. Many thanks go to Dr. Tibor Bakota, from whom I learned a lot, and to other colleagues as well, namely Dr. Judit Jász and Tamás Perényi. I would like to thank the anonymous reviewers of my papers for their useful comments and suggestions as well. And I would like to express my thanks to Andrea Kintses-Fejér for reviewing and correcting this work from a linguistic point of view.

Last, but not least, I wish to express my gratitude to my parents for providing a supporting background conducive to my studies and also for encouraging me to go on with my research.

István Kádár, oktober 2017

Contents

List of Tables

List of Figures

Listings

To my Parents

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."

— Martin Fowler

1 Introduction

It is a big challenge in software engineering to produce huge, reliable and robust software systems. Nowadays, most computer programs are not written in machine code or assembly, which tends to be very error prone, but using higher level languages like Java, JavaScript or C++. Despite that, the programs written in these languages are easier to write and understand, they do not guarantee neither a bug-free, nor a good-quality system, because the degree of complexity and the number of possibilities that a programer needs to keep track is still huge. Human beings make mistakes, even if they are programmers. Moreover, in industry, developers typically have to focus on solving the problem quickly. The importance of code quality in time pressure is frequently secondary. On the other hand, software code quality is very important, because a too complex, hard-to-maintain code results in more bugs, and makes the further development more expensive. The research work behind this thesis is inspired by the wish to develop high quality software systems in industry in a more effective and easier way, making the lives of customers and eventually end-users more comfortable and more effective.

The thesis consists of two main topics: *the utilization of symbolic execution for runtime error detection* and *the investigation of practical refactoring activities.* Both topics address the area of program source code quality.

Symbolic execution is a program analysis technique which explores the possible execution paths of a program by handling the inputs as unknown variables (symbolic variables). It simulates the execution of the program on these symbolic variables. If there is a branching statement in the code where it cannot be decided on which branch to continue on because of the aforementioned unknown input, the execution will continue on both possible paths. The main usages of symbolic execution are generating inputs of program failure and high-coverage test case generation. It is also used for finding serious code smells and bugs, which are hard to detect with other static source code analysis techniques or with testing. Symbolic execution is able to expose defects that would be very difficult and timeconsuming to find through manual testing, and would be exponentially more costly to fix if they left undetected until runtime.

In this work we focus on runtime error detection (such as null pointer dereference,

bad array indexing, division by zero, etc.) by discovering critical execution paths in Java programs. By detecting these critical source code points, we are helping programmers develop better quality software and we are also supporting the debugging of such errors by providing the execution path leading to the dangerous code part. One of the greater challenges in symbolic execution is the huge number of possible execution paths, which increase exponentially. Our research proposes approaches to the handling the aforementioned problem of path explosion by applying symbolic execution at the level of methods. We also investigated the limitations of this state space together with the development of efficient search heuristics. To make the detection of runtime errors more accurate, we propose a novel algorithm that keeps track of the conditions above symbolic variables during the analysis.

Source code refactoring is a popular and powerful technique for improving the internal structure of software systems. With refactoring, developers eliminate code smells and improve the maintainability of the code, making further developments more efficient. The concept of refactoring was introduced by Martin Fowler. He originally proposed that code smells should be the primary technique for identifying refactoring opportunities in the code. However, we lack empirical research results on how, when and why refactoring is used in everyday software development, what are its effects on short and long-term maintainability and costs. By getting answers to these questions, we could understand how developers refactor code in practice, which would help propose new methods and tools for them that are aligned with their current habits leading to more effective software engineering methodologies in the industry. To help the further empirical investigations of code refactoring, we proposed a publicly available refactoring dataset. The dataset consists of refactorings and source code metrics of open-source Java systems. We applied the dataset for an analysis on the effects of code refactoring on source code metrics and maintainability, which are primary quality attributes in software development.

The thesis has two main parts based on the two research areas we worked on.

The **first part** of the thesis is about advances in symbolic execution aiming runtime error detection and consists of four chapters. Chapter 2 provides an overview about symbolic execution, after that each chapters particularizes one thesis point. In Chapter 3 we describe our method-level symbolic execution technique for runtime error detection in Java programs. Then, Chapter 4 introduces a special constraint building method that makes the error detection more accurate with our symbolic execution engine. Finally, in Chapter 5 we show how we analyze the state space of symbolic execution and how we optimize the traversal of the possible execution paths.

The **second part** contains one chapter (Chapter 6), which stands for the fourth thesis point. Here we show the connection between the refactoring activity and the maintainability with other source code metrics, and introduce the refactoring dataset that our analysis was based on.

Chapter 7 concludes the thesis. In the Appendix we summarize our results in English and Hungarian, listing the thesis points and the contributions of the author.

Part I

Advances in Symbolic Execution for Runtime Error Detection

"Most people overestimate what they can do in one year and underestimate what they can do in ten years."

— Bill Gates

2

Background of Symbolic Execution

Program analysis is the process of analyzing the correctness and other quality properties of software systems. Besides dynamic program analysis where we analyze the program during runtime, there are static program analysis techniques, which are performed without executing the program. In most cases, static analysis is performed on some version of the source code, thus can also be used to find code smells and anti patterns in the source that influence the inner quality of the system.

The most simplified static source code analysis technique is when we tokenize the code and find patterns in the linear token sequence. On the next level, we do not merely tokenize the input source, but we build an Abstract Syntax Tree (AST), which represents the structure of the code, allowing the identification of more complex code smells more precisely. A well-known tool that uses pattern matching on AST is the PMD Java source code analyzer [73].

In the next step, control flow analysis [2] is a technique for determining the control flow of a program. The knowledge of control flow is required by any static, global analysis of the expression and data relationships and is embedded in many compilers for optimization.

On the other hand, data flow analysis [1] tracks the propagation of data values and the legality of data at multiple points in the code. With data flow analysis we are able to detect more serious rule violations and bugs like the usage of uninitialized or invalid memory, potential runtime errors and security vulnerabilities (e.g. SQL injection). For example, the tools FindBugs [43] and Cppcheck [22] use data flow analysis techniques to find such coding rule violations.

Symbolic Execution can be considered a static analysis technique that not only propagates data values, but does it on the possible execution paths through the abstract states of the program while keeping track of the conditions that have to be fulfilled on the different paths. In this chapter, we give a detailed background of the symbolic execution program analysis technique. Firstly, we describe the theoretical background of symbolic execution in Section 2.1. After that, we present existing symbolic execution implementations and their applications in Section 2.2. Finally, in Section 2.3 we describe those symbolic execution engines that our research is based on.

2.1 Overview of Symbolic Execution

During its execution, every program performs operations on the input data in a defined order. Symbolic execution [53] is based on the idea that the program is operated on symbolic variables instead of specific input data, and the output will be a function of these symbolic variables. A symbolic variable is a set of the possible values of a concrete variable in the program, thus a symbolic state is a set of concrete states. When the execution reaches a selection control structure (e.g. an if statement) where the logical expression contains a symbolic variable, it cannot be evaluated, its value might be also true and false. The execution continues on both branches accordingly. This way we can simulate all the possible execution branches of the program. The key goal of symbolic execution in the context of software testing is to explore as many different program paths as possible in a given amount of time, and for each path to (1) generate a set of concrete test input values exercising that path during a normal execution, and (2) check for the presence of various kinds of errors including uncaught exceptions, memory corruption and security vulnerabilities.

During symbolic execution we maintain a so-called *path condition (PC)*. The path condition is a quantifier-free logical formula with the initial value of true, and its variables are the symbolic variables of the program. If the execution reaches a branching condition that depends on one or more symbolic variables, the condition will be appended to the current PC with the logical operator *AND* to indicate the true branch, and the negation of the condition to indicate the false branch. With such an extension of the PC, each execution branch will be linked to a unique formula over the symbolic variables. In addition to maintaining the path condition, symbolic execution engines make use of the so called *constraint solver* programs. Constraint solvers are used to solve the path condition by assigning values to the symbolic variables that satisfy the logical formula. Path condition can be solved at any point of the symbolic execution. Practically, the solutions serve as test inputs that can be used to run the program in such a way that the concrete execution follows the execution path for which the PC was solved.

All of the possible execution paths define a connected and acyclic directed graph called *symbolic execution tree*. Each point of the tree corresponds to a symbolic state of the program. An example is shown in Figure 2.1.

Figure 2.1 (a) shows a sample code that determines the distance of two integers x and y. The symbolic execution of this code is illustrated on Figure 2.1 (b) with the corresponding symbolic execution tree. We handle x and y symbolically, their symbols are X and Y respectively. The initial value of the path condition is true. Reaching the first if statement in line 3, there are two possibilities: the logical expression can be true or false; thus the execution branches and the logical expression and its negation is added to the PC as follows:

$$true \wedge X > Y \Rightarrow X > Y, \quad \text{and} \quad true \wedge \neg(X > Y) \Rightarrow X \leq Y$$

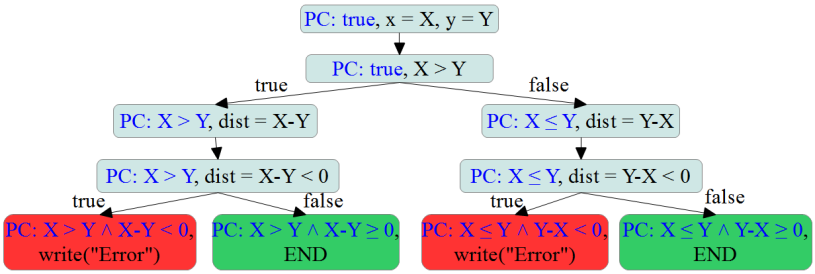
The value of variable *dist* will be a symbolic expression, X-Y on the true branch and Y-X on the false one. As a result of the second if statement (line 8) the execution branches, and the appropriate PCs are appended again. On the true branches we get the following PCs:

$$X > Y \wedge X - Y < 0 \Rightarrow X > Y \wedge X < Y,$$

$$X \leq Y \wedge Y - X < 0 \Rightarrow X \leq Y \wedge X > Y$$

```
1. int x, y, dist;
2. ...
3. if (x > y) {
4.   dist = x - y;
5. } else {
6.   dist = y - x;
7. }
8. if (dist < 0)
9.   write("Error");
```

(a)



(b)

Figure 2.1. (a) Sample code that determines the distance of two integers on the number line (b) Symbolic execution tree of the sample code handling variable x and y symbolically

It is clear that these formulas are unsolvable, we cannot specify such X and Y that satisfy the conditions. This means that there are no such x and y inputs with which the program reaches the *write("Error")* statement. As long as the PC is unsatisfiable at a state, the sub-tree starting from that state can be pruned, there is no sense to continue the controversial execution.

In accordance with the goal that we want find true positive runtime errors in the program we changed and extended the standard path condition building method described above as described in Chapter 4.

It is impossible to explore all the symbolic states. It takes unreasonably long time to execute all the possible paths. A solution for this problem can be e.g. to limit the depth of the symbolic execution tree or the number of states which, of course, inhibit to examine all the states. The approaches described in Chapter 3 and Chapter 5 address this problem.

2.2 Symbolic Execution Engines and Their Applications

The idea of symbolic execution is not new, the first publications and execution engines appeared in the 1970's. One of the earliest works is by King that lays down the fundamentals of symbolic execution [53] and presents the EFFIGY system that is able to execute PL/I programs symbolically. Even though EFFIGY handles only integers symbolically, it is an interactive system with which the user is able to examine the process of symbolic execution by placing breakpoints and saving and restoring states. Another work from the 1970's by Boyer et al. presents a similar system called SELECT [8] that can be used for executing LISP programs symbolically. The users are allowed to define conditions for variables and return values and get back whether these conditions are satisfied or not as an output. The system can be applied for test input generation and in addition, for every path it gives back the path condition over the symbolic variables.

Further description and comparison of the above mentioned and other tools can be found in the work of Coward [21] and Cadar [14].

Starting from the last decade the interest about the technique is constantly growing, numerous programs have been developed that aim at dynamic test input generation using symbolic execution.

There are approaches and tools for generating test suites for .NET programs using

symbolic execution. Pex [100] is a tool that automatically produces a small test suite with high code coverage for .NET programs using dynamic symbolic execution, similar to path-bounded model-checking. Jamrozik et al. introduce an extension of the previous approach called augmented dynamic symbolic execution [46], which aims to produce representative test sets with DSE by augmenting path conditions with additional conditions that enforce target criteria such as boundary or mutation adequacy, or logical coverage criteria. Experiments with the Apex prototype demonstrate that the resulting test cases can detect up to 30% more seeded defects than those produced with Pex.

KLEE [12] is another symbolic execution engine that aims to automatically generate tests that achieve high coverage. Similar to our work it is possible to use various heuristics to prioritize the most interesting paths first.

SonarQube [15] has its own symbolic execution engine for Java language, which similarly to us is intended to find tricky bugs that are almost uncatchable by developers unaided. It is able to find 3 kinds of issues compared to our 5: (1) null pointer dereference, (2) unclosed resource and (3) it has a rule named "condition should not unconditionally evaluate to true or false".

Java PathFinder [47] is an execution environment with the goal of verifying Java programs. JPF is a spceial Java Virtual Machine which interprets the Java bytecode in a way to be able to verify certain properties of the program. Symbolic PathFinder [75] is an extension of Java PathFinder which is able to symbolically execute Java programs by implementing a proper bytecode instruction set. This is the engine that we used to conduct our research described in Chapter 3.

The main application of the Java PathFinder and its symbolic execution extension is the verification of the internal projects in NASA. Bushnell et al. describes the application of Symbolic PathFinder in TSAFE (Tactical Separation Assisted Flight Environment) [11] that verifies the software components of an air control and collision detection system. The primary target is to generate useful test cases for TSAFE that simulates different wind conditions, radar images, flight schedules, etc.

The detection of design patterns can be performed using dynamic approaches as well as with static program analysis. With the help of a monitoring software the program can be analyzed during manual execution and conclusions about the existence of different patterns can be made based on the execution branches. In his work, von Detten [106] applied symbolic execution with Symbolic PathFinder supplementing manual execution. This way, more execution branches can be examined and the instances found by traditional approaches can be refined.

Ihantola [45] describes an interesting application of JPF in education. He generates test inputs for checking the programs of his students. His approach is that functional test cases based on the specification of the program and their outcome (successful or not) is not enough for educational purposes. He generates test cases for the programs using symbolic execution. This way the students can get feedback like “the program works incorrectly if variable a is larger than variable b plus 10”.

Song et al. applied the symbolic execution to the verification of networking protocol implementations [87]. The SymNV tool creates network packages with which a high coverage can be achieved in the source code of the daemon, therefore potential rule violations can be revealed according to the protocol specifications.

The SAFELI tool [30] by Fu and Qian is a SQL injection detection program for analyzing Java web applications. It first instruments the Java bytecode then executes

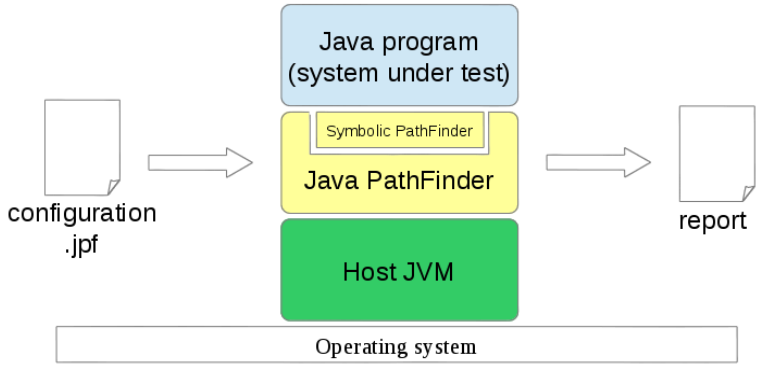


Figure 2.2. Java PathFinder as a virtual machine itself runs on a JVM, while performing a verification of a Java program

the instrumented code symbolically. When the execution reaches a SQL query the tool prepares a string equation based on the initial content of the web input components and the built-in SQL injection attack patterns. If the equation can be solved the calculated values are used as inputs which the tool verifies by sending a HTML form to the server. According to the response of the server it can decide whether the found input can be a real attack or not.

2.3 The Used Symbolic Execution Engines

2.3.1 Java PathFinder and Symbolic PathFinder

Java PathFinder (JPF) [47] is a highly customizable execution environment that aims at verifying Java programs. In fact, JPF is nothing more than a Java Virtual Machine which interprets the Java bytecode in a special way to be able to verify certain properties. It is difficult to determine what kind of errors can be found and which properties can be checked by JPF, it depends primarily on its configuration. The system has been designed from the beginning to be easily configurable and extendable. One of its extensions is *Symbolic PathFinder (SPF)* [75] that provides symbolic execution of Java programs by implementing a bytecode instruction set allowing to execute the Java bytecode according to the theory of symbolic execution.

JPF (and SPF) itself is implemented in Java, so it also have to run on a virtual machine, thus JPF is actually a middleware between the standard JVM and the bytecode. The architecture of the system is illustrated on Figure 2.2.

To start the analysis we have to make a configuration file with .jpf extension in which we specify different options as key-value pairs. The output is a report that contains e.g. the found defects. In addition to the ability of handling logical, integer and floating-point type variables as symbols, SPF can also handle complex types symbolically with the lazy initialization algorithm [51], and allows the symbolic execution of multi-threaded programs too.

SPF supports multiple constraint solvers and defines a general interface to communicate them. *Cvc3* is used to solve linear formulas, *choco* can handle non-linear logical formulas too, while *IASolver* use interval arithmetic techniques to satisfy the path condition. Among the supported constraint solvers, *CORAL* proved to be the most effective in terms of the number of solved constraints and the performance [88].

To reduce the state space of the symbolic execution SPF offers a number of options. We can specify the maximum depth of the symbolic execution tree, and the number of elementary formulas in the path condition can also be limited. Further possibility is that we can restrict the value ranges of the integer and floating point types with options *symbolic.minint*, *symbolic.maxint*, *symbolic.minreal*, and *symbolic.maxreal*. With the proper use of these options the state space and the time required for the analysis can be reduced significantly. Our concept described in Chapter 3 is based on the Symbolic PathFinder.

2.3.2 Symbolic Checker

At the Software Engineering Department of the University of Szeged we started to develop a symbolic execution engine that exactly meets our needs. The tool called Symbolic Checker is being developed with the goal of detecting runtime errors in Java applications without running the program in real-life environment. In contrast to other symbolic execution tools [12, 53, 100, 105, 109] generating test cases which lead to failure is not a goal here. Instead our aim is to produce a descriptive designation of the execution path that led to a fault.

Symbolic Checker is developed in C++. Currently the detection of four kinds of runtime faults are implemented: (1) null pointer dereferences, (2) array over-indexing, (3) array creation with negative size, and (4) division by zero errors.

Instead of starting the symbolic execution from the *main()* method, which is the entry point of a Java program, Symbolic Checker performs the analysis by symbolically executing each method of the system one by one as described in Chapter 3. This does not mean the engine cannot handle method calls, the analysis is interprocedural. The engine handles method calls by placing the actual parameters onto the stack and giving the control to the callee.

The parameters of the method and the referred but not initialized variables are handled as symbols at the beginning of the symbolic execution. Importantly, we only report an error if it is proven that the value that causes the problem during execution can be determined by constant propagation. I.e. if a method call passes a concrete null value, and Symbolic Checker finds a path in the called method that dereferences this parameter, we will report an error. If the dereferenced variable is a symbol the error will not be reported, because its value is unknown or uncertain.

The symbolic execution is performed using the language-dependent *abstract semantic graph (ASG)* [26] of the program by interpreting the nodes of this graph in a defined order. The order is defined by the language-independent *control flow graph (CFG)* [2], which is constructed to every method. Loops and recursions are not handled specially, the traversal of the CFG results simple unroll.

Symbolic Checker is able to handle integer, floating point, reference (including aliasing), and array type data. It models the memory by storing the variables in a special data structure for optimized memory usage: we do not store the whole variable set for each state, only the changes. I.e., if a state wants to read a variable which was assigned in a parent state, it will be found only in the storage of the appropriate parent, but if a variable was changed by the current state, the updated value will be stored in the variable storage of the current state. Moreover, each state has a stack used for passing parameters in method invocations and for expression evaluation.

In Symbolic Checker *Definition* is a comprehensive name for all the data that ap-

pears during symbolic execution. For example, the concrete or symbolic variables, constants, parameters of methods, their return value, or the result of sub-expressions are also Definitions. Actually, the symbolic execution of the program is the propagation of these Definition objects. Basically, there are two types of Definitions: ValueDefinition and SymbolDefinition. ValueDefinition objects store specific, concrete values and SymbolDefinition instances represent the symbolic variables. In fact, the symbolic execution of the program is nothing else than the proper propagation of these Definition objects on the different paths.

Symbolic Checker has a special way of building the path condition, which will be presented in Chapter 4. It uses the Gecode constraint solver tool-set [33].

To limit the size of the symbolic execution tree, its maximum depth and the maximum number of states can be specified.

The output of the Symbolic Checker is a CSV (Coma Separated Values) file that contains the detected errors indicating their type, the execution path from the entry point to the exact location where the error occurred and a probability that estimates how likely the analyzed method runs onto the detected fault.

2.3.3 RTEHunter

Symbolic Checker went through further developments and we gave it the new name of RTEHunter (RunTime Exception Hunter). Besides bug fixes, which result a more stable and reliable system, we redesigned and rewrote many essential parts that improved the performance and code quality. Instead of CSV files, the output that contains the detected errors (indicating their type and the execution path by a list of states from the entry point to the exact location where the error occurred) are written into well formatted txt files. Moreover, RTEHunter is integrated into the SonarQube quality management platform [15], thus the reported issues can be investigated in a SonarQube instance too. For now, RTEHunter is part of a commercial product, the SourceMeter static analysis tool-kit[1]. This engine was used in the research described in Chapter 5.

[1] https://www.sourcemeter.com/

"If you want to increase your success rate,
double your failure rate."

— Thomas J. Watson

3

A Method-level Symbolic Execution Technique for Runtime Error Detection in Real-world Software Systems

3.1 Overview

Our purpose is to develop a new method and tool, which supports maintenance activities, particularly debugging and bug fixing with detecting runtime faults in real-world Java programs, and finding dangerous parts in the source code, that could behave as time-bombs during further development.

In Java programs runtime errors are manifested as runtime exceptions which are the instances of class java.lang.RuntimeException. For example runtime exceptions occurred by an invalid type cast, an array over indexing, or by a division by zero operation. These exceptions are dangerous because they can cause a sudden stop of the program, as they do not have to be handled by the programmer explicitly. The exploration of these exceptions is done using the technique symbolic execution [53], which is able to explore possible execution paths of a program.

Java PathFinder (JPF) [47] is a software model checker which is developed at NASA Ames Research Center. This work is based on its extension, the Symbolic PathFinder (SPF) [75] which can perform the symbolic execution of Java bytecode.

However, if we start the symbolic execution from the standard entry point of the program (i.e. form method *main*()), the state space composed of the possible execution paths would explode before the symbolic execution could reach the majority of the code. To overcome this problem we perform the symbolic execution on each method of the system one by one. Concrete input parameters of the method resulting a runtime exception are also determined.

The chapter explains how the detection of runtime exceptions of the Java programming language was implemented using Java PathFinder and symbolic execution. The implemented tool called *JPF Checker* has been tested on open-source real life projects: on *log4j*, *ArgoUML*, and on *jEdit*. We found multiple errors in the *log4j* system that

were also reported as real bugs in its bug tracking system. The performance of the tool is acceptable since the analysis was finished in a couple of hours even for the biggest system used for testing.

The remainder of the chapter is organized as follows. Section 3.2 collects the works that are related to ours. In Section 3.3 we present our approach for detecting runtime exceptions. Section 3.4 discusses the results of the implemented algorithm on different small examples and real-life open source projects. Finally, we conclude the research in Section 3.5.

3.2 Related Work

In this section we present works are related to our research. We first introduce existing approaches and techniques relating to our main goal: runtime error detection. Since our method-level symbolic execution proposes an approach to handling the path explosion problem by applying symbolic execution for the portions of the program, we also mention works that reduced the state space in a similar manner.

The EXE (EXecution generated Executions) [13] presented by Cadar et al. at the Stanford University is an error checking tool made for generating input data on which the program terminates with failure. The input generation is done by the STP built-in constraint solver that solves the path condition of the path causing the failure. EXE achieved promising results on real-life systems. It found errors in the package filter implementations of *BSD* and *Linux*, in the *udhcpd* DHCP server and in different Linux file systems.

The DART [37] (Directed Automata Random Testing) by Godefroid et al. tries to eliminate the shortcomings of the symbolic execution e.g. when it is unable to handle a condition due to its unlinear nature. DART executes the program with random or predefined input data and records the constraints defined by the conditions on the input variables when it reaches a conditional statement. In the next iteration taking into account the recorded constraints it runs the program with input data that causes a different execution branch of the program. The goal is to execute all the reachable branches of the program by generating appropriate input data. The CUTE and jCUTE systems [80] by Sen and Agha extend DART with multithreading and dynamic data structures. The advantage of these tools is that they are capable of handling complex mathematical conditions due to concrete executions.

Sinha et al. deal with localizing Java runtime errors [83]. The introduced approach aims at helping to fix existing errors. They extract the statement that threw the exception from its stack trace and perform a backward dataflow analysis starting from there to localize those statements that might be the root causes of the exception.

The work of Weimer and Necula [108] focuses on proving safe exception handling in safety critical systems. They generate test cases that lead to an exception by violating one of the rules of the language. Unlike JPF Checker they do not generate test inputs based on symbolic execution but solving a global optimization problem on the control flow graph (CFG) of the program.

The JCrasher tool [23] by Csallner and Smaragdakis takes a set of Java classes as input. After checking the class types it creates a Java program which instantiates the given classes and calls each of their public methods with random parameters. This algorithm might detect failures that cause the termination of the system such as runtime exceptions. The tool is capable of generating JUnit test cases and can be

integrated to the Eclipse IDE. Similarly to JPF Checker JCrasher also creates a driver environment but it can analyze public methods only and instead of symbolic execution it generates random data which is obviously not feasible for examining all possible execution branches.

Bucur et al. [9] addresses the problem of path explosion by parallelizing symbolic execution in a way that scales well on large clusters of cheap commodity hardware. The system, called Cloud9 can automatically test real systems, that interact in complex ways with their environment.

Another approach to reduce the state space, presented by Chipounov [16], proposes the idea not to execute the whole program symbolically, but just portions of it. The engine can start the symbolic execution at arbitrary portions of a full system, including applications, libraries, operating system, and device drivers. It seamlessly transitions back and forth between symbolic and concrete execution, while transparently converting system states from symbolic to concrete and back. A similar approach is used for testing NASA Software [77]. The concept is also to start running the program in normal mode like in a real life environment then at given points, e.g. at more complex or problematic parts in the program switch to symbolic execution mode. In Section 3.3.1 we give a more detailed comparison of this approach with our method-level symbolic execution technique. The CUTE and jCUTE systems [80] by Sen and Agha, are also concolic executors, start at an arbitrary function by initializing pointers based at first on a simple heap with abstract addresses and incrementally increasing the heap complexity in subsequent runs.

System MIX [50] combines symbolic execution with static type checking based techniques. It designates type and symbolic blocks in the program, which determines which code-part should be analysed using symbolic execution and wich one using static type checking. In the border of these blocks so-called mix-rules are used to convey the neccessary information. MIX is intended to provide a compromise between the precise but resource intensive symbolic execution and the less precise but faster type checking.

Another technique is to reuse and merge the paths that are explored before. E.g. reusing the analysis of lower-level functions in subsequent computations improves the scalability of symbolic execution [35].

3.3 Detection of Runtime Exceptions

We developed a tool that is able to automatically detect runtime exceptions in an arbitrary Java program. This section explains in detail how this analysis program, the JPF checker works.

To check the whole program we use symbolic execution, which is performed by Symbolic PathFinder. However, we do not execute the whole program symbolically to discover all of the possible paths, instead we symbolically execute the methods of the program one by one. This approach is called method-level symbolic execution. Starting the analysis from the *main* method has the drawback that the state space would be too large and we would need to cut it when the execution reaches the defined maximal depth in the symbolic execution tree. Our approach results in a significant reduction in the state space of the symbolic execution.

An important question is which variables to be handled symbolically. In general, execution of a method mainly depends on the actual values of its parameters and the referred external variables. Thus, these are the inputs of a method that should be

handled symbolically to generally analyze it. Currently, we handle the parameters and data members of the class of the analyzed method symbolically.

Our goal is not only to indicate the runtime exceptions a method can throw (its type and the line causing the exception), but also to determine a parameterization that leads to throwing those exceptions. In addition, we determine this parameterization not only for the analyzed method which is at the bottom of the call stack, but for all the other elements in the call stack (i.e. recursively for all the called methods).

Our work can be divided into two stages:

1. It is necessary to create a runtime environment which is able to iterate over all the methods of a Java program, and start their symbolic execution using Symbolic PathFinder to implement method-level symbolic execution.
2. We need a JPF extension which is built on its listener mechanism, and which is able to indicate potential runtime exceptions and related parameterization while monitoring the execution.

3.3.1 The Runtime Environment

The concept of the developers of Symbolic PathFinder was to start running the program in normal mode like in a real life environment, than at given points, e.g. at more complex or problematic parts in the program switch to symbolic execution mode [77]. The advantage of this approach is that, since the context is real, it is more likely to find real errors. E.g. the values of the global variables are all set, but if these variables are handled symbolically we can examine cases that never occur during a real run. A disadvantage is that it is hard to explore the problematic points of a program, it requires prior knowledge or preliminary work. Another disadvantage is that you have to run the program manually namely, that the control reach those methods which will be handled symbolic by the SPF.

In contrast, the tool we have developed is able to execute an arbitrary method or all methods of a program symbolically. The advantage of this approach is that the user does not have to perform any manual runs, the entire process can be automated. Additionally, the symbolic state space also remains limited since we do not execute the whole program symbolically, but their parts separately. The approach also makes it possible to analyze libraries that do not have a *main* method such as log4j. One of the major disadvantages is the that we back away from the real execution environment, which may lead to false positive error reports.

For implementing such an execution environment we have to achieve somehow that the control flow reaches the method we want to analyze. However, due to the nature of the virtual machine, JPF requires the entry point of the program, which is the class containing the main method. Therefore, we generate a driver class for each method containing a main method that only passes the control to the method we want to execute symbolically and carries out all the related tasks. Invoking the method is done using the Java Reflection API. We also have to generate a JPF configuration file that specifies, among others, the artificially created entry point and the method we want to handle symbolically. After creating the necessary files, we have to compile the generated Java class and finally, to launch Symbolic PathFinder.

The architecture of the system is illustrated in Figure 3.1. The input *jar* file is processed by the *JarExplorer* component, which reads all the methods of the classes from the jar file and creates a list from them. The elements of the list is taken by the

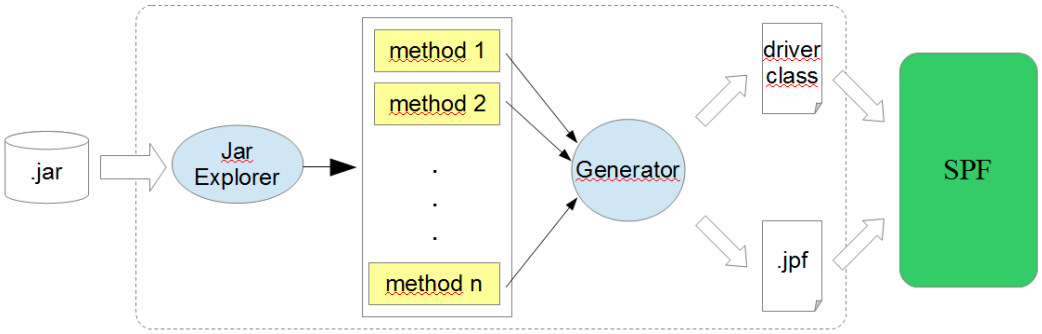


Figure 3.1. Architecture of the runtime environment

Generator one by one. It generates a driver class and a JPF configuration file for each method. After the generation is complete, we start the symbolic execution.

3.3.2 Implementing a Listener Class

During functioning, JPF sends notifications about certain events. This is realized by so-called listeners, which are based on the observer design pattern. The registered listener objects are notified about and can react to these events. JPF can send notifications of almost every detail of the program execution. There are low-level events such as execution of a bytecode instruction, as well as high-level events such as starting or finishing the search in the state space. In JPF, basically two listener interfaces exist: the *SearchListener* and *VMListener* interface. While the former includes the events related to the state space search, the latter reports the events of the virtual machine. Because these interfaces are quite large and the specific listener classes often implement both of them, adapter classes are introduced that implement these interfaces with empty method bodies. Therefore, to create our custom listener we derived a class from this adapter and implemented only the necessary methods.

Our algorithm for detecting runtime exceptions is briefly summarized below. By performing symbolic execution of a method all of its paths are executed, including those that throw exceptions. When an exception occurs, namely when the virtual machine executes an ATHROW bytecode instruction, JPF triggers and *exceptionThrown* event. Thus, we implemented the exceptionThrown method in our listener class. Its pseudo code is shown in Listing 3.1.

First, we acquire the thrown Exception object (line 2), then we decide whether it is a runtime exception (i.e. whether it is an instance of the class RuntimeException) (line 3). If it is, we request the path condition related to the actual path and use the constraint solver to find a satisfactory solution (lines 4-5). Lines 6-9 set up a summary report that contains the type of the thrown exception, the line that throws it and a parameterization which causes this exception to be thrown. The parameterization is constructed by the *parsePC()* method, which assigns the satisfactory solutions of the path condition to the method parameters. Lines 10-13 take care of collecting and determining parameterization for the methods in the call stack. If the source code does not specify any constraint for a parameter on the path throwing an exception (i.e. the path condition does not contain the variable), then there is no related solution. This means that it does not matter what the actual value of that parameter is, as it does

```
1  exceptionThrown() {
2    exception = getPendingException();
3    if (isInstanceOfRuntimeException(exception)) {
4      pc = getCurrentPc();
5      solve(pc);
6      summary = new FoundExceptionSummary();
7      summary.setExceptionType(exception);
8      summary.setThrownFrom(exception);
9      summary.setParameterization(parsePc(pc, analyzedMethod));
10     invocationChain = buildInvocationChain();
11     foreach(Method m : invocationChain) {
12       summary.addStackTraceElement(m, parsePc(pc, m));
13     }
14     foundExceptions.add(summary);
15   }
16 }
```

Listing 3.1. Pseudo code of the exceptionThrown event

```
1  void x(int a) {
2    short b = 42;
3    y(a, b);
4  }
5  void y(int a, short b) {
6    ...
7    throw new NullPointerException();
8    ...
9  }
```

Listing 3.2. An example call with both symbolic and concrete parameters

not affect the execution path, and the method is going to throw an exception due to the values of other parameters. In such cases parsePc() method assigns the value "any" to these parameters.

It is also possible that a parameter has a concrete value. Listing 3.2 illustrates such an example. When we start the symbolic execution of method *x()*, its parameter *a* is handled symbolically. As *x()* calls *y()* its parameter *a* is still a symbol, but *b* is a concrete value (42). In this case, parsePc() have to get the concrete value from the stack of the actual method.

We note that the presented algorithm reports any runtime exceptions regardless of the fact whether it is caught by the program or not. The reason of this is that we think that relying on runtime exceptions is a bad coding practice and a runtime exception can be dangerous even if it is handled by the program. Nonetheless, it would be easy to modify our algorithm only to detect uncaught exceptions.

3.4 Results

The developed tool was tested in a variety of ways. The section describes the results of these test runs. We analyzed manually prepared example codes containing instructions

that cause runtime exceptions on purpose; then we performed analysis on different open-source software to show that our tool is able to detect runtime exceptions in real programs, not just in artificially made small examples. The subject systems are the log4j[1] logging library, the ArgoUML[2] modeling tool, and the jEdit[3] text editor program. We prove the validity of the detected exceptions by the bug reports, found in the bug tracking systems of these projects, that describe program faults caused by those runtime exceptions that are also found by the developed tool.

3.4.1 Manually Prepared Examples

A small manually prepared example code is shown on Listing 3.3. The method under test is *callRun()* which calls method *run()* in line 12. Running our algorithm on this code gives two hits: the first one is an ArrayIndexOutOfBoundsException and the second one is a NullPointerException. The first exception is thrown by method run() at line 24. A parametrization leading to this exception is callRun(7, 11). Method run() will be called only if $x > 6$ (line 10) that is satisfied by 7 and it is called with the concrete value 9 and symbol y. At this point there is no condition for y. Method run() can reach line 24 only if $y > 10$, the indicated value 11 is obtained by satisfying this constraint. Throwing of the ArrayIndexOutOfBoundsException is due to the fact that in line 22 we declare a 5-element array but the following for loop runs from 0 to x. The value of x at this point is 9 which leads to an exception.

The train of thought is similar in the case of the second exception. The problem is that variable i created in line 27 initialized only in line 29 to a value different form *null*, but not in the else block, therefore line 33 throws a NullPointerException. This requires that the value of y not to be greater than 10 and not to be less than 5. These restrictions are satisfied by e.g. 5, and value 7 for x is necessary to invoke run(). So the parametrizations are callRun(7, 5) and run(9, 5). The analysis is finished in less than a second.

A second example code is presented in Listing 3.4. The resulting report refers to an ArithmeticException, which is thrown at line 39 and the stack trace highlights that the problematic method is *expand()* which is invoked at line 30 by *run()*. The control flow reaches line 30 only if variable b is false. For example, if n is -999, and *check* has the value true, as the parameter list in the error report included, b will be false and the *expand()* method on the else branch will be executed. At line 36, variable *res* has a concrete value because method *count()* will be executed. It can be seen that *res* is definitely a non-negative integer, thus the condition at line 37 is true if *n=-999*. Then the loop begins executing, and variable *res* will be reduced to 0 after a number of iterations, leading to a division by 0 fault. In the report, the third parameter of the examined *run()* method is "any". That is because this parameter does not play a role in whether or not the program runs onto the discussed ArithmeticException.

Line 25 in method *run()* also calls *expand()*, but there is no corresponding error report. In fact, due to the instructions at lines 13-23, the condition at line 24 is always false, thus this *expand()* call will never be executed. Actually, line 25 is unreachable code.

[1] `http://logging.apache.org/log4j/`
[2] `http://argouml.tigris.org/`
[3] `http://www.jedit.org/`

```
   public class Example5 {
     ...
8  void callRun(int x, int y) {
9    Integer i = null;
10   if (x > 6) {
11     int b = 9;
12     run(b, y);
13     i = Integer.valueOf(b);
14     System.out.println(i);
15   } else {
16     i = Integer.valueOf(3);
17     System.out.println(i);
18   }
19 }
20 public void run(int x, int y) {
21   if (y > 10) {
22     int[] arr = new int[5];
23     for (int i = 0; i < x; i++) {
24       arr[i] = i;
25     }
26   } else {
27     Integer i = null;
28     if (y < 5) {
29       i = Integer.valueOf(4);
30       i.floatValue();
31     } else {
32       System.out.println(
33                       i.floatValue());
34     }
35   }
36 }
37 }
```

Listing 3.3. Manually prepared example code with the analysis of method callRun()

3.4.2 Analysis of Open-source Systems

Analysis of log4j 1.2.15, ArgoUML 0.28 and jEdit 4.4.2 were carried out on a desktop computer with an Intel Core i5-540M 2.53 GHz processor and 8 GB of memory. In all three cases the analysis was done by executing all the methods of the release jar files of the projects symbolically.

Figure 3.2 (a) displays the number of methods we analyzed in the different programs. We started analyzing 1242 methods in log4j of which only 757 were successful, in 474 cases the analysis stopped due to the failure of the Java PathFinder (or Symbolic PathFinder). There are a lot of methods in ArgoUML which also could not be analyzed, more than half of the checks ended with failure. In case of jEdit the ratio is very similar. Unfortunately, in general JPF stopped with a variety of error messages.

Despite the frequent failures of JPF, our tool indicated a fairly large number of runtime exceptions in all three programs. Figure 3.2 (b) shows the number of successfully analyzed methods and the methods with one or more runtime exceptions. The hit rate is the highest for log4j and despite its high number of methods, relatively few exceptions were found in ArgoUML.

The analysis times are shown in Figure 3.2 (c). Analysis of log4j completed within an hour, while analysis of ArgoUML, that contains more than 7500 methods, took 3 hours and 42 minutes. Although jEdit contains fewer methods than ArgoUML, its full analysis were more time-consuming. The performance of our algorithm is acceptable, especially considering that the analysis was performed on an ordinary desktop PC not on a high-performance server. However, it can be assumed that the analysis time would grow with less failed method analysis.

It is important to note, that not all indicated exceptions are real errors. This happens because the analysis were performed in an artificial execution environment which might have introduced false positive hits. When we start the symbolic execution of a method we have no information about the circumstances of the real invocation. All parameters and data members are handled symbolically, that is, it is considered

```
    ...
3   public class Example3 {
    ...
8   public void run(int n,
          boolean check, A a) {
9     boolean b = check && n >= 0;
10    int max = Integer.MIN_VALUE;
11    if (b) {
12      if (a != null) {
13        int l = n;
14        int r = 2*n + 1;
15        if (a.getMember() > 120) {
16          if (l <= a.getMember()) {
17            max = a.getMember();
18          } else {
19            max = l;
20          }
21          if (r > max) {
22            max = r;
23          }
24          while (max < n) {
25            max = expand(n, 0);
26          }
27        }
28      }
29    } else {
30      max = expand(n, 0);
31    }
32    System.out.println("Maximum"
33                       + "value:␣" + max);
34  }
35  private int expand(int n, int m) {
36    double res = count(m);
37    if (res > n) {
38      do {
39        res = n / res;
40        res -= 2;
41      } while (res >= 0);
42      return n + m;
43    } else {
44      return (int)res;
45    }
46  }
47
48  private int count(int l) {
49      int count = l;
50      for (int i=100; i>0; i--) {
51        if (i % 3 == 0) {
52          count++;
53        }
54      }
55      return count;
56    }
57
58  }
```

```
1   public class A extends Letter {
2     ...
3     public int member;
4
5     public int getMember() {
6       return member;
7     }
8     ...
9   }
```

Listing 3.4. Manually prepared example code with the analysis of method run()

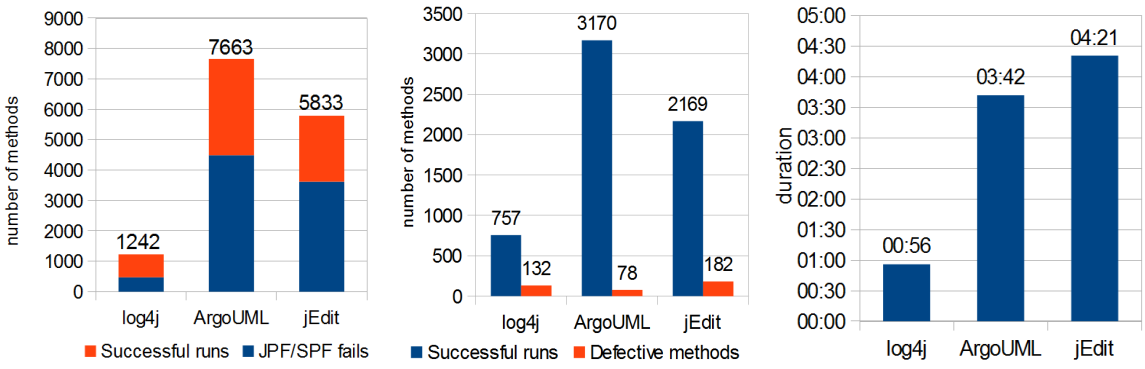


Figure 3.2. (a) The number of methods examined in the programs and the number of JPF or SPF faults. (b) The number of successfully analyzed methods and the number of defective methods. (c) Analysis time

that their value can be anything although it is possible that a particular value of a variable never occurs.

Despite the fact that not all the reported exceptions are real program errors they are definitely representing real risks. During the modification of the source code there are inevitably changes that introduce new errors. These errors often appear in form of runtime exceptions (i.e. in places where our algorithm found possible failures). So the majority of the reported exceptions do not report real errors, but potential sources of danger that should be paid special attention to.

In the following, we are going to show some interesting faults found by our tool in the above systems.

The first example method is *org.apache.log4j.SimpleLayout.format()* of log4j, which is shwon in Listing 3.5. In this method three possible runtime exceptions are found by the tool. The first two are NullPointerExceptions, both thrown at line 61. The produced report says that the first NPE will be thrown if the parameter is *null*, and the second when this parameter differs from *null*. In the first case, when the parameter is null, expression *event.getLevel()* causes the exception, since a method of a null reference cannot be called. When parameter *event* is not null, the code gets the *level* data member and calls its *toString()* method. The second NullPointerException is caused by the fact that the requested *level* data member can also be null, thus using operator '.' may raise the exception.

The third exception is a ClassCastException. As shown, at line 256 in class *LoggingEvent* there is a type cast which tries to convert the *level* member which has a type *Priority* to a *Level* object. According to the code in the bottom of Listing 3.5, class *Level* is a descendant of class *Priority*, thus the cast at line 256 is a downcast, which is incorrect in case the dynamic type of the member is not *Level*.

Three possible ClassCastExceptions are revealed in method *PredicateMType.create()* that is depicted in Listing 3.6. Lines 729, 730 and 731 cast down the three parameters from *Object* to *Class* without performing any type check. The first entry in the report says that *create(null, null, !null)* parametrization can lead to an exception thrown at line 731. If *c0* and *c1* parameters are null, lines 729 and 730 are executed without any problem, because casting a null reference to any class is permitted in Java. It is important that this does not mean that *c0* and *c1* have to be necessarily null, the

```
      public class SimpleLayout extends Layout {
        ...
58      public String format(LoggingEvent event) {
59
60        sbuf.setLength(0);
61        sbuf.append(event.getLevel().toString());
62        sbuf.append("␣−␣");
63        sbuf.append(event.getRenderedMessage());
64        sbuf.append(LINE_SEP);
65        return sbuf.toString();
66      }
        ...
      }
      public class LoggingEvent implements java.io.Serializable {
        ...
        transient public Priority level;
          ...
255     public Level getLevel() {
256       return (Level) level;
257     }
      }
      public class Level extends Priority implements Serializable{
        ...
      }
```

Listing 3.5. Method org.apache.log4j.SimpleLayout.format() and its environment

```
      public class FindDialog extends ArgoDialog ... { ... }
      class PredicateMType extends PredicateType {
        ...
727    public static PredicateType create(Object c0, Object c1, Object c2) {
728      Class[]  classes  = new Class[3];
729      classes[0] = (Class) c0;
730      classes[1] = (Class) c1;
731      classes[2] = (Class) c2;
732      return new PredicateMType(classes);
733    }
       ...
      }
```

Listing 3.6. Method org.argouml.ui.PredicateMType.create()

```
     public class MRUFileManager {
       ...
       private LinkedList mruFileList;
       ...
       public int size() {
         return mruFileList.size();
       }
97     public Object getFile(int index) {
98       if (index < size()) {
99         return mruFileList.get(index);
100      }
101
102      return null;
103    }
       ...
     }
```

Listing 3.7. Method org.apache.log4j.lf5.viewer.configure.MRUFileManager.getFile()

report just gives a sample parametrization which leads the execution to the exception. As long as the third parameter is not null a ClassCastException can be raised. Of course, to achieve this it is necessary that the parameter type is different from *Class*. Parametrization *create(null, !null, "any")* leads to potential fault at line 730. The reasoning is similar to the previous one: if *c0* is null and *c1* is non-null (and of course it is not a *Class*) ClassCastException will be thrown. The third parameter is completely irrelevant. In case of the third ClassCastException, occurring at line 729, the values of *c1* and *c2* do not matter.

The last example is a tiny method, *MRUFileManager.getFile()* shown in Listing 3.7. At line 98, *getFile()* checks whether the *index* parameter is less then the size of the *mruFileList* LinkedList. If so, the return value is the corresponding element of the LinkedList, otherwise *null*. Our report shows that the *index* can be a negative number, too. This case is not handled, and *LinkedList.get()* will throw an IndexOutOfBoundsException if method *getField()* is called for example with -999. Calling *getField()* with a negative number seems unreasonable and of course it is, but possible.

```
    public class ThrowableInformation implements java.io.Serializable {
      private transient Throwable throwable;
      ...
54    public String[] getThrowableStrRep() {
55      if(rep != null) {
56        return (String[]) rep.clone();
57      } else {
58        VectorWriter vw = new VectorWriter();
59        throwable.printStackTrace(vw);
60        rep = vw.toStringArray();
61        return rep;
62      }
63    }
      ...
    }
```

Listing 3.8. Method org.apache.log4j.spi.ThrowableInformation.getThrowableStrRep()

```
    public class NDC {
      ...
      static Hashtable ht = new Hashtable();
      ...
374   static
375   public
376   void remove() {
377     ht.remove(Thread.currentThread());
378
379     // Lazily remove dead-thread references in ht.
380     lazyRemove();
381   }
      ...
    }
```

Listing 3.9. Source code of method org.apache.log4j.NDC.remove()

3.4.3 Real Errors

In this subsection a few defects are presented which are reported in bug tracking systems, and caused by runtime exceptions found also by our tool. The first affected bug[4] reports the termination of an application using log4j version 1.2.14 caused by a NullPointerException. The reporter got the Exception from line 59 of *ThrowableInformation.java* thrown by method *org.apache.log4j.spi.ThrowableInformation.getThrowableStrRep()* as shown in the given stack trace. The code of the method and the problematic line detected by our analysis is shown in Listing 3.8.

The problem here is that the initialization of the *throwable* data member of class *ThrowableInformation* is omitted, its value is null causing a NullPointerException at line 59. This causes that the *log()* method of log4j can also throw an exception which should never happen. Our tool found other errors as well which demonstrate its strength of being capable of detecting real bugs.

[4] https://issues.apache.org/bugzilla/show_bug.cgi?id=44038

The next exception is also a NullPointerException, which occurred in log4j 1.2.15. The bug report[5] explains that the runtime exception causing the halt comes form method *org.apache.log4j.NDC.remove()*, at line 377. Listing 3.9 shows the corresponding piece of code. The fault here is that the *ht* static data member is null. Although the data member is initialized as Listing 3.9 shows, it is possible that during the execution its value is set to null. The report in the log4j bug tracking system sheds light to this. The reporter also mentions that according to his observations, the other methods of class *NDC*, which use the *ht* member, first check whether it is null or not, but in method *remove()* there is no such investigation.

We describe one more error that was also found in log4j version 1.2.15[6]. The error is at line 312 of the class *org.apache.log4j.net.SyslogAppender*. The line is inside the method *append()* in which there is a NullPointerException again. The code snippet is in Listing 3.10. The reason of this runtime error is that the *layout* data member, which is inherited from class *AppenderSkeleton*, stays uninitialized. Our report also includes a ClassCastException thrown by method *getLevel()* at line 294. This fault is the same that we already described explaining Listing 3.5 in the previous subsection.

3.5 Summary

The introduced approach that performs method-level symbolic execution for detecting runtime exceptions works well not just on small, manually prepared examples but it is able to find runtime exceptions which are the causes of some documented runtime failures (i.e. there exists an issue for them in the bug tracking system) in real world systems also. However, not all the detected possible runtime exceptions will actually cause a system failure. There might be a large number of exceptions that will never occur running the system in real environment. Nonetheless, the importance of these warnings should not be underrated since they draw attention to those code parts that might turn to real problems after changing the system. Considering these possible problems could help system maintenance and contributes to achieving a better quality software. As we presented in Section 3.4 the analysis time of real world systems are also acceptable, therefore our approach and tool can be applied in practice.

Unfortunately the Java PathFinder and its Symbolic PathFinder extension – which we used for implementing our approach – contain a lot of bugs. It made the development very troublesome, but the authors at the NASA were really helpful. We contacted them several times and got responses very quickly; they fixed some blocker issues particularly for our request. Although JPF and SPF have several bugs, it is under constant development and becoming more and more stable.

The author's contributions. The author performed the exploration of symbolic execution and the Symbolic PathFinder execution engine for the purpose of runtime exception detection. The idea of method-level symbolic execution and the entire implementation of the runtime environment which performs the analysis is the author's work. He performed the detection of runtime exceptions by implementing a module in Symbolic PathFinder which also gives a stack trace leading to the found error and the related parametrization as a test input that crashes the program. The investigation of the found runtime exceptions and proving their validity by the bug reports, found

[5] https://issues.apache.org/bugzilla/show_bug.cgi?id=45335
[6] https://issues.apache.org/bugzilla/show_bug.cgi?id=46271

```
    public abstract class AppenderSkeleton {
      protected Layout layout;
       ...
    }
    public class SyslogAppender extends AppenderSkeleton {
      SyslogQuietWriter sqw;
      private boolean layoutHeaderChecked = false;
       ...
291   public
292   void append(LoggingEvent event) {
293
294     if(!isAsSevereAsThreshold(event.getLevel()))
295       return;
296
297     // We must not attempt to append if sqw is null.
298     if(sqw == null) {
299       errorHandler.error("No syslog host is set for SyslogAppedender"
300                                       + " named " + this.name + ".");
301       return;
302     }
303
304     if (!layoutHeaderChecked) {
305         if (layout != null && layout.getHeader() != null) {
306             sendLayoutMessage(layout.getHeader());
307         }
308         layoutHeaderChecked = true;
309     }
310
311
312     String packet = layout.format(event);
313     String hdr = getPacketHeader(event.timeStamp);
314
315     if( facilityPrinting || hdr.length() > 0) {
316         StringBuffer buf = new StringBuffer(hdr);
317         if( facilityPrinting ) {
318             buf.append(facilityStr );
319         }
320         buf.append(packet);
321         packet = buf.toString();
322     }
       ...
    }}
```

Listing 3.10. Source code of method org.apache.log4j.net.SyslogAppender.append()

in the bug tracking systems of the analyzed projects was also the author's role. He contacted the authors of Symbolic PathFinder several times to report some blocker issues that held back the research. The publications related to this chapter are:

- **I. Kádár**, P. Hegedűs, R. Ferenc. Runtime Exception Detection in Java Programs Using Symbolic Execution. In *Proceedings of the 13th Symposium on Programming Languages and Software Tools – SPLST'13*, pages 215–229, 2013.
- **I. Kádár**, P. Hegedűs, R. Ferenc. Runtime Exception Detection in Java Programs Using Symbolic Execution. In *Acta Cybernetica*, pages 331–352, 2014.

"Don't say 'I cannot'. Say 'I presently struggle with.' "

— Tony Horton

4

A Constraint Building Mechanism for Symbolic Execution to Improve Runtime Error Detection Accuracy

4.1 Overview

In the work presented in this chapter, we developed a constraint building mechanism and integrated it into the Symbolic Checker symbolic execution engine, which allows the detection of runtime errors that a conventional symbolic execution system cannot do.

According to the theory of symbolic execution [53], the program does not run with specific input data, but the inputs are handled as symbolic variables. When the execution of the program reaches a branching condition containing a symbolic variable, the execution continues on both branches. At each branching point, both the affected logical expression and its negation are accumulated on the true and false branches, thus all execution paths will be linked to a unique formula over the symbolic variables called path condition. Test case generation is performed by solving these collected constraints using a constraint solver, and the feasibility of a path is checked by solving this formula as well.

This chapter describes a novel constraint system construction mechanism, which improves the accuracy of the runtime errors found by the Symbolic Checker symbolic execution engine by treating the assignments in the program as conditions too. Hence, we can track the dependencies of the symbolic variables by extending the original principles of symbolic execution. The presented method also substitutes symbolic variables with concrete values if the built constraint system unambiguously determines their value. To build and satisfy the constraint systems, we used the open-source Gecode constraint satisfaction tool-set [33].

The chapter is organized as follows: In Section 4.2 we mention publications that are related to this research, thereafter Section 4.3 explains in detail how the algorithm that builds the novel constraint system for each execution path is implemented and

how it is integrated into Symbolic Checker. After that, in Section 4.4 we describe how the algorithm enhanced the effectiveness of the engine. We evaluate the results of tests performed on real-life open-source systems and we also compare the approach to our previous tool [94], JPF Checker which uses the original constraint building principle. Finally, we conclude our work in Section 4.5.

4.2 Related Work

In this section first, we introduce some existing approaches that support specific fields in the application of symbolic executor tools, similarly to our work. We also mention constraint solvers that are used in the existing symbolic execution implementations.

Shannon and others [81] built an abstraction layer above the Java string handling using finite state automatas to extend the handling of String-type data in symbolic execution. In addition to the implementation of the java.lang.String class, StringBuilder and StringBuffer classes are modeled as well. As a result, the system is able to handle constraints that contains strings and string operations, thus it can be applied to programs that are working on more complex strings, such as SQL queries. Currently, Symbolic Checker does not handle string constraints, we plan to deliver this development in the future.

Durring symbolic execution, it may occur that the built path condition contains function calls, e.g. $if(y >= f(x))$. The so-called concolic (concrete-symbolic) [76] execution provides a possible solution to this problem using a special constraint building mechanism. The main idea of this approach is that two path conditions are maintained at the same time. One of them contains those conditions witch do not includes function calls, and the other is the so-called complex PC, in which there are conditions that includes function calls too. First, the algorithm satisfies the simple PC and assigns values to the included symbols. Then these values are used to execute those included functions concretely which execution depended on the symbols whose values have been determined in the first step. This method also capitalizes on turning symbols into concrete values, like the approach we present in this chapter.

In Symbolic PathFinder [75], there are multiple constraint solvers integrated through a generic interface: *CVC3* can be applied on linear formulas, *CHOCO* is able to handle non-linear constraints over integer and floating point variables too, *IASolver* uses interval-arithmetic methods in order to satisfy the path condition, and *CORAL*, which has been proved to be the most effective in satisfying path conditions containing complex mathematical functions considering the number of solved constraints and the time consumption of the satisfaction algorithm [88]. CORAL (italics-ban) is able to handle such complex constraints that often occur in the analysis of software in the aerospace domain, for example TSAFE [11] that helps air-traffic controllers in detecting and resolving short-term conflicts between aircrafts.

EXE [13] automatically generates a test case by solving the current path constraints to find concrete values using its own codesigned constraint solver, STP [31]. The main goal in EXE was to quickly solve the constraints generated by the code through a combination of low-level optimizations and a series of higher-level ones, such as caching and irrelevant constraint elimination.

4.3 Constraint Building

4.3.1 Principles

As Section 2.3.2 describes, Symbolic Checker reports errors only if the value causing the problem becomes concrete. The tool does not fire for symbolic variables, because if a variable is a symbol that actually means that it's value is doubtful, not known. I.e. if an expression is divided by a value which is zero ($expression/0$) on an execution path traversed by the engine, the tool fires but if it is divided by a symbol ($expression/Symbol$), it does not. It may occur that during the symbolic execution of a program most of the variables turn into symbols, which makes finding runtime errors more difficult.

The main idea behind the developed constraint building mechanism is that, if during analysis the program sets up conditions (constraints) that unambiguously determines the value of one or more symbolic variables, then we can convert these symbols into concrete values and the symbolic execution can be continued on the actual path using the concreted variables. Since these variables, handled as concrete data, it is possible to detect errors that Symbolic Checker could not otherwise find. The conditions we mentioned above are determined by the conditional control structures (if, switch, while, etc.) and expressed by the assignments of the program, including the impacts of the increment and decrement operators ($++, --$) of the Java language.

Overall, the goal of the implemented constraint building mechanism is the concretion of as many symbols as possible, which helps find more runtime errors. In order to achieve this, (1) it is necessary to build a special path condition (PC) that also contains the dependencies of the symbolic variables determined by the assignments of the program, and (2) if the constraints in the PC determine the values of some symbols unambiguously, the execution has to be continued using these concrete values on the actual path. This extended path condition also includes – in some form – those conditions that can be found in the conventional PC. Hence, if the extended PC cannot be satisfied, the code parts that are unreachable can also be skipped with the novel approach. Therefore, false positive defects can be eliminated.

To demonstrate the basic idea of extending the PC, consider the code snippet in Figure 2.1 (a). In this example, the conventional path condition of the path which passes through the true branch of the if statement in line 3, and the false branch of the if in line 8 is the following:

$$X > Y \ \wedge \ \neg(X - Y < 0) \ \Rightarrow \ X > Y \ \wedge \ X - Y \geq 0.$$

According to our concept, the extended PC of the same path is the following:

$$X > Y \ \wedge \ \neg(dist < 0) \ \wedge \ dist = X - Y.$$

It is clear that the variable *dist* is also included in the constraint system as a symbol, once in the negation of condition in line 8 where X-Y was not substituted and also in the constraint that expresses the assignment in line 4. As a result, the constraint system also contains information about variable *dist* that could be useful in the later stages of the execution.

In this example, the extended PC does not contain constraints which could unambiguously determine any variable, thus the benefit of the extension is not obvious here. The code snippet in Listing 4.1 shows an example where the extended PC has some gains indeed.

```
1  // c is an int symbol
2  double b = 2*c + 4;
3  int a = b + 9;
4  if (a > 8) {
5     ...
6     if (a < 10) {
7        // concretion of b
8        int p = 1/b;
9     }
10 }
```

Listing 4.1. Sample code that provides symbol concretion which helps find runtime errors that a conventional symbolic execution tool cannot

```
1  // a and b are symbols
2  if (b > 0) {
3     ...
4     if (a == 0) {
5       // concreting a?
6        ...
7     }
8  }
```

Listing 4.2. Code snippet which points out a path condition that has more solutions but concreted the symbol a

Executing the code symbolically in Listing 4.1, handling variable c as a symbol, the following constraint system will be built in the program state at line 7:

$$a > 8 \ \wedge \ a = b + 9 \ \wedge \ b = 2 \cdot c + 4 \ \wedge \ a < 10.$$

The constraint system above includes constraints that are introduced by the if statements of the code and the dependencies of symbol a, i.e. those constraints that are given by the assignments that define variable a. After satisfying this constraint system, it can be obtained that a can only be 9, which implies that the values of b and c symbols are unambiguous too: $b = 0.0$ and $c = -2$. In such a situation, the execution continues on the path for which the extended PC was satisfied. In the case of this example, if the execution continues with the b=0.0 value, a division by zero error can be detected at line 8. As long as symbol b is not included in the PC, and if its unambiguous value are not used, the detection of division by zero will fail.

In real-life programs quite large constraint systems are built with many symbols. It is clear that satisfying such a large set of constraints as a whole has a low probability for only one possible solution.

Listing 4.2 shows a code snippet that highlights the problem in question. Considering the path that passes along the true branches of both if statements, the path condition is $b > 0 \ \wedge \ a = 0$. Although there are infinite solutions for this formula because the $b > 0$ constraint can be satisfied by any positive integer, the formula determines the value of symbol a unambiguously, which would be preferred in later stages of the execution.

To overcome the problem, we decompose the path condition into connected components, that is, to constraint sets that are independent i.e. does not contain the same variables. The connected components can be satisfied individually and, if some

of them determine a variable unambiguously, the obtained values can be used later in the execution. Two constraints are in the same component if they contain at least one common variable. After such a decomposition, the path condition becomes a set of constraint sets.

```
1  Constraint constraint;
2  set<Constraint> actualConstraints;
3  if (onTrueBranch()) {
4    constraint = constraintBuilder.createConstraint();
5  } else if (onFalseBranch()) {
6    constraint = constraintBuilder.createNegatedConstraint();
7  }
8  actualConstraints.insert(constraint);
9  actualConstraints.union(dependenciesOfSymbolsInConstraint);
10 pathCondition.insert(actualConstraints);
11 decomposedPC = decompose(pathCondition);
12 foreach (set<Constraint> s : decomposedPC) {
13   constraintSolver.solve(s);
14   if (s.hasSolution) {
15     if (!s.hasMoreSolution) {
16       buildBackSolutions(s);
17     }
18   } else {
19     weight = 0.0;
20     break;
21   }
22 }
```

Listing 4.3. Pseudo code of the algorithm of constraint system building

The essential steps of the algorithm of our constraint system building are shown in Listing 4.3. This algorithm will be executed after each branching point in the symbolic execution tree.

First, it is determined whether the accumulation of the PC happens on the true or on the false branch then dependent upon this, the created logical expression or its negation is stored in variable *constraint* (lines 3-7) (the handling of switch statement of the Java programming language is not shown in the pseudo code). It is important to note that we build constraint exactly from that logical expression that is determined in the source code, there is no substitution of variables like in case of variable *dist* in Section 2.1, in example 2.1. Next, the created constraint is added to the *actualConstraints* constraint set (line 8), and the dependencies of the symbols included in this constraint are also inserted (line 9). These dependencies are defined by the assignments of the code (we will later discuss how they are created). In the next step, the path condition of the current execution path is extended by the *actualConstraints* constraint set (line 10), then the PC will be decomposed into connected components in line 11. As long as one of the connected components cannot be satisfied, the weight of the current path is set to 0.0 indicating that there is no point continuing the execution because of the contradictory conditions (line 19). On the other hand, if there is at least one solution, the algorithm examines its uniqueness (line 13). If the solution is unique, the concrete values are built back into the current symbolic state (line 16).

It has to be emphasized that a concreted symbol is built back into the a state only once and only into the state for which the constraint system concreted it.

4.3.2 Implementation

We used the Gecode constraint solver tool-set [33] for building and satisfying our constraint systems. Basically, we can differentiate two kinds of constraints: (1) conditions in the conditional control structures of the program (including the loops too) and (2) the dependencies of symbols which are included in these conditions. Below, we describe how we implemented the building of the constraint system and integrated into Symbolic Checker.

As we described in Section 2.3.2, for every kind of data that appears in a program during the symbolic execution (e.g. variables, literals, sub-expressions, etc.), a *Definition* object is created. The symbolic execution of the program is the proper propagation of these Definition objects. The task is to achieve the tracking that determines what other Definitions a Definition object is created from and what operations it uses. This is how the relations between symbolic variables are described.

For the implementation, we added a so-called *constraintSolverExpression* data member to the class Definition and a dependency set as well, which is a set of constraints. These attributes are propagated with the Definitions along the program by the symbolic execution.

The constraintSolverExpression represents an expression object created using the Gecode constraint solver. With this, Gecode can represent the inner structure of expressions (i.e. the operands they are created from using which operators). The constraintSolverExpressions is propagated in the following way: when an operation is performed on Definition objects, we take the constraintSolverExpressions of the operands and perform the operation on the expressions too, and the resulting compound constraintSolverExpression will be set in the resulting Definition object. In the case of operations performed on ValueDefinitions for efficiency reasons, the operation on the constraintSolverExpressions is not performed. Instead, we simply create a new Gecode expression which stores the calculated value.

The dependency set contains the dependencies of those symbols which are in the constraintSolverExpression defined by the assignments of the program. In the case of assignments where the right side is a SymbolDefinition, we create a new symbol for the variable which is on the left. This new symbol will not take over the constraintSolverExpression of the right side, but the relation between left and right Definitions is expressed by an equality constraint between them. The dependency set is propagated in the following manner. After performing an operation, the dependency set of the resulting Definition will be the union of the dependency sets of the operands. In the case of an assignment which has SymbolDefinition on the right side, the dependency set of the newly created SymbolDefinition on the left will be the dependency set of the right side symbol extended by the constraint that defines equality between the two sides.

```
1  // d is a symbol
2  int b = d + 3;
3  int c = 2*d;
4  int a = b - c;
5  if (42 == a) {
6     ...
7  }
```

Listing 4.4. Sample code for demonstrating the propagation of dependency sets

The branching conditions in selection control structures which define the branching points of the symbolic execution tree are also expressions in the program, thus they appear as Definition objects (actually as SymbolDefinitions) in Symbolic Checker. Variable *constraint* in the algorithm shown in Figure 4.3 is created from the constraitnSolverExpression of such a Definition object, and constraint set *dependenciesOfSymbolsInConstraint* is the dependency set of this Definition too. Constraint set *actualConstraints* by which the path condition will be extended is the union of the above mentioned *constraint* and *dependenciesOfSymbolsInConstraint*.

In the following, we demonstrate the building of the constraint system and the propagation of dependency sets for the example code in Listing 4.4. Variable d is handled as a symbol, it is a SymbolDefinition where the dependency set is empty and its constraintSolverExpression is a Gecode expression which contains only a simple unknown variable. Firstly, in line 2 a ValueDefinition is created for literal 3, with an dependency set, then operation $+$ creates the $d{+}3$ SymbolDefinition. The dependency set of $d{+}3$ is the dependency set of the left and the right side, which is also an empty set:

$$SymbolDef(d+3).depset \;=\; SymbolDef(d).depset \cup ValueDef(3).depset \;=\; \emptyset.$$

After the execution of the assignment the dependency set of b is:

$$SymbolDef(b).depset \;=\; SymbolDef(d+3).depset \cup \{b = d+3\} \;=\; \{b = d+3\}\,.$$

Dependency set of symbol c created at line 3 is quite similar:

$$SymbolDef(c).depset \;=\; SymbolDef(2*d).depset \cup \{c = 2\cdot d\} \;=\; \{c = 2\cdot d\}\,.$$

At the left hand side of assignment at line 4, dependency set of SymbolDefinition b-c is the union of dependency set of b and c:

$$\begin{aligned} SymbolDef(b-c).depset \;&=\; SymbolDef(c).depset \cup SymboldDef(b).depset \\ &=\; \{b = d+3,\; c = 2\cdot d\}\,. \end{aligned}$$

Then the dependency set of a:

$$\begin{aligned} SymbolDef(a).depset \;&=\; SymbolDef(b-c).depset \cup \{a = b-c\} \\ &=\; \{b = d+3,\; c = 2\cdot d,\; a = b-c\}. \end{aligned}$$

Dependency set of SymbolDefinition created from expression $42 == a$ at line 5 is the same as the dependency set of a, thus the path condition on the true branch of the if statement is the following:

$$\begin{aligned} PC \;&=\; \{42 = a\} \cup \{b = d+3,\; c = 2\cdot d,\; a = b-c\} \\ &=\; \{42 = a,\; b = d+3,\; c = 2\cdot d,\; a = b-c\}. \end{aligned}$$

4.4 Evaluation

Symbolic Checker with the novel constraint building mechanism was tested in a variety of ways. This section contains the results of these tests. First, we demonstrate the advantages of our algorithm through two examples emphasizing the difference compared (1) to JPF Checker which uses traditional constraint building and (2) to Symbolic Checker without using any constraint building mechanisms. After that, we write about the experiences of the tests we have performed on large, real-life systems.

```
1  class Example {
2
3   public void run(int n) {
4     int max = getCharPos('w');
5     A[] arr = new A[max];
6     if (n > 0) {
7       for (int i = 0; i < max; ++i) {
8         arr[i] = new A(max - i);
9       }
10      int sum1 = 0;
11      while (n < max) {
12        if (n % 2 == 0) {
13          sum1 += arr[n].getMember();
14        }
15        n++;
16      }
17      System.out.println("Sum1:" + sum1);
18      int negOfGcd = -gcd(n, arr[0]);
19      int sum2 = 0;
20      while (negOfGcd < max) {
21        sum2 += arr[negOfGcd++]
                          .getMember();
22      }
23      System.out.println("Sum2:" + sum2);
24    }
25  }
26
27    public int getCharPos(char c) {
28      return c - 'a' + 1;
29    }
30
31    private int gcd(int x, int y) {
32      while (y != 0) {
33        int m = x % y;
34        x = y;
35        y = m;
36      }
37      return x;
38    }
39
40  }
41
42  class A {
43    private int member;
44
45    public A(int member) {
46      this.member = member;
47    }
48
49    public int getMember() {
50      return member;
51    }
52  }
```

Listing 4.5. Example code with the analysis of method run()

In *run()* method of the example code shown in Listing 4.5, Symbolic Checker with constraint building detects an array over-indexing fault. First, we follow the cause of the runtime error, then we look at how the new approach helps to detect it. Line 5 defines an array called *arr* with size of *max*. As long as parameter n is grater than 0 (line 6), a sequence of operations will be performed which aims to calculate two sums based on the content of the array. This sequence of operations fills the array at first (lines 7-9), then starting from n, summarizes the *member* data members of objects on every second index into the variable *sum1* (lines 11-16). Next, the code calls method *gcd()* with arguments n and the 0th element of array *arr* (line 18) and summarizes the elements of the array starting from the negation of the return value of *gcd()* (lines 20-22). Method *gcd()* calculates the greatest common divisor of the numbers and its return value must be a positive integer if the arguments are n and *arr[0]*. Because of this, variable negOfGcd is guaranteed to be negative causing an ArrayIndexOutOfBoundsException runtime error that results in the halt of the program.

When starting the analysis with method run(), variable n is the only symbol. Variable *max* is concrete, array arr is also instantiated concretely and all of its elements are concrete values too. However, the execution of the loop in line 11 depends on n. On the false branch the execution continues from line 16, on the true branch we enter into the body of the loop, and – after executing it – we will branch again depending on the condition at line 11. This operation will continue until it reaches the maximum

depth of the symbolic execution tree. If the execution paths entered into the loop at least once and then exited, the following constraints must be part of the extended path condition:

$$n_{prev} < max \ \wedge \ n = n_{prev} + 1 \ \wedge \ \neg(n < max) \Rightarrow$$

$$n_{prev} < max \ \wedge \ n = n_{prev} + 1 \ \wedge \ n >= max.$$

In this formula, n_{prev} means the instance of symbol n when the execution just entered the loop. The $n_{prev} < max$ constraint can be defined in this state. As the result of the incrementation, a new symbol is created in line 15 and the $n = n_{prev} + 1$ constraint is built. Since in the next iteration the execution do not enters into the loop but continues on the false branch, it is necessary to create the $\neg(n < max)$ constraint too. After satisfying the constraint set above, symbol n will be determined unambiguously, and its value is equal to the value of variable *max*. This means that if the execution exits the while loop, the value of n must be *max*.

Building back the unambiguous value of n into the current symbolic state, the arguments of method call *gcd()* are both concrete values, thus it will be executed concretely and its return value will also be a concrete number. As we assumed, the return value must be a positive integer leading to a bad array indexing in line 21.

```
    // SMTPAppender.java
    public class SMTPAppender extends
                                AppenderSkeleton {
      ...
      protected Layout layout;
      protected CyclicBuffer cb =
                        new CyclicBuffer(bufferSize);
      ...
      protected
      void sendBuffer() {
        ...
224     int len = cb.length();
225     for(int i = 0; i < len; i++) {
226
227       LoggingEvent event = cb.get();
228       sbuf.append(layout.format(event));
229       if(layout.ignoresThrowable()) {
230         String[] s =
                event.getThrowableStrRep();
231         if (s != null) {
232           for(int j=0; j<s.length; j++) {
233             sbuf.append(s[j]);
234           }
235         }
236       }
237     }
        ...
      }
      ...
    }
```

```
    public class CyclicBuffer {
      int numElems;
      ...
101   public
102   LoggingEvent get() {
103     LoggingEvent r = null;
104     if(numElems > 0) {
105       numElems--;
106       r = ea[first];
107       ea[first] = null;
108       if(++first == maxSize)
109         first = 0;
110     }
111     return r;
112   }
      ...
119   public
120   int length() {
121     return numElems;
122   }
      ...
    }

    public class SimpleLayout extends Layout {
      ...
56    public
57    String format(LoggingEvent event) {
58
59      sbuf.setLength(0);
60      sbuf.append(event.getLevel().toString());
61      sbuf.append(" - ");
62      sbuf.append(event.getRenderedMessage());
63      sbuf.append(LINE_SEP);
64      return sbuf.toString();
65    }
      ...
    }
```

Listing 4.6. Method org.apache.log4j.net.SMTPAppender.sendBuffer() and its environment

The example detailed above highlights that a concreted symbolic variable can make a significant part of the execution concrete. The spread of symbols can be reduced, thus fewer variables have to be handled as unknown and uncertain data. As a result, the analysis becomes faster because fewer execution paths have to be examined. In this example, without concreting variable n we would have had to explore the whole symbolic execution tree of method gcd(), which is rather expensive because of the loop inside.

The demonstrated ArrayIndexOutOfBoundException cannot be detected by the JPF Checker, nor by Symbolic Checker without constraint building.

In the second example, we show a real code part from the log4j logging system. Consider method *org.apache.log4j.net.SMTPAppender.sendBuffer()* in Listing 4.6 from

log4j version 1.2.11, in which we point out that our new approach can also eliminate false positive faults as the conventional path condition construction.

In line 228 of method *sendBuffer()*, *get()* method of class *CyclicBuffer* is called, which returns a *LoggingEvent* reference. First of all, method get() initializes the reference r to *null* (line 103), then if the *numElems* data member is greater than 0, r gets a new value. However, on the false branch it returns the null-initialized r reference. Following this false branch, in method *SMTPAppender.sendBuffer()* variable event is initialized to null in line 227, this null value will be propagated into method *SimpleLayout.format()*, which dereferences it in line 60.

However, this null dereference would be a false positive error, because in line 60 the null value never occurs. In line 224, we get the *numElems* member of object *cb* for which the first iteration of the for loop at line 225 defines a constraint. The PC looks like this:

$$0 < len \ \wedge \ len = cb.numElems.$$

Nevertheless, method *get()* called at line 227 returns null only on the false branch where the $numElems > 0$ constraint is not satisfied, thus the path condition is the following:

$$0 < len \ \wedge \ len = cb.numElems \ \wedge \ \neg(numElems > 0).$$

This formula, however, is unsatisfiable, which means that the execution can not continue on this path. Variable *event* will not get the null value in line 227, so the method *format()* of class *SimpleLayout* cannot dereference it. This actually means that the execution enters the for loop in line 225 only if the value of variable *len* is at least 1, but in this case method *get()* cannot return null on the false branch of the if statement in line 104.

The elimination of the discussed false positive error would fail using Symbolic Checker without the constraint building mechanism, but JPF Checker would also eliminate it because no symbols are concreted and the unsatisfiability of the path condition is also tested by the JPF/SPF based tool.

We ran Symbolic Checker with the presented constraint building mechanism on large Java systems too, however, the evaluation of the results is not entirely finished yet. Manually reviewing the reported errors is rather time-consuming because of the difficulty in interpreting the long execution paths from the entry point to the point where the error was detected in the source code. What we have seen in the results so far is that there are significantly fewer runtime errors in the resultant report obtained by Symbolic Checker that uses the constraint building mechanism compared to the ones that JPF Checker produces. This does not mean that the report of Symbolic Checker does not contain false positive results, but most of them draw attention to real errors and potential sources of errors.

Considering the duration of the analyses, the run-time of Symbolic Checker using the constraint building stays below the run-time of JPF Checker, but this duration is about twice as long than Symbolic Checker running it without the constraint building mechanism. The analysis of the log4j logging library took slightly less than half an hour without constraint building, and the duration is about an hour using the new approach. We of course expected such a time requirement because the building of the constraint system, decomposing it to connected components, and especially its satisfaction are rather computation-intensive tasks.

4.5 Summary

The basic principles of symbolic execution have been known for decades and several tools were made that utilize the possibilities offered by this technique. Symbolic Checker, the tool that we developed at the Software Engineering Department of the University of Szeged differs from most of these tools because it does not aim to generate test inputs, but to detect execution paths that lead to runtime errors and dangerous code parts as accurately as possible. In order to reach this goal, we developed a constraint building mechanism and integrated it into Symbolic Checker. The presented approach builds a constraint system for each execution path, which also includes constraints over the variables that depend on the inputs handled as symbolic variables. In case of unambiguity the concreted values are used in the later stages of the analysis. As a result, it enables the detection of runtime errors that would not be possible using a conventional symbolic execution tool. For example, the demonstrated ArrayIndexOutOfBoundsException in Section 4.4 cannot be detected by the JPF Checker, nor by Symbolic Checker without constraint building. By concreting symbolic variables, the size of the symbolic execution tree can be reduced as well, which also implies improvements in performance. The ability to eliminate false positive results is preserved, because the proposed method also ignores paths that carry contradictory constraints similarly to the original constraint building approach.

The author's contributions. The author took part in the design and development of the Symbolic Checker symbolic execution engine as the lead developer. He devised the concept of the proposed constraint building mechanism. He implemented and integrated it into the symbolic execution engine. The evaluation of the proposed method by comparing it to the conventional approach and performing tests on example codes and on real-life systems are also the author's work. The publication related to this chapter is:

- **I. Kádár**, P. Hegedűs, R. Ferenc. Adding Constraint Building Mechanisms to a Symbolic Execution Engine Developed for Detecting Runtime Errors. In *Proceedings of the International Conference on Computational Science and Its Applications – ICCSA*, volume 9159 Lecture Notes in Computer Science (LNCS), pages 20–35, Springer International Publishing, 2015.

"The only way to do great work is to love what you do."

— Steve Jobs

5

Novel Search Strategies for Symbolic Execution and Empirical Investigation of State Space Limitations

5.1 Overview

Symbolic execution explores the possible execution paths of a program. However, the number of execution paths increases exponentially with the number of branching points, thus symbolic execution engines still struggle to achieve scalability. To overcome this problem, these tools set up different kinds of constraints over the tree composed of execution paths (symbolic execution tree). For example, the number of symbolic states, the depth of the execution tree, or the time consumption of the analysis is limited. In our improved symbolic execution tool called RTEHunter, the maximum depth of the symbolic execution tree (means symbolic state depth) and the maximum number of states can be adjusted and arbitrary strategies of the tree traversal can also be implemented. This way, the detection of possible runtime failures is done by traversing a sub-tree of the whole symbolic execution tree of the program.

Our goal is to find the optimal parametrization of RTEHunter in terms of maximum number of states, maximum depth of the symbolic execution tree and search strategy in order to find more runtime issues in less time. This means we have to figure out which part of the whole execution tree contains the most runtime issues while taking into consideration the time consumption of the exploration. Moreover, the search strategy is also essential in directing the exploration towards those states in the sub-tree where it is more likely to find issues and skip those that are supposed to be error-free. Maximum depth limits the height of the tree, and with a fixed depth the maximum number of states defines its width, while the search strategy determines the order in which the states in this limited size tree will be visited.

The main contributions of this research are the following:

- We found out how the *maximum number of states affects* the execution time and the number of found errors without any constraint on the depth.

- We found out how the *maximum number of states* together with the *maximum depth of the symbolic execution tree* impact both the amount of detected runtime issues and the analysis time.
- As our main contribution we propose two novel search strategies that successfully increase the number of detected runtime issues by guiding the search towards the more error-prone source-code parts.

The chapter is organized as follows: In the next section we summarize related works describing techniques that proposed for handling state explosion in symbolic execution, and how they prioritize the paths to be explored. In Section 5.3 we give a more detailed description of how RTEHunter works and builds the execution tree to understand our approach in Section 5.4. In Section 5.5 we publish and evaluate the results, Section 5.6 describes the possible threats to validity, and finally in Section 5.7 we conclude our work.

5.2 Related Work

Here we summarize studies that are similar in the handling of the program path explosion and the methods used to select and prioritize the most valuable execution path.

To reduce the state space in symbolic execution, the Symbolic PathFinder [75] based on the Java PathFinder model checker offers a number of options. Similar to the RTEHunter, the maximum depth of the symbolic execution tree can be specified and the number of elementary formulas in the path condition can be limited. A further possibility is that with options we can restrict the value ranges of the integer and floating point type symbolic variables. In addition, Symbolic PathFinder lazily initializes object references and uses types to infer aliasing.

In order to make symbolic execution more scalable, Majumdar and Xu propose using symbolic grammars to guide symbolic execution by reducing the space of possible inputs [60]. Godefroid et al. had a similar approach. They set up grammar-based specifications of highly-structured inputs of symbolic execution, such as compilers and interpreters [36].

Loops and recursions with conditions that cannot be evaluated during the symbolic execution results in infinite constructs that can explode the state space without benefit. For this reason, the handling of these constructs can have crucial effects. CBMC is a Bounded Model Checker for C and C++ programs [19]. CBMC is able to verify array bounds, exception handling correctness, pointer safety, and user defined assertions as well. In bounded model checking, the potentially infinite constructs (e.g. while loops, recursion) are unwound only n times where the number n is the upper bound. CBMC pre-processes the program into an equivalent program that only uses while, if, goto statements and assignments. Next, all while loops are unwound using the following transformation n times:

while (cond) instruction; $\rightarrow$ *if (e) {instruction; while(cond) instruction}*

The last while loop is replaced by assertion *!cond* ensuring that the program never performs more iterations. This unwinding assertion is verified along with the user defined assertions, and one more iteration is needed in the unwinding in the case of failure. Following this, the program only consists of if instructions, assignments, assertions, labels, and forward goto instructions, which are then transformed into static

single assignment (SSA) form. From this, a bit vector equation is assembled together with the target rule to be verified. If this equation is satisfiable, the tool finds a violation. The mechanism used here to unroll loops as many times as necessary would be promising to integrate into RTEHunter to reduce the state space generated by loops. Currently, we are using a simple unrolling until we reach the depth or the overall state limit. Another strategy would be to recognize patterns in the basic block sequence during a loop unwinding. E.g. when a basic block is visited too many times, the execution is deep into a recursion or a loop. This strategy works well for the Clang Static Analyzer [55].

One of the key mechanisms used by symbolic execution tools to prioritize path exploration is search heuristics. Most heuristics focus on achieving high statement or branch coverage, but they could also be employed to optimize other desired criteria. The main difference compared to our work is that we optimize for execution time and also for the number of detected issues.

Similar to RTEHunter, in Klee [12] it is possible to use various heuristics to prioritize the most interesting paths first. KLEE selects the state to run at each instruction by interleaving the two search heuristics: random state selection and coverage-optimized search that tries to select states likely to cover new code. Our search heuristics does not consider any test coverage information yet, but we are planning to test its efficiency in a future work.

Random exploration is proved to be efficient for test generation by Burnim et al [10] too.

The error checking tool EXE (EXecution generated Executions) [13] presented by Cadar et al. generates input data on which the program terminates with failure. The heuristic in EXE favors previously visited statements that were run the fewest number of times.

The AUSTIN tool uses fitness function to drive evolutionary search of the test input space with dynamic symbolic execution [56].

Ma et al. [58] focuses on debugging scenarios for cases when the developer already knows about a faulty line but might not know exactly how to reproduce the failure or even whether it is reproducible. The approach also uses search strategies that aim to direct symbolic execution to the target line. One strategy is the *shortest-distance symbolic execution (SDSE)*, which will pick the path that currently has the shortest distance to the target line according to the control flow graph (CFG) of the program. The other one starts at the target line and works backwards until it finds a realizable path from the start of the program, using standard forward symbolic execution as a subroutine.

In general, our problem can be placed in the area of Search Based Software Engineering (SBSE), where search based optimization algorithms are used to address problems in software engineering, for example to figure out the smallest set of test cases that cover all branches in the program or the set of requirements that balance software development cost and customer satisfaction. Harman et al. provides a comprehensive survey of this area [39]. In our case, we look for the symbolic execution tree that has smallest exploration time and covers the greatest number of runtime issues.

5.3 Deeper insights into RTEHunter

This subsection gives a description of how RTEHunter constructs and traverses the symbolic execution tree in order to understand the optimization approaches and investigations presented in this chapter. We also mention how RTEHunter handles cases where an issue is found multiple times, because this is essential in the evaluation of the results.

Symbolic execution is performed using the language-dependent abstract semantic graph (ASG) [26] of the program by interpreting the nodes of this graph in the order defined by the language-independent control flow graph (CFG) [2]. The ASG and the CFG are assembled by the SourceMeter toolchain.

The nodes of the control flow graph are called *basic blocks*. A basic block represents a straight-line piece of code that is guaranteed to execute sequentially (i.e. it does not include any jumps or jump targets) by lining up the appropriate ASG nodes according to the sequential execution. Directed edges between the basic blocks are used to represent jumps in the control flow. In RTEHunter, for each analyzed method the symbolic execution tree is constructed by traversing the CFG and for every basic block a symbolic state will be created in the tree.

```
1  int distance(int x, int y) {
2      int dist;
3      if (x > y) {
4          dist = x - y;
5      } else {
6          dist = y - x;
7      }
8      if (dist < 0) {
9          System.out.println("Error");
10     }
11     return dist;
12 }
```

Listing 5.1. Java method *distance()* that determines the distance of two integers on the number line

Figure 5.1 shows the control flow graph constructed for the Java method *distance()* presented in Listing 5.1. The symbolic execution tree that RTEHunter creates using the control flow graph is in Figure 5.2.

Listing 5.2 presents the simplified algorithm in RTEHunter that builds the symbolic execution tree while symbolically executing each path. The presented search and build strategy is depth-first search. The construction of the execution tree starts with method *search()*. Here, we first get the root state of the tree and initialize the *strategy* object with that. The while loop always gets the next state to be executed. The execution of a state is done by *executeState()*. This interprets the nodes that are in the basic block which the state is created from according to the semantics of the Java programing language. The strategy object gives the next state according to the implemented strategy. In this listing, the strategy is implemented in class *DepthFirstSearchStrategy*, in which the *getNextState()* method is the essential part of the traversal. It gets the top-most element from the stack and expands this state meaning to get all of its descendants then puts them onto the stack. The *expandState()* method of our expander object constructs the child states according to the descendent basic blocks in the CFG

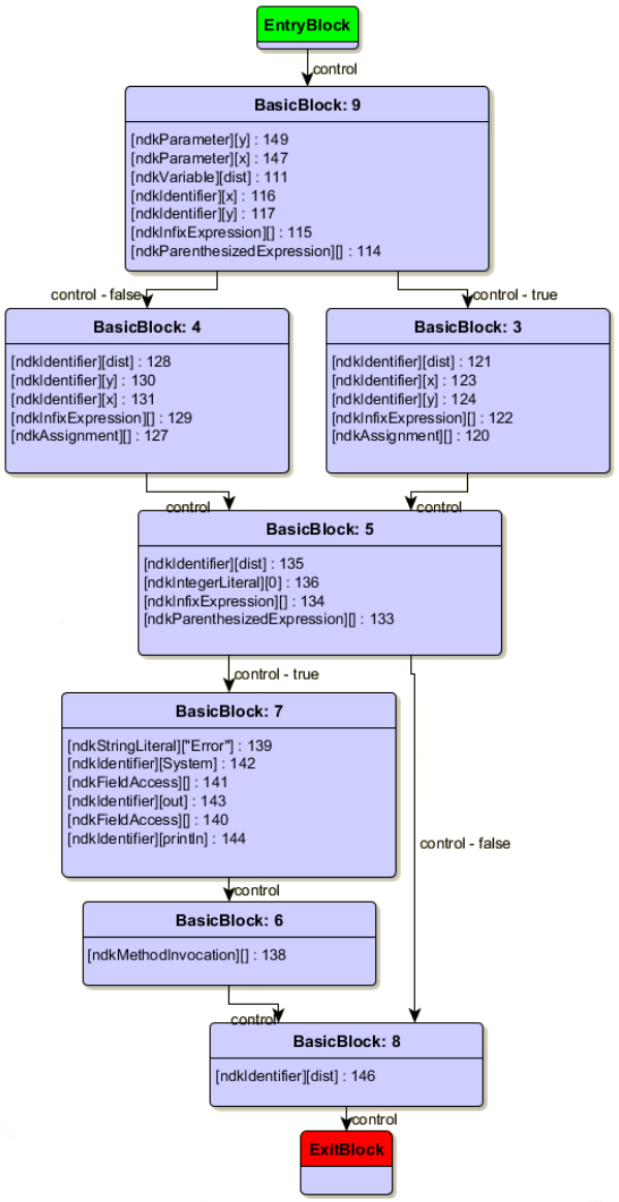


Figure 5.1. The control flow graph (CFG) constructed for the method in Listing 5.1

and the information that comes from the execution of the parent states. For example, if the parent state stands for an if statement which has two children, but the logical expression can be evaluated by the execution of the parent, the *expandState()* will only give the appropriate child, not both of them. The stack data structure (LIFO queue) provides the depth-first search traversal.

In this chapter, when we talk about "maximum depth" and "maximum state number" we mean the depth and the number of nodes (states) of the tree constructed by the algorithm above. It is also possible to define custom search strategies by replacing the demonstrated depth-first search strategy to guide the execution towards other paths.

The same runtime failures can be detected multiple times if we explore multiple execution paths which led to the same program location. However, it is not obvious when two errors should be considered the same, because the original cause of two errors can be different despite the fact that detection points are the same. For example, a reference type variable can be set to null at different locations in the program, and

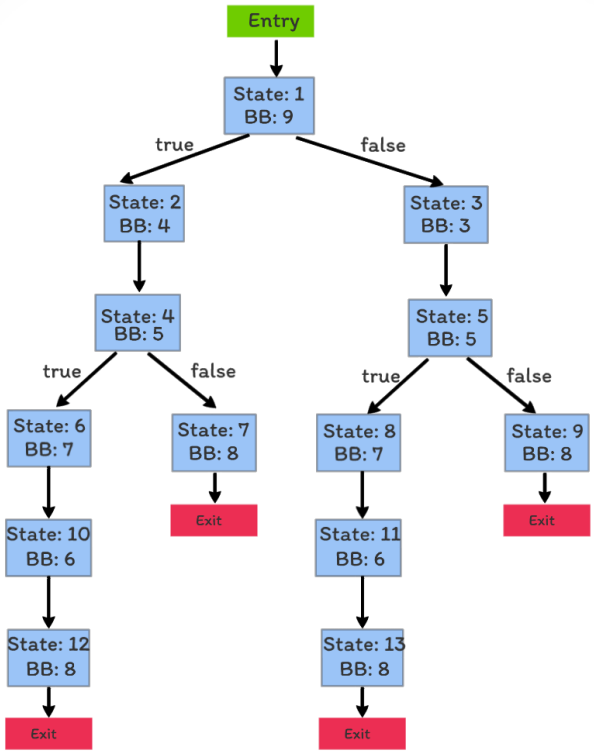


Figure 5.2. The symbolic execution tree constructed by RTEHunter to the code in Listing 5.1.

then dereferenced at the same place causing a *NullPointerException*. The detection location is the same, the causes (and the possible fixes) may differ. On the other hand, when examining the results we realized that in most cases, the cause was the same when multiple paths led to the same location because one path is a suffix to the others. Due to this observation, we only keep the shorter paths and filer out the others when multiple paths lead to the same error location.

5.4 Approach

5.4.1 Optimal Maximum Depth and State Number

As we mentioned in Section 5.3, two kinds of limitations can be set up in RTEHunter in order to limit the size of the symbolic execution tree preventing the state explosion: (1) the maximum depth (which means state depth), and (2) the maximum number of states can be adjusted. With a given maximum depth, the maximum state number defines the width of the tree. It should be added that these limitations are applied to each symbolic execution tree built for each method of the given system separately and these are not global limits for the system-wide analysis.

We ran RTEHunter with different depth and state number limits on three open source Java systems. The systems analyzed with their total lines of code metric are listed in Table 5.1.

In addition to the maximum depth and the maximum state number, the final shape of the symbolic execution tree is also determined by the applied search strategy.

In our experiments, we sought to ascertain the optimal depth and state number in order to find more runtime errors in less time, and we used the default depth-first

```
1  Strategy* strategy = new DepthFirstSearchStrategy(expander);
2
3  void search(StateFactory stateFactory) {
4    State *rootState = stateFactory.getRootState();
5    strategy->initialize(*rootState);
6    State* nextState = NULL;
7    while (nextState = strategy->getNextState())
8      executeState(*nextState);
9  }
10
11 class DepthFirstSearchStrategy: public SearchStrategy {
12   private:
13     std::stack<State*> stack;
14
15   public:
16     DepthFirstSearchStrategy(StateExpanderInterface& expander)
17       : SearchStrategy(expander), stack() {}
18
19     void initialize (State& rootState) {
20       stack.push(&state);
21     }
22
23     State* getNextState() {
24       State* front = stack.top();
25       stack.pop();
26       std::vector<State*> children=expander.expandState(*front);
27       for (State* child : children)
28         stack.push(child);
29       if (stack.empty())
30         return NULL;
31       return stack.top()
32     }
33 };
```

Listing 5.2. The main algorithm of tree building and execution shown in RTEHunter.

System	TLOC
ArgoUML	372K
Jetspeed	275K
JFreeChart	329K

Table 5.1. The Java systems on which the measurements were carried out

search strategy.

5.4.2 Custom Search Strategies

Since the search strategy that is used to direct the traversal influences the shape of the final symbolic execution tree, it plays a significant role in finding those states where runtime errors might occur.

In the actual stage of the traversal of the state space (i.e. the symbolic execution tree), a search strategy tells us from which state among the current leaf states the exploration have to continue, i.e. which state have to be expanded as the next step of the traversal.

In contrast to depth-first search where the actual leaf states placed into a LIFO (last in, first out) queue, in our custom search strategies a score is assigned to every actual leaf, which will be placed into a priority queue. Upon the engine have to choose a state as the next step, it chooses the one with the highest score to continue the traversal with. The implementations of these strategies are very similar to the code of class *DepthFirstStrategy* shown on Listing 5.2, but the stack member is replaced by a priority queue that orders the states by score, and the top is the one has the highest score.

Next, we will describe two new search strategies that implement heuristics to direct the search towards the potential runtime issues.

The null-heuristic Search Strategy

This search strategy seeks to drive the traversal to find more null pointer dereference issues. Our motivation of focusing on null pointer dereferences is that according to static analysis the most common checks against exceptions are null-checks in Java sources, implying that this is the most common runtime issue that may occur [27, 79]. We also discovered that this type of runtime error is the most common one that RTEHunter detects.

For each state we summarize the number of reachable reference-type values (variable values, literals, function return values, etc.) reachable, whose value is *null* at the current symbolic state of the given Java program. To continue the traversal, the engine chooses the state with the highest number of null values assigning higher probability value to find possible null pointer dereferences in that state and in the sub-tree obtained from it.

A Linear Regression-Based Search Strategy

We also developed a search strategy that supports the detection of not just null pointer dereferences, but all the four types of issues that RTEHunter is able to detect. To implement such a search strategy, we used a linear regression model that assigns a score to each leaf state during the search. The score is the estimated number of runtime issues that might have been detected in the sub-tree reachable from the state. We chose linear regression because it can be applied on continuous class labels and provides a relatively quick result for an unseen example.

The training data of the linear regression model contains one training example for each state got from symbolic execution trees that were traversed previously by the engine. The label (i.e. the supervisory signal) for each example is the number of

runtime issues that were detected in the sub-tree under the state that the example belongs to.

We defined five attributes as predictors that can be determined for each state. Attributes may containe both static source-code information and dynamic information that the symbolic execution supplies.

The attributes are the followings:

1. *The depth of the state in the symbolic execution tree.* If there is a tendency of in which depth the significant part of the faults occur, the information will be encoded into the model.

2. *The number of null values in the state*, as described previously.

3. *The sum of the number of zero numeric type values (variable values, literals, function return values, etc.) in the state, and the number of division operators reachable from the state according to the control flow graph in 15 basic block depth.* Here, we combined the dynamic information of zero values and the static information of the number of division operators in the possible future of the execution. This attribute is a heuristic for finding division-by-zero errors.

4. *The Logical Lines of Code (LLOC) metric of the method that the state belongs to.*

5. *The cyclomatic complexity metric [63] of the method that the state belongs to.*

As lines of code (LOC) and cyclomatic complexity have proved to be promising defect predictors [65, 66], attributes 4 and 5 should be useful in our heuristic. Both of them were calculated using static source code analyzer tool, called *SourceMeter*.

We applied the linear regression algorithm implemented in the *Shark* machine learning library [44].

5.5 Results

5.5.1 Optimal Maximum Depth and State Number

In order to ascertain the optimal limitations of the symbolic execution tree built by RTEHunter with the goal of finding runtime issues in a minimal time frame, we performed numerous analyses and applied different constraints.

First of all, we found that the number of executed states is closely correlated with the analysis time. The Pearson-correlation coefficients are all above 0.99, which is a strong positive correlation, and it means that the high state limit corresponds to a high run-time. The result is significant at $p < 0.05$. However, the correlation coefficients among the number of states and the number of found issues range from 0.3 to 0.8 indicating a weaker relationship, and the results are not significant in many cases at $p < 0.05$. Hence the number of states seems to determine the execution time, but the number of errors probably depends on other factors as well.

To understand the role of maximum depth, we ran RTEHunter with different depth limits. At each depth limit level we used different maximum state sizes which allows us to investigate how the results vary by increasing the analysis time. The depth limits chosen were 50, 100, 200, 400, 600 and 800, at each depth limit the maximum state

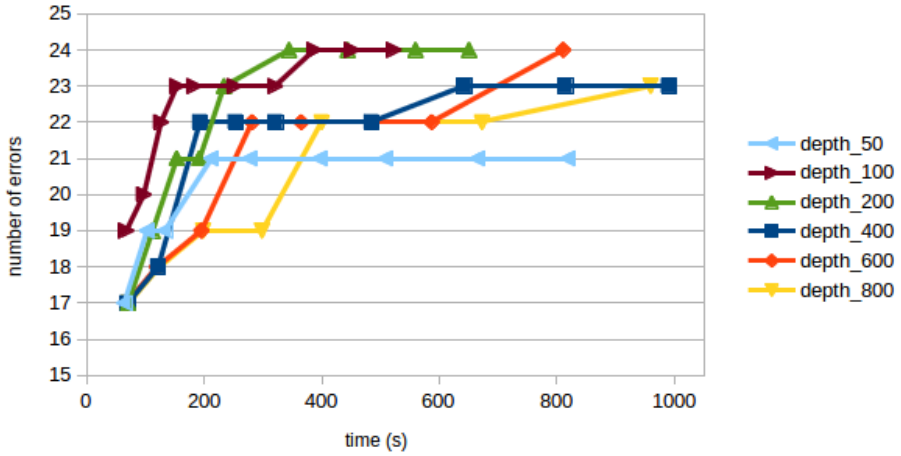


Figure 5.3. The increase in the number of errors at analysis time using different depth limits in ArgoUML

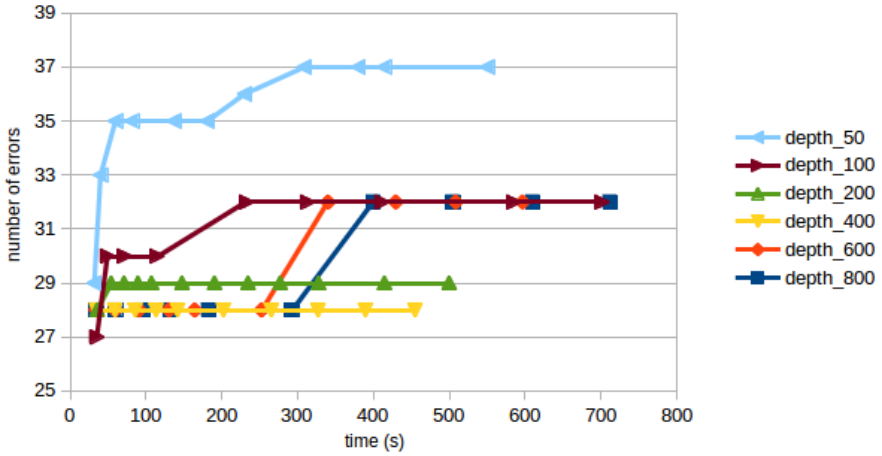


Figure 5.4. The increase in the number of errors at analysis time using different depth limits in Jetspeed

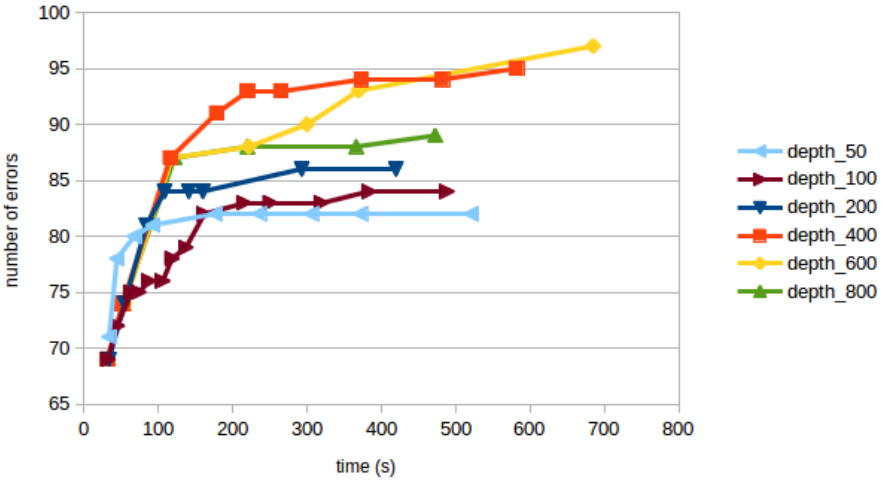
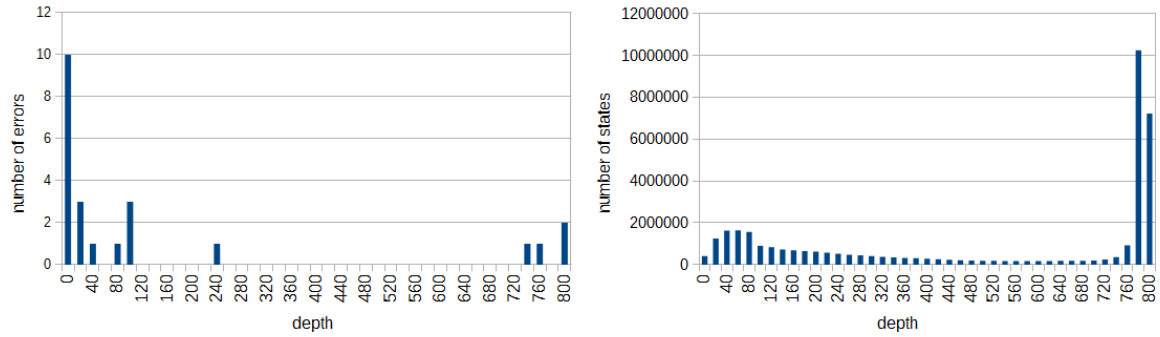
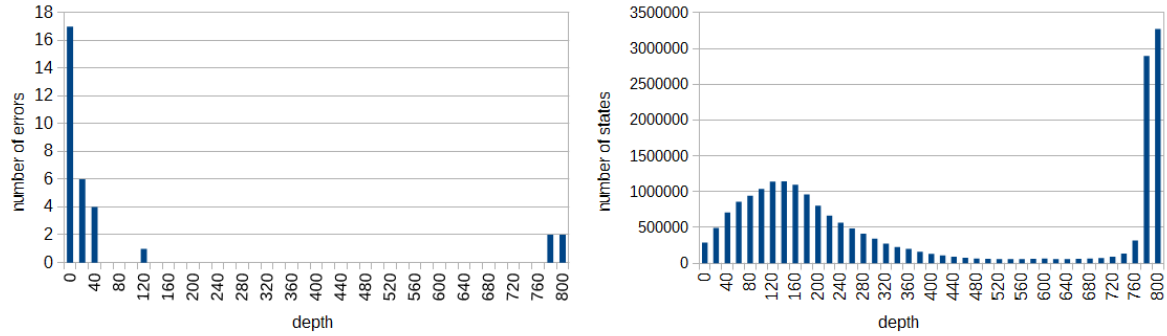


Figure 5.5. The increase in the number of errors at analysis time using different depth limits in JFreeChart



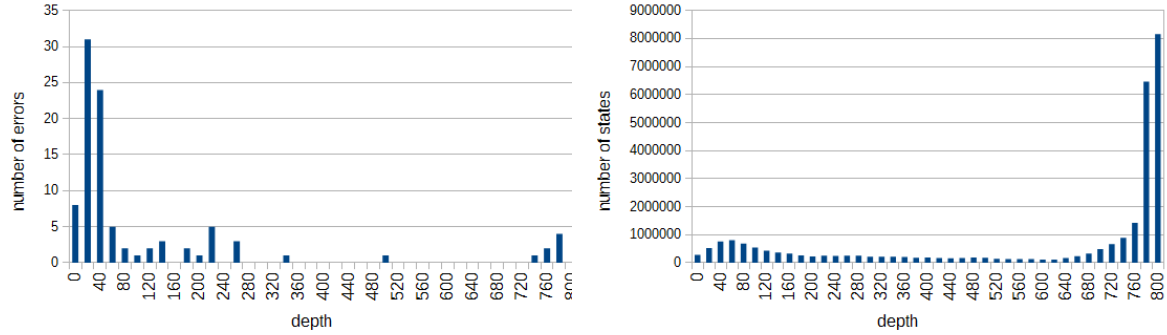
(a) (b)

Figure 5.6. (a) The error distribution at different depth levels in ArgoUML. (b) The overall state distribution at different depth levels in ArgoUML



(a) (b)

Figure 5.7. (a) The error distribution at different depth levels in Jetspeed. (b) The overall state distribution at different depth levels in Jetspeed.



(a) (b)

Figure 5.8. (a) The error distribution at different depth levels in JFreeChart. (b) The overall state distribution at different depth levels in JFreeChart

values differ from 200 to 10,000. The results were put onto line diagrams shown in figures 5.3, 5.4 and 5.5 for ArgoUML, Jetspeed and JFreeChart respectively. The diagrams show how the number of errors grows in time for each depth limit level. On each line, the dots represents an RTEHunter run with a specific state number limit. For the consecutive runs we used higher and higher state number limits (between 200 and 10,000) extending the analysis time. For each line, the last dot is formed to extend the analysis at least nearly 500 seconds.

In general, each line starts increasing, and after a while the number of errors does not increase anymore. The reason why the number of errors stagnate after a time may be because all of the errors were found at that depth level, and to detect new ones the analysis needs to go deeper. The depth limit is considered to be better which rises more rapidly and with which the engine detects more errors. The best depth limit varies from system to system.

As regards ArgoUML, the 100-depth configuration increases the most rapidly, but the 200-depth one finds 24 errors in slightly shorter time. By decreasing the depth limit to 50, the results get worse, and the 400- 600- and 800-depth configurations also perform worse.

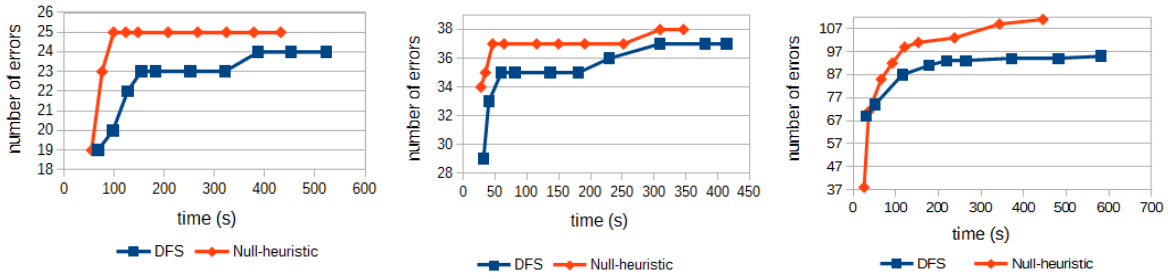
With Jetspeed, the 50-depth limit is far better than the others. While with the depths of 100, 600 and 800 we can reach only 32 errors under 500 seconds, we managed to detect 37 errors with the 50-depth configuration. These numbers imply that the majority of the errors are below the depth level of 50 in Jetspeed.

Among the investigated depth limits, the 400-depth can be considered to be the optimum in the case of JFreeChart. However, the 50-depth one starts to rise the most rapidly, it stops growing after 100 seconds. After 600 seconds the 600-depth limit becomes slightly better than the 400-depth, but in general the 400-depth one performs the best.

To understand why the above-mentioned depth limits performed the best in the experiments, we analyzed how many errors can found at different depth levels in general. Subfigures (a) of Figures 5.6, 5.7 and 5.8 show the number of errors found at each depth level. Each bar represents a depth interval formed to be 20-length. The height of a bar represents the number of errors found in the particular depth interval. The data is derived from an 800 depth and 15000 state limit run, with which we attempt to analyze as big symbolic execution trees as the memory consumption made possible.

In general, the same pattern appears to be present in all three systems. The number of errors are significant at shallower levels, then in the middle where just a few of them were detected. Close to the overall depth limit (800) errors occur again but not at such high numbers as in the shallower levels. E.g. the error distribution of Jetspeed in Figure 5.7 (a) tells us that the majority of the errors were found at a depth limit of 40 or less, which explains why the 50-depth limit is so satisfactory in Figure 5.4. Although deeper configurations reached more states, the error density is rather low there, hence we wasted the time spent to execute the states here.

We have also plotted the distributions of the overall number of states that were explored by RTEHunter in figures 5.6, 5.7 and 5.8 (b). These diagrams show the global shape of the symbolic execution trees traversed through the analysis. Similar pattern can be seen in the error distribution diagrams, which partially explains the error distribution: at depth levels where more states are explored, more errors can be found. However, there are many more states near the 800-depth limit than in the shallower parts of the tree, but the number of errors are higher in the shallow levels than



(a) (b) (c)

Figure 5.9. The efficiency of null-heuristic search compared to the default depth-first search on ArgoUML (a), Jetspeed (b), JFreeChart (c)

at the bottom. This leads us conclude that the the runtime issues that RTEHunter can detect are more common between levels 0 to 60 basic block depths compared to the deeper levels in general.

It should be added that the search strategy which is used to explore the state space has a marked effect on the state distribution, and hence on the error distribution too.

5.5.2 Null-heuristic Search Strategy

The goal of the null-heuristic search strategy is to increase the number of runtime issues detected in the given time frame, compared to the default depth-first search. Especially, we focused on the number of null pointer dereferences. To make a comparison, we used those configurations which are evaluated to be the best in Section 5.5.1 as a reference. The maximum depth for ArgoUML is 100, 50 for Jetseed and 400 for JFreeChart. We also chose the same state number limit sequence to expand the analysis time as before. With these parameters, but with our novel null-heuristic we repeated the experiments. The result of these experiments are shown in Figure 5.9.

In each case the null-heuristic approach performed better, because the number of detected issues increases more rapidly and the values higher, i.e. it found more runtime issues in less time, which surely confirm the efficiency of our algorithm. It is worth mentioning here that over 90% of the errors found in these systems are null pointer dereferences, and the new search strategy does not affect this ratio significantly.

What is more interesting is that the same analysis sequence with the null-heuristic approach finished in less time than before. E.g. in the case of ArgoUML the last analysis (the last dot on the line) lasted 431 seconds with the null-heuristic, and 522 seconds with the conventional DFS. This point is surprising because we need to calculate the number of null reference values for each state and also maintain a priority queue to keep the leaves in for the null-heuristic algorithm. Probably the reason why it is still faster is that it guides the search towards states whose execution time is shorter.

5.5.3 Linear Regression-Based Search Strategy

In the evaluation of the linear regression-based search strategy, we use 10-fold cross-validation on each system in the following way. Firstly, to form the folds we sort the methods of the system by lines of code (LOC). In the sorted list, each $j^{th} method$ is placed into $fold_i$ if equation $j \ mod \ 10 \ = \ i$ is satisfied. In other words, every tenth

		500 max state	1000 max state	1500 max state
ArgoUML	DFS	22	23	23
(max depth: 100)	LR based	51	54	55
Jetspeed	DFS	33	35	35
(max depth: 50)	LR based	66	74	73
JFreeChart	DFS	91	93	94
(max depth: 400)	LR based	122	131	138

Table 5.2. The number of detected runtime issues using the linear regression-based search strategy (LR based) compared to the default depth-first search (DFS)

method will go to the same fold (see Figure 5.10), ensuring that no fold differs too much from the others in the length of the methods contained.

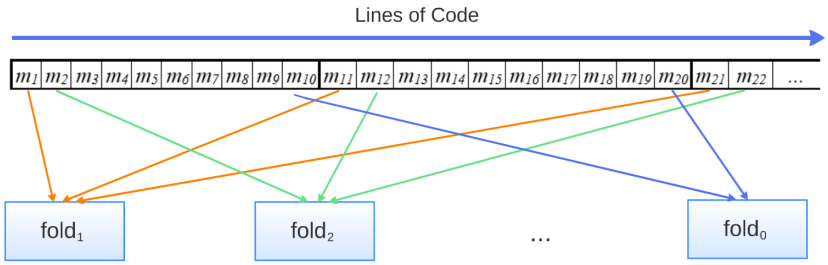


Figure 5.10. The formation of the folds used to perform 10-fold cross validation

After running RTEHunter on 10 folds, we summarized the number of errors that were detected in each fold. The structure of the folds ensures that each error is counted only once. The number of errors found using this strategy (LR-based) is shown in Table 5.2 and it is compared to the errors found by the default depth-first search strategy (DFS). With all the subject systems we set the depth limit that was found to be the best in Section 5.5.1 using DFS: 100 for ArgoUML, 400 for JFreeChart, and 50 for Jetspeed. As regards the maximum state number constraint, we provide results for 500, 1000 and 1500 maximum state values.

As the results demonstrate, our novel algorithm outperforms the default one in each case. In ArgoUML and Jetspeed, we found more than twice as many errors as we did with DFS. However, in the case of JFreeChart this ratio is smaller, the difference being still significant: it formed around 30 to 40 more issues were discovered.

The reason why we do not present the runtime here is due to the implementation details of the 10-fold cross validation. Currently, before RTEHunter starts the analysis of each fold, it has to load the ASG and then rebuild the CFG for architectural reasons. This introduces an overhead, which is not present in the case of a conventional run. Apart from this shortcoming, the difference in the number of detected issues is still significant.

In this work we did not put emphasis on the practical usage of the developed search strategies in a real life product considering which strategy is good for which error type. I.e. the null heuristic search might be great for finding null dereferences but also might degrade the quality of the detection of other issue types. In real-life usage one option would be to develop a specific well performing search strategy for each issue type. However, it can be time consuming to rerun the symbolic execution for each type. The second option would be to develop one complex search strategy with general predictors

that performs well for all or for most of the issues. This approach might consume less time, because we need to build up the state space only once, but might not scale well as the number of checks are increasing or not perform as well as the issue specific heuristics.

5.6 Threats to Validity

In the present study we do not focus on the precision of RTEHunter. Some of the issues found may be false positives and these may change or invalidate the result of our investigations. Manual validation needs a considerable amount of work in the case of such a high number of errors presented here. However, in a future work we plan to perform the manual validation and repeat the experiments with the updated dataset. We also intend to examine how the false positive rate is affected by the traversal strategy because we wish to avoid situations where a deeper analysis on a path produces more false positives, for instance because of an incorrectly handled coding pattern.

5.7 Summary

The goal of this study is to optimize the RTEHunter symbolic execution engine to detect more runtime issues faster. Because of the path explosion problem, the limitation of the state space of symbolic execution gets major importance in this scenario. The empirical investigations of three open-source Java systems showed that adjusting the maximum number of states for the symbolic execution trees directly impacts execution time but not the number of found issues. On the other hand, the limitations of the depth of the tree have more importance in the detection of runtime errors. We found different optimal depth limits for the three different systems, but we can conclude that errors occur more often in the basic block depth of 0 to 60 compared to the deeper levels in the analyzed systems, but it also highly depends on the applied search strategy.

We propose two novel search strategies that strive to guide symbolic execution towards the more error prone source-code fragments using both static and dynamic information. The null-heuristic search strategy performs better by finding up to 16 % more errors within the same time frame compared to the default depth-first search. The linear regression-based heuristic also outperforms DFS, it detects more than twice as many errors in ArgoUML and Jetspeed.

The author's contributions. The author performed the entire empirical analysis to find the connection between the maximum number of states for the symbolic execution trees, the analysis time and the number of issues found by running RTEHunter many times on three open source systems. He also performed many experiments to find the optimal depth limits for each system and revealed the depth level where most errors can be found. The idea of the two search strategies for the state space exploration, their implementation and their entire evaluation are the author's work. The publication related to this chapter is:

- **I. Kádár.** The Optimization of a Symbolic Execution Engine for Detecting Runtime Errors. In *Acta Cybernetica*, pages 573–597, 2017.

Part II

Investigation of Refactoring Activities Based on a New Dataset

"It would appear that we have reached the limits of what it is possible to achieve with computer technology, although one should be careful with such statements, as they tend to sound pretty silly in five years."

— John von Neumann

6

Assessment of Refactoring Activities on Classes and Methods Based on a New Public Dataset

6.1 Overview

Source code refactoring is a popular and powerful technique for improving the internal structure of software systems. The concept of refactoring was introduced by Fowler [29], and nowadays IT practitioners think of it today as an essential part of the development process. Despite the high acceptance of refactoring techniques by the software industry, there are some aspects that software companies should take into consideration as they might affect the practical application of these techniques (for example, time constraint, cost effectiveness, or return on investment). Due to this shift of priorities between industry and research, we should also explore how developers tend to use refactoring in practice and not merely focus on the theoretical concepts of code refactoring. Fowler proposed that code smells should be the primary technique for identifying refactoring opportunities in the code and a lot of research effort [28, 49, 62, 103] has been put into examining them. However, there is evidence in literature [4, 72, 111] that engineers are aware of code smells but are not concerned about their impact as refactoring activity is not focused on them. A similar counter intuitive result by Bavota et al. [7] suggests that only 7% of refactoring operations actually remove the code smells from the affected class. Besides exploring how, when and why refactoring is used in everyday software development, their effects on short and long-term maintainability and costs are vaguely supported by empirical results.

To help address further empirical investigations of code refactoring, we proposed a publicly available refactoring dataset [98] that we assembled using the *Ref-Finder* [52, 74] tool for refactoring extraction and the *SourceMeter* static source code analyzer tool for source code metric calculation. The dataset consists of refactorings and source code metrics for 37 releases of 7 open-source Java systems. We also store exact version and line information in the dataset to supports reproducibility. In addition to source code

metrics, the dataset includes the relative maintainability indices (RMI) of source code elements calculated by the *QualityGate*[1] tool, an implementation of the *ColumbusQM quality model* [5]. This makes it possible to directly analyze the connection between source code maintainability and code refactoring.

We applied the dataset for an analysis on the effects of code refactoring on source code metrics and maintainability by investigating the following research questions:

RQ1. *Are code elements with lower maintainability value subject to more refactorings in practice?*

RQ2. *Which quality attributes (source code metrics) are affected most by code refactorings and to what extent?*

Since the assembled dataset contains source code metrics, RMI, and refactoring values for classes and methods as well, we investigated the research questions at both class- and method-level.

At the level of classes, our results showed that classes with poor maintainability are subject to more refactorings in practice than classes with higher technical quality. Considering metrics, number of clone instances, complexity, coupling, and size metrics have improved, although comment related metrics decreased.

Literature lacks studies on the evolution of methods in systems due to refactorings, therefore, we examined them now by using the proposed dataset. We performed empirical investigations at the level of individual methods similarly to the level of classes. We found that lower maintainability indeed triggers more code refactorings in practice at the level of methods and these refactorings significantly decrease code lines, coupling, and clone metrics.

According to the authors of Ref-Finder, the precision of the tool is 79% [74], however, after our analysis, it is turned out that the quality of the refactoring data is lower due to the false positive instances. Hence, we propose an improved dataset that is a manually validated subset of our original dataset. It contains one manually validated release for each of the 7 systems. Although the manually validated refactoring dataset is in itself a major contribution, we also utilized it to replicate our studies on the base dataset and re-examine the connection between maintainability and code refactoring as well as the distribution of the individual source code metrics in the refactored and non-refactored source code elements. The results showed that the overall average maintainability of refactored entities was much lower in the pre-refactoring release than the entities subjected to no refactorings, which is in line with the results we got in the earlier research. The analysis on the source code metrics has not been accepted and published until the author finished writing this thesis, thus we only give a brief overview of the findings in Section 6.7.2.

The rest of the chapter is organized as follows. First, we start with a related literature overview in Section 6.2. Next, Section 6.3 outlines the data collection and validation process of creating the datasets. We describe the data analysis methodology applied for answering the research questions in Section 6.4. In Section 6.5, we display the results of our empirical investigation on the maintainability and source code metrics of refactored and non-refactored classes, and Section 6.6 gives the method-level results on the unvalidated dataset. We describe the assessment of the manually validated dataset in Section 6.7, the threats to the validity of our results are listed in Section 6.8, and finally, we conclude the research in Section 6.9.

[1] `http://www.quality-gate.com/`

6.2 Related Work

There are several studies that have investigated the relationship between practical refactoring activities and the software quality through different quality attributes. Many of them used the Ref-Finder tool [52] to extract refactorings from real-life open-source systems, similarly as we did.

Bavota et al. [7] made observations on the relations between metrics/code smells and refactoring activities. They mined the evolution history of 2 open-source Java projects and revealed that refactoring operations are generally focused on code components for which quality metrics do not suggest there might be a need for refactoring operations. In contrast to this work, by considering maintainability instead of code smells, we found significant and quite clear relationship with refactoring activities. Bavota et al. also provided a large refactoring dataset with 15,008 refactoring operations, but it contains file level data only without exact line information. Our open dataset contains method-level information as well and refactoring instances are completely traceable.

In a similar work to ours, Murgia et al. [68] studied whether highly coupled classes are more likely to be targets of refactoring than less coupled ones. Classes with high fan-out (and relatively low fan-in) metric consistently showed to be targets of refactoring, implying that developers may prefer to refactor classes with high outgoing rather than high incoming coupling. Kataoka et al. [48] also focused on the coupling metrics to evaluate the impact of refactorings and showed that their method is effective in quantifying the impact of refactoring and helped them to choose the appropriate refactoring types.

Contrary to these two works [68, 48], we did not select a particular metric to assess the effect of refactorings, but rather used statistical tests to find those metrics that change meaningfully upon refactorings. This way we could identify that complexity and size metrics also play an important role in connection with refactorings applied in practice.

Kosker et al. [54] introduced an expert system for determining candidate software classes for refactoring. They focused on the complexity measures as primary indicators for refactoring and built machine learning models that can predict whether a class should be refactored or not based on its static source code metrics. In lack of real refactoring data, they assumed that classes with decreasing complexity over the releases are the ones being refactored actively. Using this heuristic, they were able to build quite efficient prediction models.

Although it might seem that our work is very similar to that of Kosker et al., there are numerous differences. We mined and manually verified real refactoring instances instead of using heuristics to determine which classes are refactored. We also analyzed the values of static source code metrics of the refactored and non-refactored elements, but our focus was not on selecting the best predictors for building machine learning models, but to generally explore the connection between each and every metric and refactorings. Moreover, we examined 50+ metrics, which is almost the double that Kosker et al. used and also contain for example, cohesion and clone related metrics that were not examined by them. Furthermore, we applied a statistical approach instead of machine learning, and published results at the level of methods as well, not just for classes as Kosker et al did. A significant part of our work was dedicated to the analysis of the changes in metric values that was entirely omitted by Kosker et al.

In the study conducted by Silva et al. [82] the authors monitored Java projects

on GitHub and asked the developers to explain the reasons behind their decision to refactor the code. They composed a set of 44 distinct motivations of 12 refactoring types such as "Extract reusable method" or "Introduce alternative method signature" and found that refactoring activity is mainly triggered by changes in the requirements and much less by code smells. The authors also made the collected data and the tool called RefactoringMiner publicly available, which was used to detect the refactorings.

The case study by Ratzinger et al. [78] investigated the influence of refactoring activities on software defects. The authors extracted refactoring and non-refactoring related features that represent several domains such as code measures, team and co-change aspects, or complexity that served as input to build prediction models for software defects. They found that the number of software defects decreased if the number of refactorings increased in the preceding time period.

Similarly to us, Murphy-Hill et al. [69] empirically analyzed how developers refactor in practice. They found that automatic refactoring is rarely used: 11% by Eclipse developers and 9% by Mylyn developers. Unlike this paper, we did not focus on how refactorings are introduced (i.e. manually or using a tool), but rather on their effect on source code.

Negara et al. [70] conducted an empirical study considering both manual and automated refactoring. Using a continuous refactoring inference algorithm, they composed a corpus of 5,371 refactoring instances collected from developers working in their natural environment. According to their findings, more than half of the refactorings were performed manually, more than one third of the refactorings performed by developers were clustered in time, and 30% of the applied refactorings did not reach the version control system.

The approach presented by Hoque et al. [42] investigates the refactoring activity as part of the software engineering process and not its effect on code quality. The authors found that it is not always true that there are more refactoring activities before major project release dates than after. The authors were able to confirm that software developers perform different types of refactoring operations on test code and production code, specific developers are responsible for refactorings in the project and refactoring edits are not very well tested.

Tsantalis et al. [102] identified that refactoring decision-making and application is often performed by individual refactoring "managers". They found a strong alignment between refactoring activity and release dates and revealed that the development teams apply a considerable amount of refactorings during testing periods.

Measuring clones (code duplications) and investigating how refactoring affects them has also attracted a lot of research effort. Our dataset also includes clone metrics, thus clone oriented refactoring examinations can also be performed.

Choi et al. [18] identified that merged code clone token sequences and differences in token sequence lengths vary for each refactoring pattern. They found that "Extract method" and "Replace method with method object" refactorings are the most popular when developers perform clone refactoring.

Choi et al. [17] also presented an investigation of actual clone refactorings performed in open-source development. The characteristics of refactored clone pairs were also measured. From the results, they again confirmed that clone refactorings are mostly achieved by "Replace method with method object" and "Extract method".

We found that refactoring activities are not related to clone metrics significantly in general. However, we did not distinguish our analysis based on the types of refactorings

(due to the relatively small number of true positive refactoring instances), which might introduce new results for specific refactoring types that differ from the overall case.

An automated approach to recommend clones for refactoring by training a decision tree-based classifier was proposed by Wang et al. [107]. The approach achieved a precision of around 80% in recommending clone refactoring instances for each target system, and similarly good precision is achieved in cross-project evaluation. By recommending which clones are appropriate for refactoring, the approach allows for better resource allocation for refactoring itself after obtaining clone detection results.

Fowler informally linked bad code smells to refactorings and according to Beck, bad smells are structures in the code that suggest refactoring [29]. Despite that many studies showed that practitioners apply code refactoring differently, probably the most widespread approach in the literature to detect program parts that require refactoring is still the identification of bad smells.

Tourwé and Mens recommended a semi-automated approach based on logic meta programming to formally specify and detect bad smells and to propose adequate refactorings that remove these bad smells [101]. Another approach to point out structural weaknesses in object-oriented programs and solve them in an automated fashion using refactorings was proposed by Dudziak and Wolak [25]. Tahvildari and Kontogiannis proposed a framework in which a catalog of object-oriented metrics was used as indicators to automatically detect where a particular refactoring can be applied to improve software quality [91]. Szőke et al. [90] introduced a tool called FaultBuster that identifies bad code smells using static source code analysis and automatically applies algorithms to fix selected code smells by refactoring.

Although Ref-Finder can detect 63 refactoring types from Fowler's catalog and many studies used it to extract refactorings [17, 20, 57, 34], there are other approaches for refactoring detection in practice. A method by Godfrey and Zou [38] identified merge, split and rename refactorings using extended origin analysis in procedural code, which served as a basis of refactoring reconstruction by matching code elements. Demeyer et al. [24] proposed an approach that compares two program versions based on a set of lightweight, object-oriented metrics such as method size, class size, and the number of method calls within a method to detect refactorings. Rysselberghe and Demeyer exploited also clone detection to detect move refactorings [104]. Xing et al. [110] presented an approach by analyzing the system evolution at the design level. They used a tool called UMLDiff to match program entities based on their name and structural similarity. However, the tool did not analyze method bodies, so it did not detect intra-method refactoring changes, such as a 'Remove Assignment To Parameter'.

The survey by Soares et al. [85] compared different approaches to detect refactorings in a pair of versions. They performed comparisons by evaluating their precision and recall in randomly selected versions of JHotDraw and Apache Common Collections. The results showed that Murphy-Hill [69] (manual analysis) performed the best, but was not as scalable as the automated approaches. Ratzinger's approach [78] is simple and fast, but it has low recall; SafeRefactor [84] is able to detect most applied refactorings, although they get low precision values in certain circumstances. According to experiments, Ref-Finder has a precision of around 35% and a recall of 24%, which is similar to our evaluation results.

A history querying tool called QWALKEKO [89] was also applied to the problem of detecting refactorings. The main difference between QWALKEKO and Ref-Finder is that Ref-Finder is limited to reason about two predefined versions while QWALKEKO

is able to detect refactorings that happen across multiple versions. Besides the ones presented above, many other approaches exist in the literature [3, 32, 59, 86, 92], however, our focus is not on refactoring miner tools, but to utilize refactoring instances found by those tools to analyze their connection with software maintainability in practice.

6.3 Dataset Construction

In order to support empirical research on source code refactorings, we built a dataset of the applied refactorings and source code metrics between two subsequent releases of 7 open-source Java systems available on GitHub. In our later research we assembled the manually validated subset (from now on *improved dataset*) of the mentioned original dataset (from now on *base dataset*) [98]. Table 6.1 provides an overview of the projects, their names, URLs, number of analyzed releases and the covered time interval by the releases in the base dataset. These projects were found ideal for our research purposes because of the adequate number of release versions and the amount of the code modifications between two adjacent releases. We investigated 3 to 8 releases of each project.

System	Git URL	# Rel.	Time interval
antlr4	`https://github.com/antlr/antlr4`	5	21/01/2013-22/01/2015
junit	`https://github.com/junit-team/junit`	8	13/04/2012-28/12/2014
mapdb	`https://github.com/jankotek/MapDB`	6	01/04/2013-20/06/2015
mcMMO	`https://github.com/mcMMO-Dev/mcMMO`	5	24/06/2012-29/03/2014
mct	`https://github.com/nasa/mct`	3	30/06/2012-27/09/2013
oryx	`https://github.com/OryxProject/oryx`	4	11/11/2013-10/06/2015
titan	`https://github.com/thinkaurelius/titan`	6	07/09/2012-13/02/2015

Table 6.1. Descriptive statistics of the systems included in the refactoring base dataset

To reveal refactorings between two adjacent release versions we used the Ref-Finder [52] refactoring reconstruction tool. In order to use Ref-Finder to automatically extract refactorings not just between two adjacent versions of a software but between each of the versions in a given version sequence we improved Ref-Finder to be able to perform an automatic batch analysis. To make further examinations possible, we also implemented an export feature in Ref-Finder that writes the revealed refactorings and all of their attributes into CSV files for each refactoring type.[2] The base dataset is composed of the output of Ref-Finder grouped by refactoring types (e.g. extract method, remove parameter) and the more than 50 types of source code metrics extracted by the SourceMeter static code analysis tool, mapped to the classes and methods of the systems. The full list of extracted source code metrics is available on the tool's website.[3] Instead of selecting several metrics to analyze, we applied all the statistical methods on each of the provided metrics, which include all the most widely used code metrics.

The refactoring types are different at the class- and method-levels: there are 23 refactoring types at class-level, and 19 at method-level. For a complete list of method and class-level refactorings see Table 6.2.

Beyond the plain source code metrics the datasets include the so-called *relative maintainability index* (RMI) which was measured by QualityGate SourceAudit [6] for

[2]The corresponding code changes can be found in a pull-request to the original repository: `https://github.com/SEAL-UCLA/Ref-Finder/pull/1`

[3]`https://www.sourcemeter.com/resources/java/`

Refactoring type	**Class level**	**Method level**
Add parameter	✓	✓
Consolidate conditional expression	✓	✓
Consolidate duplicate conditional fragments	✓	✓
Extract method	✓	✓
Inline temporary variable	✓	✓
Introduce assertion	✓	✓
Introduce explaining variable	✓	✓
Remove assignment to parameters	✓	✓
Remove parameter	✓	✓
Rename method	✓	✓
Replace magic number with constant	✓	✓
Replace method with method object	✓	✓
Inline method	✓	✓
Introduce null object	✓	✓
Remove control flag	✓	✓
Replace exception with test	✓	✓
Replace nested condition with guard clauses	✓	✓
Hide method	✓	✓
Replace temporary variable with query	✓	✓
Move field	✓	
Extract superclass	✓	
Extract interface	✓	
Introduce local extension	✓	

Table 6.2. The type of refactorings extracted by RefFinder at class and method level

each method and class of the systems. RMI, similarly to the well-known maintainability index [71], reflects the maintainability of a code element, but it is calculated using dynamic thresholds from a benchmark database, not by a fixed formula. Thus, RMI expresses the maintainability of a code element compared to the maintainability of other elements in the benchmark [40].

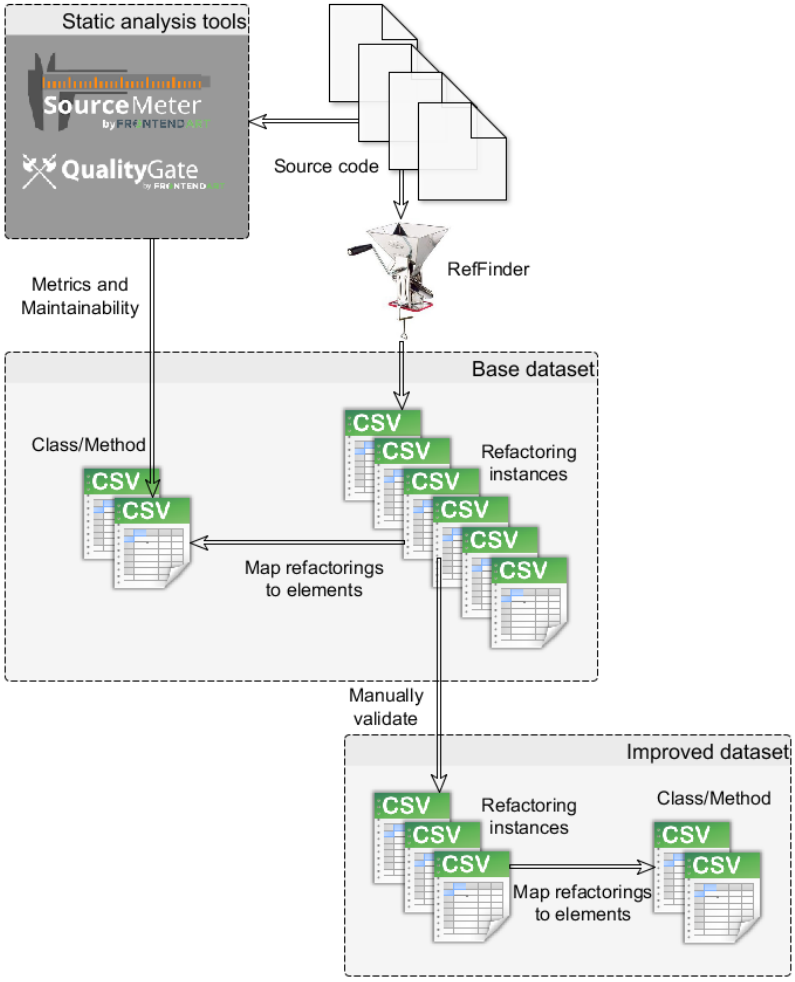


Figure 6.1. An overview of the process applied for constructing the base and the improved datasets

The high-level overview of the dataset creation process is shown in Figure 6.1. First of all, the Java source code is processed by the extended Ref-Finder version that reveals and exports the refactoring instances to CSV files for each refactoring type. The source code is also analyzed by the SourceMeter and QualityGate tools to calculate the source code metrics and RMI values for each method and class of the input project. The base dataset is assembled by mapping the extracted refactorings to the affected code elements and extending the output of static code analysis with the number of refactorings for each type that is mapped to the element.

To compose the improved dataset we performed a manual validation that resulted in a subset of the refactoring instances detected by Ref-Finder (i.e. we left only the true positive instances), then we mapped this validated subset of refactoring instances to the code elements again.

The datasets are available in the *PROMISE* data repository [64]:
`http://openscience.us/repo/refactoring/refact.html`
`http://openscience.us/repo/refactoring/refact_val.html`

6.3.1 Dataset Validation

As false positive instances may seriously affect the validity of empirical investigations using the dataset, we decided to manually validate the refactoring instances extracted by Ref-Finder and propose an improved dataset. Since it requires an enormous amount of human effort, we started by selecting one release from each of the 7 systems and validated every refactoring instance candidate proposed by Ref-Finder. The releases were selected to contain as many different types of refactorings as possible. We also kept in mind that the number of refactorings within each type has to be large enough in the releases given that some of them will be marked as false positives. We did not choose releases with huge amount of refactorings due to the necessity of an enormous validation effort.

Note, that we made a compromise in selecting the refactoring instances for validation. We chose to evaluate all instances between two selected releases for each of our subject systems. This resulted in an uneven proportion of validated refactorings from system to system (e.g. we evaluated almost 58% of refactoring instances for *oryx*, but only about 2% for *titan*), see Table 6.3. Moreover, there are refactoring types from which we did not evaluate a single instance, Table 6.4 lists only those refactoring types that were encountered during manual validation (Ref-Finder is able to extract 23 different types of refactorings [99]). The reason why we did this contrary to choosing for example, a fixed $x\%$ of refactoring instances for evaluation, is that it would not allow us to answer our research questions meaningfully. Validating a fixed proportion of refactorings for each system would not ensure a fully validated release for each system, instead we would end up with releases containing refactoring instances from a couple of which are manually validated and the rest are not. Analysis on such a dataset would be by no means more precise than using the base dataset, as the unvalidated instances might bias the statistical tests performed on the data between two releases of a system. As the manually validated subset of refactoring instances for analyzing our research questions is meaningful only if we have at least one fully validated release for each system, we made this compromise.

The validation was carried out by two researchers. One of them is the author of this thesis. Unfortunately, performing the evaluation in an optimal way, namely to examine all the possible refactoring instances by both of the evaluators, was not feasible due to our available resources. Instead, we distributed the refactorings between each other nearly equally and we validated only our corresponding instances. This strategy reduced the amount of required human resources to half of the optimal strategy; however, it also introduced some issues. To mitigate the possible inconsistency in the judgment of the evaluators, we performed a random sample cross-validation on about 10% of each other's data. Additionally, in each and every problematic case all the authors of the published paper [95] (not just the two evaluators) mutually agreed on how those specific refactorings should be classified.

Table 6.3 shows the total number of refactoring instances found by Ref-Finder in all the releases of the systems (# All), the selected revisions for manual validation (Release), the number of manually validated instances per system (# Eval.), the number

of true/false positive refactoring instances (TP and FP) and the overall precision of Ref-Finder on the analyzed systems (Prec.).

System	# All	Release	# Eval.	TP	FP	Prec.
antlr4	269	30/06/2013 [3468a5f]	112	50	62	44.64%
junit	1,080	08/04/2010 [a30e87b]	29	14	15	48.28%
mapdb	4,547	30/07/2014 [967d502]	171	4	167	2.34%
mcMMO	448	11/07/2013 [4a5307f]	63	6	57	9.52%
mct	716	27/09/2013 [f2cdf00]	97	28	69	28.87%
oryx	123	11/04/2014 [0734897]	71	25	46	35.21%
titan	3,661	13/02/2015 [fb74209]	84	18	66	21.43%
Total	**10,844**	–	**627**	**145**	**482**	**23.13%**

Table 6.3. Number of all and manually validated refactorings with precision information for each subject system

The evaluated release means that the refactoring instances between this and the previous release was considered for validation. As can be seen, only the fraction of the total number of refactorings has been validated (less than 6%). Even this work took more than one person month work from the two evaluators. However, as the overall precision of the Ref-Finder tool was only around 23% in total (and approximately 27% if we take the average of the system-wise precision values) on the base dataset, even these few hundred manually validated instances of the improved dataset bear a significant additional value compared to the base dataset. Considering the projects, we got the lowest precision value in case of mapdb and mcMMO resulting a relatively low number of refactorings in these projects.

Table 6.4 summarizes the number of various refactoring types within each subject system. As can be seen, Add and Remove Parameter are the two most frequently applied refactorings types. Together with the third most common Introduce Explaining Variable, they constitute nearly 60% of the total refactoring count. The majority of the Add Parameter refactoring is in the antlr4 system, while most of the Remove Parameter refactorings appear in oryx.

Refactoring Type	antlr4	junit	mapdb	mcMMO	mct	oryx	titan	Total
Add Parameter	22	2	0	1	11	1	2	**39**
Remove Parameter	2	0	0	0	4	18	5	**29**
Introduce Explaining Variable	6	0	2	0	3	4	2	**17**
Extract Method	4	4	0	2	0	0	0	**10**
Introduce Assertion	2	1	0	0	3	0	4	**10**
Rename Method	0	2	0	1	2	0	4	**9**
Replace Method with Method Object	8	0	0	0	0	0	1	**9**
Inline Temp	0	1	0	1	4	1	0	**7**
Move Method	3	2	1	0	0	0	0	**6**
Move Field	2	1	0	0	0	0	0	**3**
Extract Interface	0	0	1	0	1	0	0	**2**
Inline Method	0	1	0	1	0	0	0	**2**
Remove Assignment to Parameters	1	0	0	0	0	0	0	**1**
Replace Magic Number with Constants	0	0	0	0	0	1	0	**1**
Total	**50**	**14**	**4**	**6**	**28**	**25**	**18**	**145**

Table 6.4. Total number of refactoring occurrences in the improved dataset grouped by their types

6.3.2 Dataset Structure

The datasets contains one folder for each release of the analyzed systems. Within each folder there are two files (*$proj-Class.csv* and *$proj-Method.csv*) and a sub-folder containing a list of CSV files named by refactoring types. The CSV files with the names of refactorings (e.g. ADD_PARAMETER) lists the refactoring instances found by RefFinder. The improved datset contains only the true positive refactoring instances that were manually checked by one of the evaluators. The structure of these CSV files may differ based on the refactoring types, but they always contain enough information to uniquely identify the entities affected by the refactoring in the previous and actual releases (e.g. unique name, path of classes/methods, parameters or line information).

The *$proj-Class.csv* and *$proj-Method.csv* files hold an accumulated result of the above. Each line of these CSV files represents a class or method in the system (identified in the same way as in the refactoring CSVs). In the columns of the CSV, there are the source code metrics with the RMI scores and the various refactoring types. For each row we have the source code metrics calculated for this element and the number of refactorings of a certain type affecting the source code element (i.e. the source code element appears in the refactoring type CSV in an arbitrary role).

6.4 Data Analysis Methodology

This section describes the way we utilized the constructed refactoring dataset to answer our research questions. We applied the same statistical methods at class- and method-level, thus in this section we will use the general term *code element* to denote both classes and methods.

For answering RQ1, we performed a correlation analysis on the RMI values of the code elements and the number of refactorings affecting these elements. We took the RMI values from release x_i, and the number of refactorings from release x_{i+1}. This way we assessed whether poor quality code elements got refactored more intensively than others or not. Since we cannot assume anything about the distribution of the maintainability indices nor the number of refactorings, we performed a Spearman rank correlation analysis. In our latest research which assesses the improved dataset, we used a slightly different analysis methodology from the above to answer RQ1, because of the low number of the remaining refactorings. This difference will be highlighted in Section 6.7.

For answering RQ2, first we calculated the differences of the metric values between the subsequent releases. In most cases negative differences mean an improvement, as lower metric values (e.g. lower complexity) are better. To decide whether there is a significant difference among the metric decreases in the refactored and non-refactored classes, we run a Mann-Whitney U test [61], which is a non-parametric statistical test to analyze whether the distribution of the values differ significantly between two groups. The p-value of the test helped us judging whether there is significant difference in the metric decreases between the source code entities subjected to refactoring and the entities unaffected by refactoring.

The result of this test gave us a hint on what are those metric values that improve significantly upon refactorings. To estimate the volume of these metric changes, we calculated the Cliff's delta (δ) effect size measure as well [41]. Cliff's δ measures how often the values in one distribution are larger than the values in a second distribution.

It ranges from -1 to 1 and is linearly related to the Mann-Whitney U statistic, however it captures the direction of the difference in its sign as well. Simply speaking, if Cliff's δ is a positive number, the metric value differences (thus the metric value decreases) are higher in the refactored code elements, while negative value means that the metric value differences are higher in the non-refactored elements. The closer the $|\delta|$ is to 1, the more values are larger in one group than the values in the other group. Besides the Cliff's δ measure, in the class-level assessment of the base dataset (Section 6.5) we calculated the odds ratio (OR) effect size measure as well [67], which denotes how many times the chance is higher of that the values are differ in the two groups.

6.5 Results of the Class-level Assessment on the Base Dataset

In this section we summarize the assessment results of the assembled base refactoring dataset regarding software maintainability at the level of classes. First, we describe the results of the analysis on the maintainability of refactored classes to answer RQ1. Afterwards, we present the findings on the effect of refactorings on source code metrics to answer RQ2.

6.5.1 The Maintainability of Refactored Classes

To answer RQ1, we performed a correlation analysis between the number of refactorings affecting the classes and their maintainability indices in the previous release. Figure 6.2 depicts the Spearman correlation coefficients between the RMI values in release x_i and the number of refactorings affecting the corresponding classes in release x_{i+1}.

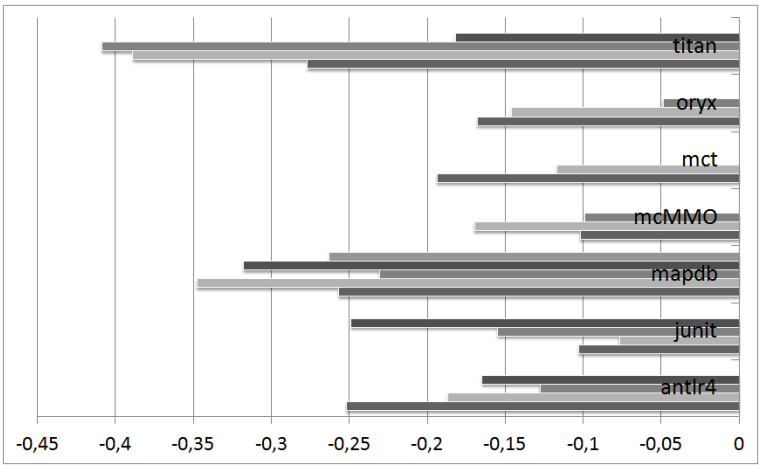


Figure 6.2. Correlation of maintainability and number of refactorings in classes

As can be seen, all the values are negative, meaning that the worse the maintainability of a class is the more refactorings touch it. Although the coefficients are moderate, they are consistently negative and significant at the level of 0.05 (except for the two lowest values of mcMMO and oryx). There are less correlation coefficients than releases for some systems because we were unable to calculate them when Ref-Finder found no refactorings between two releases, which happened a couple of times.

Answer to RQ1 at the level of classes: Based on the findings on our dataset it seems that classes with poor maintainability are subject to higher number of refactorings during their lifetime.

6.5.2 The Effect of Refactorings on Source Code Metrics

We found that refactorings affect poorly maintainable code, so the question arises whether applying refactorings really improves the internal quality of the code? And if yes, what are the source code metrics that show the highest improvement (i.e. decrease significantly)?

According to the process described in Section 6.4, we first calculated the metric value differences for every class between the adjacent releases. Then, we grouped these metric difference values into two groups: in the first group we put the metric differences of classes touched by at least one refactoring, and in the second group the metric differences of non-refactored classes. Finally, we analyzed which metrics show significant differences between the values of the two groups with the help of Mann-Whitney U test.

System name	CI	WMC	NOI	RFC	TCLOC	TLLOC	TNOS
antlr4	**0.033**	0.428	**0.010**	**0.031**	0.136	**0.002**	0.122
junit	0.728	**0.042**	0.170	N/A	**0.012**	0.101	0.113
mapdb	**0.030**	**0.006**	**0.005**	**0.000**	**0.05**	**0.000**	**0.000**
mcMMO	**0.005**	0.608	**0.003**	**0.013**	0.066	0.257	0.594
mct	0.905	0.200	N/A	0.941	N/A	0.115	0.703
oryx	0.667	0.575	0.381	0.533	0.800	0.743	0.159
titan	**0.022**	**0.016**	**0.000**	**0.000**	0.260	**0.002**	**0.042**

Table 6.5. The results of the Mann-Whitney U Test (p-values)

System name	CI		WMC		NOI		RFC	
	OR	δ	OR	δ	OR	δ	OR	δ
antlr4	9.38	-0.86	11.68	0.21	10.95	0.75	10.95	0.58
junit	6.04	0.13	20.14	0.69	5.31	0.35	7.07	0.06
mapdb	10.56	0.40	2.86	0.50	4.32	0.33	2.85	0.55
mcMMO	5.07	0.96	2.27	-0.22	3.57	0.89	2.80	0.79
mct	19.33	0.06	4.30	-0.89	5.95	0.00	2.42	0.13
oryx	6.17	-0.40	6.17	0.2	12.34	0.50	15.42	0.50
titan	3.76	0.39	5.53	0.15	4.92	0.24	4.57	0.30
Average	8.62	0.10	7.56	0.09	6.76	0.44	6.58	0.42

System name	TCLOC		TLLOC		TNOS	
	OR	δ	OR	δ	OR	δ
antlr4	5.47	0.61	2.09	0.64	6.97	0.40
junit	11.73	0.50	8.43	0.69	19.94	0.35
mapdb	8.43	0.30	2.30	0.78	2.51	0.78
mcMMO	4.63	0.63	3.77	0.30	2.82	0.19
mct	19.33	0.00	5.80	0.59	11.90	-0.15
oryx	7.12	0.13	3.59	-0.10	2.50	0.50
titan	4.98	0.15	4.43	0.26	4.95	0.17
Average	8.81	0.33	4.34	0.45	7.37	0.32

Table 6.6. Effect size measures

Out of 50+ source code metrics, the ones listed in Table 6.5 had the lowest p-values, meaning that the differences in the metric value changes for refactored and non-refactored classes are the most significant for these metrics.

To get an impression about the magnitude of the differences between the metric value decreases of the refactored and non-refactored classes, we calculated the ratio of

classes with metric value decreases within the two groups. The results are depicted on the heat map shown in Figure 6.3. The left columns contain the proportion of refactored classes having decreased metric value, while right columns show the same ratio for the non-refactored classes. The darker values mark higher ratios. As can be seen, all the dark values are in the left columns, thus metric value decreases are far more frequent in the group of refactored classes than in non-refactored classes.

To quantify what we observed visually in the heatmap, we calculated the odds ratio (OR) and Cliff's delta (δ) effect size measures. The detailed results are presented in Table 6.6. The average OR values vary between approximately 4-9, which means that on average the chances of a metric value decrease is 4-9 times higher in the classes affected by refactorings than in the non-refactored classes. The Cliff's δ values suggest a similar conclusion, though not as obviously as the OR values.

System name	CI		WMC		NOI		RFC		TCLOC		TLLOC		TNOS	
	R_{impr}	NR_{impr}	R_{impr}	NR_{impr}	R_{impr}	NR_{impr}	R_{impr}	NR_{impr}	R_{impr}	NR_{impr}	R_{impr}	NR_{impr}	R_{impr}	NR_{impr}
antlr4	3.57%	0.38%	9.52%	0.82%	7.14%	0.65%	8.33%	0.76%	3.57%	0.65%	9.52%	4.57%	8.33%	1.20%
junit	3.17%	0.53%	3.97%	0.20%	4.76%	0.90%	6.35%	0.90%	8.73%	0.74%	7.94%	0.94%	8.73%	0.44%
mapdb	13.46%	1.27%	10.58%	3.70%	16.35%	3.78%	14.42%	5.06%	18.27%	2.17%	11.54%	5.01%	10.58%	4.21%
mcMMO	20.00%	3.95%	20.00%	8.81%	30.00%	8.40%	30.00%	10.73%	30.00%	6.48%	50.00%	13.26%	30.00%	10.63%
mct	2.83%	0.15%	0.94%	0.22%	1.89%	0.32%	0.94%	0.39%	0.94%	0.05%	2.83%	0.49%	3.77%	0.32%
oryx	1.96%	0.32%	7.84%	1.27%	3.92%	0.32%	3.92%	0.25%	5.88%	0.83%	9.80%	2.73%	5.88%	2.35%
titan	2.69%	0.71%	11.98%	2.17%	25.41%	5.56%	25.41%	5.56%	5.79%	1.16%	14.05%	3.17%	14.46%	2.92%
Average	6.81%	1.04%	9.26%	2.45%	12.78%	2.85%	12.77%	3.38%	10.45%	1.73%	15.10%	4.31%	11.68%	3.15%

Figure 6.3. Metric improvements heat map

According to the above results of the statistical analysis, we can conclude that coupling metrics, namely Response Set for Classes (RFC) and Number of Outgoing Invocations (NOI) indeed decrease significantly upon refactorings in accordance with the previous findings of other studies [48, 68]. But besides coupling, we found a significant decrease in size metrics as well, namely in the case of Total Logical Lines of Code (TLLOC) and Total Number of Statements (TNOS). This finding is not really surprising, nor that the complexity metric Weighted Methods per Class (WMC) also decreased significantly. What is more interesting is that the number of Clone Instances (CI) also decreased, thus refactoring activity seems to remove copy-paste code parts in practice. Finally, an interesting result is that the Total Comment Lines of Code (TCLOC) also decreased significantly. This might mean a degradation in maintainability if the developer did not take the time to document the modifications, but it can also mean an improvement if out-of-date comments were removed, or even better, if the developer adhered to the clean code principle.

Generally, most of the Cliff's δ values are positive (i.e. the average δ values are positive for every metric), meaning that the metric value differences (the metric value decreases) are higher in the refactored classes, than in the non-refactored ones. Nonetheless, there are several large negative δ values for the CI and WMC metrics. This might suggest that cloned code and complexity is decreased by other targeted changes, while refactorings often have a side effect to remove code clones or reduce complexity as well. However, this phenomenon needs further investigation.

Answer to RQ2 at the level of classes: We found that size (TLLOC, TNOS), coupling (RFC, NOI), clone (CI), complexity (WMC) and comment (TCLOC) related metrics decrease the most in refactored classes. Regarding the volumes of the differences, we can say that for these metrics the average chances of a decrease is 4-9 times higher in the classes affected by refactorings than in the non-refactored classes.

6.6 Results of the Method-level Assessment on the Base Dataset

In this section we summarize the assessment results of the base datset on the connection between refactoring activity and maintainability of methods. First, we describe the results of the analysis on the maintainability of refactored methods to answer RQ1. Afterwards, we present the findings on the effect of refactorings on method-level source code metrics to answer RQ2.

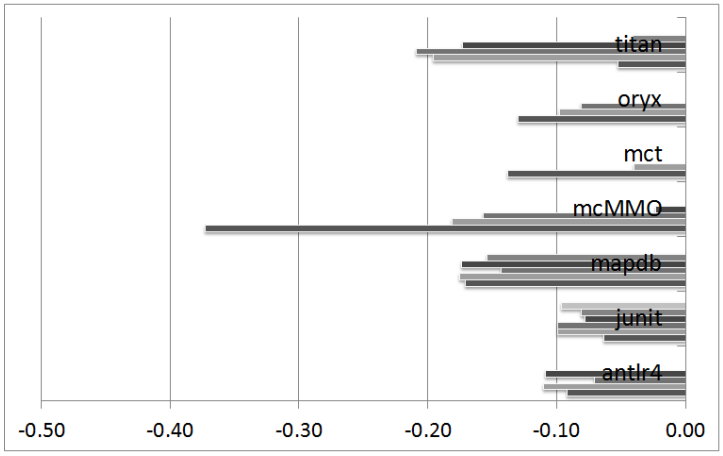


Figure 6.4. Correlation of maintainability and refactorings in methods

6.6.1 The Maintainability of Refactored Methods

To answer RQ1, we performed a correlation analysis between the number of refactorings affecting the methods of the subject systems and their maintainability indices in the previous release. Figure 6.4 depicts the Spearman correlation coefficients between the RMI values in release x_i and the number of refactorings affecting the corresponding methods in release x_{i+1}.

As can be seen, all the values are negative. Although the coefficients are not particularly high, they are consistently negative and significant at the level of 0.05. The negative values simply mean an inverse proportionality, namely that the worse the maintainability of a method or class is (the lower its RMI value) the more refactorings touch it (the higher the number of refactorings affecting it). There are less correlation coefficients than releases for some systems because we were unable to calculate them when Ref-Finder found no refactorings between two releases, which happened a couple of times. Table 6.7 summarizes the mean correlation coefficients both for method and class-level, their deviation and the number of evaluated intervals between releases. It can be noticed that the correlation coefficients and the deviations are somewhat larger in the case of classes, but the differences are negligible.

Answer to RQ1 at the level of methods: Based on the findings on our dataset we can conclude that methods with poor maintainability are subject to higher number of refactorings during their lifetime compared to those with better maintainability.

6.6.2 The Effect of Refactorings on Source Code Metrics

We found that refactorings affect poorly maintainable code more (i.e. methods), so the questions arises again: Whether applying refactorings really improves the internal

System	Method level			Class level		
	Mean corr.	Deviation	Intervals	Mean corr.	Deviation	Intervals
antlr4	-0.096	0.018	4	-0.183	0.052	4
junit	-0.086	0.014	6	-0.146	0.076	4
mapdb	-0.163	0.014	5	-0.283	0.048	5
mcMMO	-0.183	0.144	4	-0.124	0.040	3
mct	-0.089	0.069	2	-0.156	0.054	2
oryx	-0.103	0.025	3	-0.121	0.063	3
titan	-0.134	0.081	5	-0.314	0.106	4

Table 6.7. Average Spearman correlation coefficients between RMI and number of refactorings at method and class level

quality of the code? What are the method-level source code metrics that show the highest improvement (i.e. decrease significantly) upon refactoring?

As before, we first calculated the metric value differences for every method between the adjacent releases. Then, we grouped these metric difference values into two groups: in the first group we put the metric differences of methods affected by at least one refactoring, and in the second group the metric differences of non-refactored methods and finally, we analyzed which method-level metrics show significant differences between the values of the two groups with the help of the Mann-Whitney U test.

System name	CC	LLOC	NOS	NOI
antlr4	**0.049**	**0.000**	**0.002**	**0.001**
junit	0.058	0.923	0.667	0.403
mapdb	**0.010**	**0.003**	0.965	**0.002**
mcMMO	0.815	0.824	0.516	0.251
mct	0.703	0.924	0.547	0.660
oryx	0.654	0.555	0.306	1.000
titan	0.601	**0.016**	**0.003**	**0.000**

Table 6.8. The results of the Mann-Whitney U Test (p-values) for method-level metrics

The source code metrics listed in Table 6.8 had the lowest p-values, meaning that the differences in the metric value changes for refactored and non-refactored methods are the most significant for these metrics. We observed that the Number of Outgoing Invocations (NOI), which can be considered as a coupling metric indeed decreases significantly upon refactoring in accordance with the previous findings of other studies [48, 68].

But besides NOI, we found a significant decrease in size metrics as well, namely in the case of Logical Lines of Code (LLOC) and Number of Statements (NOS). These can be explained by the fact that typical refactorings, like extract method and pull up method, often have a side effect of reducing the amount of source code. This phenomena is clearly observable on these pure size related metrics.

While this finding is not really surprising, the fact that McCabe's cyclomatic complexity [63] did not show a significant correlation with the number of refactorings applied on methods is just the opposite of what we were expecting. Our perception was that using better code structures will lead to less complex code, but we could not

confirm this hypothesis. It is an even more interesting finding in the light of our previous results on the effect of refactorings on the Weighted Method Complexity (WMC) metric of classes, which shows a significant reduction upon refactorings. However, this is not a contradiction. Consider the *Extract method* refactoring for example. In this case duplicated methods in the child classes are extracted and put into their parent class, leading to the removal of the method from several classes and inserting it to their parent. On one hand, this yields to reduction in the average WMC metric as the complexity of child classes decrease, while only the complexity of their parent class increases. On the other hand, at method-level the average McCabe's complexity values do not change. So the above results might indicate that refactoring operations tend to decrease complexity at class-level, but not really at the level of methods.

It is interesting that the Clone Coverage (CC) metric also decreased, thus refactoring activity seems to remove copy-paste code parts in practice. This phenomena is similar to the code size reduction, e.g. by extracting common code snippets into a method reduces the copy-pasted code parts, too.

System name	CC	LLOC	NOS	NOI
antlr4	**0.70**	**0.63**	**0.48**	**0.71**
junit	-0.68	**0.01**	-0.08	**0.14**
mapdb	-0.34	**0.27**	0.00	**0.28**
mcMMO	**0.10**	**0.05**	**0.17**	**0.27**
mct	-0.15	-0.03	-0.18	-0.15
oryx	-0.18	-0.14	**0.31**	-0.02
titan	-0.09	**0.16**	**0.21**	**0.33**
Average	-0.09	**0.14**	**0.13**	**0.22**

Table 6.9. Cliff δ effect size measures for method-level metrics

To quantify the magnitude of the differences between the metric value decreases of the refactored and non-refactored methods, we calculated the Cliff's delta (δ) effect size measure again. The detailed results are presented in Table 6.9. If Cliff's δ is a positive number, the metric value differences are higher in the refactored methods, while negative value means that the metric value differences are higher in the non-refactored methods. Generally, the Cliff's δ values are quite hectic; however, the average δ values are positive for every metric except for CC. While in case of LLOC, NOS and NOI the majority of values are positive, only two projects have positive δ values for CC. This might suggest that cloned code is decreased by other targeted changes that are not refactorings, while refactorings often have a side effect to remove code clones as well (e.g. extract method).

To have a better overview of the above explained phenomena, we visualized the average size metric differences for the refactored and non-refactored methods in Figure 6.5. This boxplot clearly shows that the maximum, minimum, and average numbers of code line reduction are far smaller in case of methods that are not refactored than in the case of refactored methods. While the median of LLOC decrease is 2 in case of non-refactored methods, it is two times larger (around 4) for refactored methods. Based on these findings, we can now conclude RQ2.

Answer to RQ2 at the level of methods: We found that size (LLOC, NOS), coupling (NOI), and clone (CC) related metrics decrease the most in refactored meth-

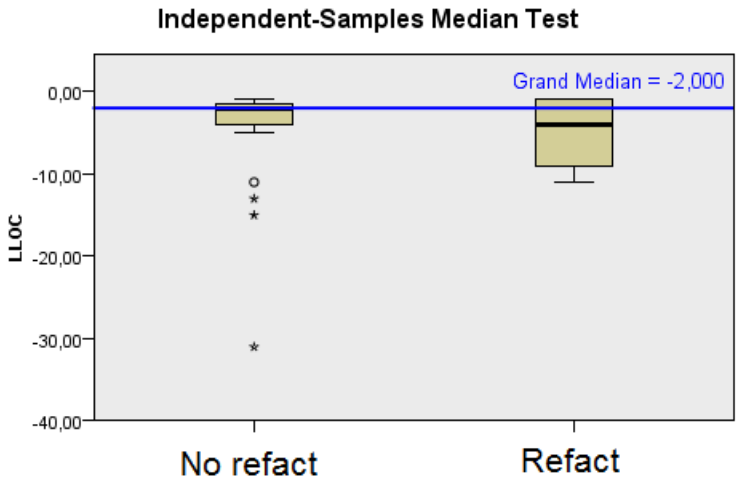


Figure 6.5. Boxplot of the LLOC metric decreses in the refactored and non-refactored methods

ods. Regarding the volumes of the differences, we can say that for these metrics the average Cliff's δ values are mostly positive suggesting a small to medium effect size on the metric decreases in the refactored methods compared to the non-refactored ones.

6.7 The Assessment of the Manually Validated Dataset

This section describes how we utilized the improved dataset and gives the findings regarding to software maintainability and source code metrics. However, we have done the research for answering RQ2 (which source code metrics are affected the most by refactoring and to what extent), the paper has not been accepted until the author finished writing this thesis. Thus, we give only a brief overview of our results regarding to the source code metrics and their change upon refactorings in Section 6.7.2.

6.7.1 Maintainability Analysis

To check whether source code elements with lower maintainability subject to more refactorings, we followed a method, which differs from the one described in Section 6.4.

Instead of correlation analysis, we used the Man-Whitney U test for the RMI values of code elements in the same way we applied it for metric value differences before to answer RQ2. More precisely, we took all the RMI values in release x_{i-1} for each system, where x_i is the release selected for manual validation. We formed two groups by RMI values based on the fact if a corresponding source code entity was affected by any refactorings in release x_i. So we mapped the source code entities (i.e. classes and methods) from version x_{i-1} to x_i and put all the RMI values for the entities in x_{i-1} into the not affected group if the entity had zeros in all refactoring columns in x_i, otherwise we put the RMI values of the entity into the affected group. Once we had these two groups we run the Mann-Whitney U test. The p-value of the test helped us judging whether there is significant difference in the maintainability values between the source code entities subjected to refactoring and the entities unaffected by refactoring. Moreover, we used the mean rank values produced by the test to decide the direction

of the differences, namely whether the maintainability value is lower or higher within one of the groups.

Firstly, to get an impression about the difference of the RMI of the refactored and non-refactored code elements we considered the average RMI values of the entities falling into these groups. The result is shown in Figure 6.6.

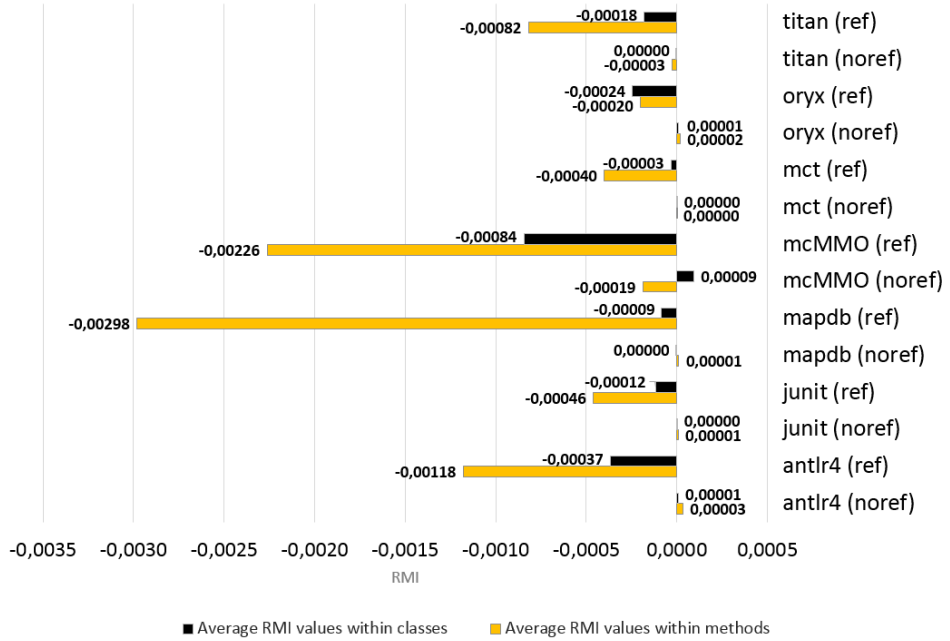


Figure 6.6. Average RMI values within the refactored and non-refactored entities

Yellow columns denote the average RMI values within methods, while black color is for classes. The *ref* and *noref* marks next to the systems stand for the two groups, *ref* is the first group of entities (i.e. the ones affected by refactorings) and *noref* is the second group. As can be seen, all the average maintainability values of the refactored group (regardless whether for classes or methods) is much lower than the non-refactored group. What is more, almost all the average maintainability values for non-refactored classes and methods are positive (except for titan and classes of mcMMO), while the values related to refactored entities are negative. Given that positive RMI value means above average maintainability, negative RMI means maintainability below the average, it is clear that the source code entities targeted by refactorings have lower maintainability than the average. We note that code size affects the RMI values, thus further examinations are required to find out whether large code size in itself implicates more refactoring or not.

To formally evaluate the above hypothesis, we performed a Mann-Whitney U test on the RMI values of the two groups defined above. We executed the test on each system both for the groups of classes and methods. Tables 6.10 and 6.11 summarize the results of the test runs.

The main result of the test is the p-value (two-tailed) shown in the second column. This indicates whether the null-hypothesis should be rejected, which states that *there is no significant difference between the RMI values of the entities affected by refactorings in adjacent releases and the RMI values of non-refactored entities.* Thus a p-value below 0.05 indicates that the hypothesis should be rejected and the alternative hypothesis

System	p-value	N^o noref cl.	N^o ref cl.	M. rank^{noref}	M. rank^{ref}	Cliff's δ
antlr4	**0.00001**	385	23	210.61	102.26	0.53
junit	**0.00628**	646	9	330.38	156.83	0.53
mapdb	**0.01186**	415	4	211.46	58.37	0.73
mcMMO	0.27604	85	4	45.65	31.25	0.32
mct	**0.00000**	2013	15	1019.94	284.60	0.73
oryx	**0.04467**	489	15	254.78	178.13	0.30
titan	**0.00009**	1145	13	583.61	217.50	0.63

Table 6.10. The Mann-Whitney U test results for refactored and not refactored classes

should be accepted, namely that there is a significant difference in the RMI values between the two groups.

As can be seen all the p-values are well below 0.05, except for mcMMO and mapDB at method granularity, so we can conclude that according to the statistical test, *there is a significant difference in the RMI values between the source code entities being refactored and the entities not affected by any refactoring.*

System	p-value	N^o noref mth.	N^o ref mth.	M. rank^{noref}	M. rank^{ref}	Cliff's δ
antlr4	**0.00000**	3104	40	1583.10	750.16	0.53
junit	**0.00466**	2253	12	1135.84	600.21	0.47
mapdb	0.20610	3358	3	1681.63	973.00	0.42
mcMMO	0.06529	813	5	410.69	215.40	0.48
mct	**0.00346**	11068	16	5545.88	3205.34	0.42
oryx	**0.00034**	2333	19	1181.03	620.82	0.48
titan	**0.00530**	7950	17	3987.32	2431.26	0.39

Table 6.11. The Mann-Whitney U test results for refactored and not refactored methods

To tell something about which group has higher RMI values, thus better maintainability, we should observe the mean ranks. The column Mean rank^{ref} shows the mean rank values within the refactored groups, while Mean rank^{noref} displays the mean ranks in the not refactored groups. If the mean rank value of one group is higher, it means the RMI values in that group is significantly higher than in the other group. We report Cliff's δ values in the last columns. Since in each row of both tables the mean ranks of refactored group are lower than the not refactored group and all the Cliff's δ values are positive, we can answer RQ1.

Answer to RQ1 assessing the improved dataset: *The maintainability of source code entities subjected to refactorings is significantly lower than the maintainability of not refactored entities according to the manually validated dataset.*

To compare this result to the findings of the previous studies (Sections 6.5 and 6.6), we can say that the early results seem to remain valid after using the validated dataset. In our not published work we did a more detailed comparison between the results of the base and the improved dataset.

6.7.2 Source Code Metrics Analysis

To study the effect of code refactoring on metrics we used the approach described in Section 6.4: we calculated the metric differences between versions x_{i-1} and x_i and run the Mann-Whitney U test on the differences (we refer to this test as MWU^{Diff}). Besides the above, we run the Man-Whitney U test on the pure metric values in

version x_{i-1} the same way like for RMI values before (for now MWU^{Prev}), which helps us discover what are those source code properties the developers pay attention to find refactoring candidates. We calculated the Cliff's δ effect size measure as well (δ^{Diff} and δ^{Prev} accordingly). We investigated the patterns we revealed from the tests above and give an overview about our main findings.

The results show that the LOC values of refactored classes are significantly larger in 6 out of the 7 subject systems. Thus developers tend to select and refactor large classes, what is not surprising. LLOC (Logical Lines Of Code) and NOS (Number of Statements) are also size metrics (and strongly correlate with LOC), thus their strong effect is also not surprising.

Another group of metrics showing clear patterns are the coupling metrics: RFC (Response set For a Class), CBO (Coupling Between Object classes), CBOI (Coupling Between Object classes Inverse), NII (Number of Incoming Invocations) and NOI (Number of Outgoing Invocations). In 4 out of 7 cases both the metric values and their changes showed a significant difference, while in 1-1 case only the metric values or their changes showed a significant difference between the refactored and non-refactored group of classes.

The third group of metrics with remarkable patterns are the complexity metrics: WMC (Weighted Methods per Class), NL (Nesting Level) and NLE (Nesting Level Else-If). For all three metrics in 4 out of 7 systems we found significant differences both in the metric values and their changes, while in 2 and 1 cases (for WMC, and NL, NLE, respectively) only the metric values differed between the refactored and non-refactored group of classes.

The comment, code clone related and inheritance metrics do not show clear patterns.

Considering the effect size, what is also remarkable is that all the available δ^{Prev} values are positive, which reflects that the appropriate metric values in the refactored group is much likely to be larger than in the non-refactored group. Although most of the values reflect a medium level effect size, such high values like 0.85 (CBOI for mapdb) and 0.84 (RFC for mct) also appear. It suggests that coupling is one of the main factors that developers consider when they select the targets for refactoring (which is in line with other research results [68]). Very similar phenomenon applies to size (i.e. LLOC, LOC, NOS) and complexity (i.e. WMC, NL, NLE) metrics in general.

An interesting observation can be made in connection with the CLOC (Comment Lines Of Code) metric. It measures the amount of comments in a class, and for 5 out of the 7 systems its average value is larger in the classes that are refactored in version x_i (it is even true for the CD – Comment Density metric – for 4 out of 7 cases). Thus developers refactor classes with more comments, which might seem to be a contradiction at first glance. However, comments are often outdated, misleading or simply comment out unnecessary code, that are all indicators of poor code quality, thus refactoring is justified.

We also compared these results with our previous findings performed on the base refactoring dataset, where just like here size, complexity and coupling metrics showed the highest differences in their distributions within the refactored and non-refactored groups. However, while previous results displayed 2-4 significant cases out of 7, we had 3-6 significant cases in the new tests on the improved dataset, with much stronger p-values, thus we can be much more confident in the results. We found one major difference as well; the clone related metrics (CI at class-level and CC at method-level)

are much less significant in the analysis of the improved dataset.

6.8 Threats to Validity, Limitations

In this section, we summarize the threats to validity of our study.

From the point of view of the researches on the base dataset, we note that Ref-Finder, the tool we used to mine refactorings from the selected projects, is not perfect. According to its authors the precision of the tool is 79% [74], which means there might be false positive refactorings included in our dataset. To mitigate this threat we performed the manually validation of our entire dataset and eliminate false positive instances. However, in our subject systems we found lower precision after the manual validation than the authors of Ref-Finder published, the results are in line with ones we get on the base dataset, and even we can be much more confident in our findings.

The key attribute in the datasets is the fully qualified name of the method with parameter descriptions. If a source code element is renamed between two consecutive releases, we do not track it and its metrics, and handle it as a new one in the next release. Following such renamed entities throughout code versions is a really hard task in general, but the number of renaming is relatively small compared to other changes, thus we consider this to be a minor threat.

In addition, there might be arbitrary changes between the two examined releases of the systems, not just refactorings. Therefore, we cannot be sure that changes in a source code element that is affected by a refactoring are only due to the refactoring itself, or other unrelated modifications cause it. The optimal solution would be to find those particular commits that introduce the refactoring, although there is no guarantee that the commit contains only code related to the refactoring itself. Finding those commits would require running Ref-Finder for each subsequent revision between two releases, which is obviously unfeasible. However, during manual validation we found that most of the refactored source code elements did not contain any additional change, thus the impact of this threat is limited.

Another threat to our results is that we investigated only seven Java systems, which may not represent correctly the general characteristics of all of the software systems considering refactoring activities in practice. Moreover, since manual validation requires huge human effort, the number of refactoring instances in the improved dataset is also limited.

As we employed human evaluation in the manual validation, we cannot be 100% sure that each and every refactoring instance was correctly classified by the authors. However, both evaluators are very experienced researchers and also software developers that mitigates this threat. Moreover, all the authors consulted about refactoring instances that were not straightforward to classify, and resolved these cases by majority voting.

By manual validation we can ensure the high precision of the refactoring instances in the improved dataset, however, we cannot guarantee complete recall. It is possible that there are true refactoring instances that Ref-Finder did not find, thus we also omitted these during the manual validation, what might cause a bias in the evaluation. Nonetheless, the extraction rules of Ref-Finder are quite conservative, thus the tool is more likely to report false positive instances than to omit true negative ones. Therefore, the effect of missed refactoring instances is low.

Regarding the statistical analysis, the relatively small number of refactoring instances results in unbalanced datasets, which might cause a loss of statistical power. The unbalanced property comes from the fact that there could be much less refactored source code elements than unaffected elements and the tests compare the properties of these two sets, the latter one containing a much larger number of samples. However, we chose the Mann-Whitney U method to perform the hypothesis testing, which is not sensitive to population sizes and is able to handle highly unbalanced datasets applying exact distributions of small samples.

6.9 Summary

In this chapter, we present a publicly available dataset which is intended to assist the research of refactoring activities in practice. The dataset contains fine-grained refactoring information extracted by the Ref-Finder tool and more than 50 types of source code metrics for 37 releases of 7 Java open-source systems at class- and method-level. We also store exact version and line information in the dataset to support reproducibility. In addition to the source code metrics, the dataset includes the relative maintainability indices of source code elements. The dataset is available at the PROMISE data repository. Another contribution is the extension of the Ref-Finder tool that allows batch-style analysis and result reporting attached to the source code elements.

By utilizing the constructed dataset, we investigated the relationship between maintainability and refactoring activities, and we also assessed how refactorings affect different source code metrics at class- and method-level. We found that classes with poor maintainability are subject to more refactorings in practice than classes with higher technical quality. Considering metrics, the number of clone instances, complexity, and coupling have improved, although comment related metrics decreased. We found a significant decrease in size metrics as well.

The result of method-level assessment regarding to software maintainability is in line with the class-level finding. We found that methods with poor maintainability are subject to more refactorings than methods with higher maintainability. Clone coverage, size metrics, and the number of outgoing invocations decreased most intensively in the methods subjected to frequent refactorings. This might indicate that doing code refactoring in practice indeed mitigates unwanted code characteristics, such as clones, size, or coupling and result in more maintainable software systems.

The other main contribution of this work is the manually validated subset of our original dataset. It contains one manually validated release for each of the 7 systems from which we expect better quality results. This dataset was also made publicly available in PROMISE. Although, the manually validated refactoring dataset is in itself a major contribution, we also utilized it to replicate and extend our earlier studies and re-examine the connection between maintainability and code refactoring as well as the distribution of the individual source code metrics in the refactored and non-refactored source code elements. The results showed that the overall average maintainability of refactored entities was much lower in the pre-refactoring release than in the entities subjected to no refactorings. This strongly suggests that refactoring is indeed used in practice on deteriorated entities whether it is a conscious activity of the developers or not. Moreover, we were interested in how the distribution of typical source code metrics looks like in the refactored and non-refactored source code elements. We found that the size, complexity and coupling-related metric values were significantly higher

in the source code elements being refactored. We could also confirm that developers do not only select their targets for refactoring based on these metrics, but they even try to control and reduce their values, as these metrics grow much slower (or even decrease) in the source code elements touched by refactorings.

Even though this work presents fundamental research, the results can be used as a first step towards understanding refactoring practices more deeply. Having full understanding of developers' actions, we can propose new methods and tools for them that are aligned with their current habits and help in performing refactoring faster, cheaper, and better.

The author's contributions. To compose the refactoring dataset, the author improved Ref-Finder with an automatic batch analysis feature, i.e., to be able to automatically extract refactorings not just between two adjacent versions of a software but between any number of adjacent version pairs. The author also implemented an export feature in Ref-Finder that writes the revealed refactorings and all of their attributes into CSV files for each refactoring type. The author took part in the dataset construction by mapping the refactorings to the source code elements and half of the manual validation is also his work. He participated in the elaboration of the analysis methodology and in the evaluation of the results. The publications related to this chapter are:

- **I. Kádár**, P. Hegedűs, R. Ferenc, T. Gyimóthy. A Code Refactoring Dataset and Its Assessment Regarding Software Maintainability. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, pages 599–603. IEEE Computer Society, 2016.
- **I. Kádár**, P. Hegedűs, R. Ferenc, T. Gyimóthy. Assessment of the Code Refactoring Dataset Regarding the Maintainability of Methods. In *International Conference on Computational Science and Its Applications*, volume 9789 Lecture Notes in Computer Science (LNCS), pages 610–624. Springer International Publishing, 2016.
- **I. Kádár**, P. Hegedűs, R. Ferenc, T. Gyimóthy. A Manually Validated Code Refactoring Dataset and Its Assessment Regarding Software Maintainability. In *Proceedings of the The 12th International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 10–14. ACM, 2016.
- P. Hegedűs, **I. Kádár**, R. Ferenc, T. Gyimóthy. Empirical Evaluation of Software Maintainability Based on a Manually Validated Refactoring Dataset. Second revision submitted to journal *Information and Software Technology*. 44 pages. Elsevier B.V.

"Elegance is not a dispensable luxury, but a factor that decides between success and failure."

— Edsger Dijkstra

7 Conclusions

In this thesis, we covered two main topics: symbolic execution and refactoring activity analysis.

In the field of *symbolic execution*, we focused on the handling of the exponentially growing state space and on making the detection of runtime errors more efficient. To summarize our work, we successfully applied the symbolic execution technique on real-life Java systems using the proposed method-level symbolic execution and made it more efficient by a novel constraint building mechanism. We also investigated the possible state space limitations while giving new algorithms in order to prioritize the paths to explore. As a result of this research work, we developed new methods and a tool that is now part of a commercial static source code analysis toolchain.

In *refactoring activity analysis*, we revealed the connection between practical refactoring activity in open-source Java systems and the maintainability quality indicator. We also showed the connection with other quality attributes (source code metrics), assessed at both class- and method-level. We are not aware of any publications that performed similar investigations at method-level granularity. To perform the assessment, we built a refactoring dataset that contains detailed refactoring information mapped to classes and methods and more than 50 types of source code metrics as well. We made this dataset publicly available together with its manually validated subset. We believe that the possible utilization of the assembled datasets goes much beyond our investigations presented in this thesis, and we would like to encourage the research community to use it to reveal more complex phenomena regarding practical refactoring.

Despite the results of our research efforts, there is still work left to be done. Here we summarize some possible future research directions.

The presented approach to symbolic execution is not limited to runtime exception detection. It would be promising to utilize the potential of the technique by implementing other types of error and rule violation checkers. E.g. we could detect special types of infinite loops, dead or unused code parts, or even SQL injection vulnerabilities. We believe that one of the limitations here is the explosion of the state space. If we would be able to handle the size of the state space efficiently or keep it reduced, that would open a large number of opportunities. In the future, we plan to include more

systems into the empirical experiments in order to try and find the optimal depth and state number limits of the symbolic execution tree in general to make RTEHunter and other symbolic execution engines more efficient. We also plan to develop more efficient search strategies by adding new features and using other machine learning approaches. For example, we believe that the inclusion of including test coverage information to the search could be a promising approach. We would also like to validate the errors reported by the RTEHunter to find out more about its precision. The manual validation would need enormous efforts, but it is also possible to write a framework that executes those paths that led to the reported issues to enable automatic decision whether an issue is true positive or not. Regarding to constraint building, optimization would be an interesting research area: building and satisfying constraints at each and every branching point is necessary to avoid unfeasible paths, but it is also very time and memory consuming. For example, making a heuristic that can decide that it is worth to satisfy the path condition at a point might worth the research effort. It would also be interesting to evaluate the approach that runs symbolic execution without any constraint building and satisfying in the first round, then in the second round it builds and satisfies path conditions only for the paths that led to an issue reported in the first round.

In the topic of practical refactoring activity investigation, we plan to continuously extend the number of systems in the improved dataset as well as the number of manually evaluated true positive refactoring instances, making the dataset more valuable to support further research. It would have been interesting to see how our results change when only considering individual refactoring types, but unfortunately, we do not have enough refactoring data in the validated, improved dataset to be able to derive meaningful results at this fine-grained level. Thus, continuous extension of the manually validated instances is one of our major goals, but a community supported common effort would also be very welcome. We are planning to reveal more complex phenomena in connection with practical refactorings, especially the relationship between bugs and refactoring activities.

Personally, these years of research have given me code-quality-centered thinking in daily software development. I firmly believe that it is worth energy to find and code solutions that support better quality, rather than apply instant ones. Elegance always pays off. It results in a program that is easier to understand, debug and maintain, and that is where the real value comes in. I have learnt a lot of technical details about the Java programming language especially in the first part of our research, and I also understood the importance of quality control. I will always support the fight for source code quality in both industry and research.

Appendices

A

Summary in English

The research work behind this thesis is inspired by the wish to develop high quality software systems in industry in a more effective and easier way. The thesis consists of two main topics: *the utilization of symbolic execution for runtime error detection* and *the investigation of practical refactoring activities*. Both topics address the area of program source code quality. The result statements have been grouped into four major thesis points. The relation between thesis points and supporting publications is shown in Table A.1.

Part I - Advances in symbolic execution for runtime error detection.

The first part of the thesis deals with symbolic execution and with the question of how to apply this technique efficiently for detecting runtime faults in Java software systems.

1. **A method-level symbolic execution technique for runtime error detection in real-world software systems.**

   The contributions of this thesis point are related to the utilization of symbolic execution for runtime error detection and are discussed in Chapter 3.

   Most of the runtime failures of a software system can only be revealed during test execution, which has a very high cost and the maintenance activities, particularly bug fixing of the system also require a considerable amount of resources. Our purpose was to develop a new method and tool, which supports this phase of the software engineering lifecycle by detecting runtime exceptions in Java programs (such as NullPointerException or ArithmeticException) and finding dangerous parts in the source code that could behave as time-bombs during further development.

   We use the technique called symbolic execution [53] to implement an approach that is able to explore the possible execution paths of the program. The runtime environment we implemented on top of *Symbolic PathFinder* [75] starts the symbolic execution for every method of an arbitrary Java system keeping the state space reduced. If the symbolic execution starts from the entry point of the

program i.e. from the main() method only, the state space will explode before the execution reaches the majority of the code. By executing the methods of the program symbolically, we can determine those execution branches that throw exceptions, and our algorithm is also able to generate concrete test inputs that cause the program to fail in runtime. Besides small example codes, we evaluated our algorithm on three open source systems and found multiple runtime issues. We found multiple errors in the log4j system that were also reported as real bugs in its bug tracking system.

The author's contributions. The author performed the exploration of symbolic execution and the Symbolic PathFinder execution engine for the purpose of runtime exception detection. The idea of method-level symbolic execution and the entire implementation of the runtime environment which performs the analysis is the author's work. He performed the detection of runtime exceptions by implementing a module in Symbolic PathFinder which also gives a stack trace leading to the found error and the related parametrization as a test input that crashes the program. The investigation of the found runtime exceptions and proving their validity by the bug reports, found in the bug tracking systems of the analyzed projects was also the author's role. He contacted the authors of Symbolic PathFinder several times to report some blocker issues that held back the research. The publications related to this thesis point are:

- **I. Kádár**, P. Hegedűs, R. Ferenc. Runtime Exception Detection in Java Programs Using Symbolic Execution. In *Proceedings of the 13th Symposium on Programming Languages and Software Tools – SPLST'13*, pages 215–229, 2013.
- **I. Kádár**, P. Hegedűs, R. Ferenc. Runtime Exception Detection in Java Programs Using Symbolic Execution. In *Acta Cybernetica*, pages 331–352, 2014.

2. **A constraint building mechanism for symbolic execution to improve runtime error detection accuracy.**

   The contributions of this thesis point are related to the utilization of symbolic execution for runtime error detection and are discussed in Chapter 4.

   Symbolic Checker, the symbolic execution engine developed at the Software Engineering Department of the University of Szeged is able to detect runtime errors (such as null pointer dereference, bad array indexing, division by zero) in Java programs without running the program in real-life environment. According to the theory of symbolic execution, the program does not run with specific input data, but the inputs are handled as symbolic variables. When the execution of the program reaches a branching condition containing a symbolic variable, the execution continues on both branches. At each branching point, both the affected logical expression and its negation are accumulated on the true and false branches accordingly, thus all execution paths will be linked to a unique formula over the symbolic variables called path condition.

   We introduced a novel constraint system construction mechanism, which improves the accuracy of the runtime errors found by Symbolic Checker by treating

the assignments in the program as conditions too. Thus we can track the dependencies of the symbolic variables extending the original principles of path condition building. The presented method also substitutes the symbolic variables with concrete values if the built constraint system unambiguously determines their value. As a result, it enables the detection of runtime errors that would not be possible using a conventional symbolic execution tool, and by concreting symbolic variables the size of the symbolic execution tree can be reduced as well, which also implies improvements in performance. We demonstrate the advantages of our algorithm through example codes emphasizing the difference compared to a conventional symbolic execution tool and to Symbolic Checker without using the novel constraint building mechanism. We also found runtime errors in large, real-life systems that would not have been possible with the traditional constraint building mechanism.

The author's contributions. The author took part in the design and development of the Symbolic Checker symbolic execution engine as the lead developer. He devised the concept of the proposed constraint building mechanism. He implemented and integrated it into the symbolic execution engine. The evaluation of the proposed method by comparing it to the conventional approach and performing tests on example codes and on real-life systems are also the author's work. The publication related to this chapter is:

- **I. Kádár**, P. Hegedűs, R. Ferenc. Adding Constraint Building Mechanisms to a Symbolic Execution Engine Developed for Detecting Runtime Errors. In *Proceedings of the International Conference on Computational Science and Its Applications – ICCSA*, volume 9159 Lecture Notes in Computer Science (LNCS), pages 20–35, Springer International Publishing, 2015.

3. **Novel search strategies for symbolic execution and empirical investigation of state space limitations.**

   The contributions of this thesis point are related to the utilization of symbolic execution for runtime error detection and are discussed in Chapter 5.

   Our symbolic execution engine, at the Department of Software Engineering at the University of Szeged further developed and received the new name of RTEHunter. According to the theory of symbolic execution, RTEHunter builds a tree, called symbolic execution tree, composed from all the possible execution paths of the program. RTEHunter detects runtime issues by traversing the symbolic execution tree and, if a certain condition is fulfilled, the engine reports an issue. However, the number of execution paths increases exponentially with the number of branching points, thus the exploration of the whole symbolic execution tree is impossible in practice. To overcome this problem, different kinds of constraints can be set up over the tree. E.g. the number of symbolic program states, the depth of the execution tree, or the time consumption can be limited. Because of the path explosion problem, the limitation of the state space of the symbolic execution is of major importance in this scenario. Our goal in this work is to find the optimal parametrization of RTEHunter in terms of maximum number of states, maximum depth of the symbolic execution tree, and search strategy in order to find more runtime issues in less time.

The empirical investigations on three open-source Java systems showed that adjusting the maximum number of states for the symbolic execution trees directly impacts the execution time, but not the number of found issues. On the other hand, the limitations of the depth of the tree has more importance in the detection of runtime errors. We found different optimal depth limits for the three different systems, but we can conclude that errors occur more often in the state depth of 0 to 60 compared to the deeper levels, but it also highly depends on the applied search strategy. We propose two novel search strategies that strive to guide the symbolic execution towards the more error prone source-code fragments using both static and dynamic information. The null-heuristic search strategy performs better by finding up to 16 % more errors within the same time frame than the default depth-first search. The linear regression-based heuristic also outperforms DFS, it detects more than twice as many errors in ArgoUML and Jetspeed.

The author's contributions. The author performed the entire empirical analysis to find the connection between the maximum number of states for the symbolic execution trees, the analysis time and the number of issues found by running RTEHunter many times on three open source systems. He also performed many experiments to find the optimal depth limits for each system and revealed the depth level where most errors can be found. The idea of the two search strategies for the state space exploration, their implementation and their entire evaluation are the author's work. The publication related to this chapter is:

- **I. Kádár.** The Optimization of a Symbolic Execution Engine for Detecting Runtime Errors. In *Acta Cybernetica*, pages 573–597, 2017.

Part II - Investigation of refactoring activities based on a new dataset.

In the second part of the thesis, we deal with how developers apply refactoring operations in practice. To perform the investigation, we assembled a publicly available dataset of refactorings found in open-source Java systems. This dataset does not serve only our research, but we would also like to encourage the research community to utilize it in the future.

4. **Assessment of refactoring activities on classes and methods based on a new public dataset.**

   The contributions of this thesis point are related to the investigation of practical refactoring activities based on a refactoring dataset and are discussed in Chapter 6.

   Source code refactoring is a popular and powerful technique for improving the internal structure of software systems. The concept of refactoring was introduced by Fowler [29] and nowadays IT practitioners think of it as an essential part of the development process. Despite the high acceptance of refactoring techniques by the software industry, it has been shown that practitioners apply code refactoring differently than Fowler originally suggested and we lack empirical research results on the real effect of code refactoring and its applications.

   We present an excessive open dataset of source code metrics and applied refactorings through several releases of 7 open-source Java systems intended to assist the

research of refactoring activities in practice. The dataset contains fine-grained refactoring information revealed by the Ref-Finder open-source refactoring extraction tool [52] and more than 50 types of source code metrics for 37 releases of 7 open-source systems at class- and method-level. To construct the dataset, we extended the Ref-Finder tool to allow batch-style analysis and result reporting attached to the source code elements as well.

By utilizing the dataset, we investigated the relationship between maintainability and refactoring activities, and we also assessed how refactorings affect different source code metrics at both class- and method-level. We found that classes with poor maintainability are subject to more refactorings in practice than classes with higher technical quality. Considering metrics, number of clone instances, complexity, and coupling have improved, although comment related metrics decreased. We found a significant decrease in size metrics as well. At method-level, we found that methods with poor maintainability are subject to more refactorings in practice than methods with higher technical quality. The clone coverage, size metrics, and number of outgoing invocations decreased the most intensively in the methods subjected to frequent refactorings, which might indicate that doing code refactoring in practice indeed mitigates unwanted code characteristics such as clones, size, or coupling, and results in more maintainable software systems.

Moreover, we present a manually validated subset of the dataset of applied refactorings mentioned above that helps ensure the high precision of this improved dataset. Using this data, we studied several aspects of the refactored and non-refactored source code elements (classes and methods). The results on the manually validated dataset are in line with the numbers on the base dataset, though they are much more consistent and significant for more subject systems. The reason for this is that the preliminary tests were biased by the many false positive refactoring instances that have been removed from the validated dataset. The results showed that the overall average maintainability of refactored entities was much lower in the pre-refactoring release than the entities subjected to no refactorings. Size, complexity, and coupling metrics show the highest differences in their distribution within the refactored and non-refactored groups.

The possible utilization of the assembled datasets goes way beyond our investigations. We made them publicly available at the PROMISE data repository [64] to encourage the research community to reveal more complex phenomena in connection with practical refactorings.

The author's contributions. To compose the refactoring dataset, the author improved Ref-Finder with an automatic batch analysis feature, i.e., to be able to automatically extract refactorings not just between two adjacent versions of a software but between any number of adjacent version pairs. The author also implemented an export feature in Ref-Finder that writes the revealed refactorings and all of their attributes into CSV files for each refactoring type. The author took part in the dataset construction by mapping the refactorings to the source code elements and half of the manual validation is also his work. He participated in the elaboration of the analysis methodology and in the evaluation of the results. The publications related to this chapter are:

- **I. Kádár**, P. Hegedűs, R. Ferenc, T. Gyimóthy. A Code Refactoring Dataset

and Its Assessment Regarding Software Maintainability. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, pages 599–603. IEEE Computer Society, 2016.

- **I. Kádár**, P. Hegedűs, R. Ferenc, T. Gyimóthy. Assessment of the Code Refactoring Dataset Regarding the Maintainability of Methods. In *International Conference on Computational Science and Its Applications*, volume 9789 Lecture Notes in Computer Science (LNCS), pages 610–624. Springer International Publishing, 2016.
- **I. Kádár**, P. Hegedűs, R. Ferenc, T. Gyimóthy. A Manually Validated Code Refactoring Dataset and Its Assessment Regarding Software Maintainability. In *Proceedings of the The 12th International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 10–14. ACM, 2016.
- P. Hegedűs, **I. Kádár**, R. Ferenc, T. Gyimóthy. Empirical Evaluation of Software Maintainability Based on a Manually Validated Refactoring Dataset. Second revision submitted to journal *Information and Software Technology*. 44 pages. Elsevier B.V.

Here, we also summarize the main publications related to the various thesis points in the table below.

No.	[96]	[94]	[97]	[93]	[98]	[99]	[95]
1	•	•					
2			•				
3				•			
4					•	•	•

Table A.1. Thesis contributions and supporting publications

B

Magyar nyelvű összefoglaló

A disszertációban közzétett kutatási munkát az inspirálta, hogy hatékonyabban és könnyebben tudjunk magas minőségű szoftverrendszereket előállítani az ipari szoftverfejlesztésben. A disszertáció két témát ölel fel: *szimbolikus végrehajtás felhasználását futásidejű hibák detektálása céljából*, valamint *refaktoring tevékenységek vizsgálatát a gyakorlatban*. Mindkét témakör a forráskód minőség területén helyezhető el. Kutatásaink eredményét négy tézispontba szerveztük. Az egyes tézispontokat alátámasztó publikációkat a B.1 táblázat foglalja össze.

I. rész - Új eredmények a szimbolikus végrehajtás alkalmazásában futásidejű hibák detektálása céljából.

A disszertáció első része a szimbolikus végrehajtás statikus kódelemző technikával, illetve annak felhasználásával foglalkozik abból a szempontból, hogy hogyan tudjuk hatékonyan alkalmazni futásidejű hibák megtalálására nagy méretű Java nyelvű programokban.

1. **Metódus szintű szimbolikus végrehajtás futásidejű hibák detektálására valós méretű szoftverrendszerekben.**

   Ezen tézispont eredményei a szimbolikus végrehajtás futásidejű hibadetektálásra történő alkalmazásához tartoznak és a 3. fejezetben tárgyaltuk részletesen.

   A futásidejű hibák többsége gyakran csak teszteléssel deríthető fel. A tesztelés azonban rendkívül költséges, illetve a karbantartási munkák, különösen a hibajavítás szintén jelentős költségeket emészt fel. A célunk az volt, hogy kifejlesszünk egy olyan módszert és eszközt, amely a szoftverfejlesztési folyamat fenti fázisait támogatja azáltal, hogy futásidejű kivételeket (mint például NullPointerException vagy ArithmeticException) detektál Java programokban, illetve olyan veszélyes kódrészletekre mutat rá, amelyek mintegy időzített bombaként viselkednek a fejlesztés későbbi szakaszaiban.

   A szimbolikus végrehajtás programelemző technikát [53] használtuk fel arra, hogy megvalósítsuk ezt az eszközt. Szimbolikus végrehajtással be tudjuk járni a program lehetséges végrehajtási útvonalait. Azonban ahhoz, hogy a program végre-

hajtási útvonalaiból álló állapottér mérete kezelhető maradjon, egy olyan futtatókörnyezetet valósítottunk meg, amely a Symbolic PathFinderre [75] felhasználásával tetszőleges Java program metódusait külön-külön, egyesével képes szimbolikus végrehajtással elemezni. Ha hagyományosan alkalmaznánk ezt a technikát, vagyis kizárólag a program belépési pontját jelentő *main()* metódusból indítanánk az elemzést, az állapottér kezelhetetlen méretűre nőne már az elemzés elején, ezért a programkód túlnyomó részét el sem érnénk. Azáltal, hogy a program metódusait szimbolikusan végrehajtjuk, meghatározzuk azokat a végrehajtási útvonalakat, amelyek kivételt dobnak. Ezen felül teszteseteket is generálunk, amelyek ezeken az útvonalakon futtatják programot elérve annak leállását a kivétel dobása miatt. Amellett, hogy kisebb példakódokon teszteltük az algoritmust, három nyílt forrású rendszert is leelemeztünk, amelyekben szintén találtunk potenciális futásidejű hibákat. A kifejlesztett módszer helyessége és hasznossága azzal igazolható, hogy a log4j naplózókönyvtár hibakövető rendszerében több olyan hibajelentést is találtunk, amelyek okát az általunk fejlesztett eszköz is megtalált.

A szerző hozzájárulása az eredményekhez. A szerző térképezte fel a szimbolikus végrehajtás módszerét és a Symbolic PathFinder végrehajtó motor lehetőségeit, illetve azt, hogy hogyan alkalmazható mindez futásidejű programhibák megtalálására. A metódus szintű szimbolikus végrehajtás kidolgozása és a futtatókörnyezet megvalósítása, ami elvégzi az elemzést, teljes egészében a szerző munkája. Az ő érdeme a futásidejű hibák detektálása azzal, hogy megvalósított egy új modult a Symbolic PathFinderben, ami visszaadja a hibára vezető végrehajtási útvonalat és az elemzett metódusnak ehhez az útvonalhoz tartozó paraméterezését, amely tesztinputként felhasználható. A megtalált futásidejű kivételek átvizsgálása és azok érvényességének bizonyítása azzal, hogy megkereste őket az elemzett progoramok hibakövető rendszereiben szintén a szerző munkája volt. A szerző számos alkalommal felvette a kapcsolatot a Symbolic PathFinder fejlesztőivel, hogy olyan blokkoló hibákat jelentsen be, amelyek hátráltatták a kutatást.

A tézisponthoz tartozó publikációk:

- **I. Kádár**, P. Hegedűs, R. Ferenc. Runtime Exception Detection in Java Programs Using Symbolic Execution. In *Proceedings of the 13th Symposium on Programming Languages and Software Tools – SPLST'13*, pages 215–229, 2013.
- **I. Kádár**, P. Hegedűs, R. Ferenc. Runtime Exception Detection in Java Programs Using Symbolic Execution. In *Acta Cybernetica*, pages 331–352, 2014.

2. **Új feltételrendszer építő mechanizmus szimbolikus végrehajtáshoz futásidejű hibadetektálás pontosítására.**

   Ez a tézispont a szimbolikus végrehajtás futásidejű hibadetektálásra történő alkalmazásához tartozik és a 4. fejezetben tárgyaltuk részletesen.

   A Szegedi Tudományegyetem Szoftverfejlesztés Tanszékén fejlesztett szimbolikus végrehajtó motor, a *Symbolic Checker* Java programokban képes futásidejű problémákat (mint például null pointer dereferencia tömb alul- és túlindexelés, nullával való osztás) detektálni anélkül, hogy a programot valós környezetben futtatni kellene. A szimbolikus végrehajtás elmélete szerint a programot nem konkrét

input adatokon futtatjuk, hanem ismeretlenekként, ún. szimbolikus változókként kezeljük a bemenetet. Amikor a program végrehajtása egy olyan feltételes vezérlési szerkezethez ér, amelyben a logikai kifejezés tartalmaz szimbolikus változót, az igaz és a hamis ágon is folytatódni fog a végrehajtás. Ezeknél az elágazási pontoknál a vezérlési szerkezetben szereplő logikai kifejezést feljegyezzük és továbbvisszük az igaz ágon, illetve annak negáltját a hamis ágakon. Ennek megfelelően az összes végrehajtási útvonalhoz egy egyedi, szimbolikus változók feletti formulát rendelünk, amelyet path condition-nek (PC) nevezünk.

Ebben a tézispontban bemutatunk egy a fent leírtaktól eltérő feltételrendszer építő mechanizmust, amely pontosítja a Symbolic Checker által adott futásidejű hibatalálatokat azzal, hogy a programban lévő értékadásokat is feltételként kezeli. Ennek köszönhetően követni tudjuk a szimbolikus változók közötti függőségeket, kibővítve a path condition felépítésének eredeti koncepcióját. Ezen felül, amennyiben a feltételrendszer egyértelműen meghatározza valamely szimbolikus változó értékét, azt a változót a meghatározott konkrét értékkel helyettesítve folytatjuk a szimbolikus végrehajtást. Ennek eredményeként olyan futásidejű hibákat is megtalálhatunk, amelyeket egy hagyományos feltételrendszer építést használó szimbolikus végrehajtó eszköz nem képes. Azáltal, hogy szimbolikus változókat konkretizálunk, a szimbolikus végrehajtási fa mérete is redukálódik, ami csökkenti az elemzéshez szükséges időt. A kifejlesztett algoritmus használatának előnyeit példakódokon keresztül szemléltetjük. Kiemeljük a különbségeket egy hagyományos szimbolikus végrehajtó eszköz, az új feltételrendszer építő mechanizmus nélkül futtatott Symbolic Checker, valamint az új módszer eredményei között. Ezen felül bemutatunk nagyméretű, valós rendszerekben talált olyan futásidejű hibatalálatokat, amelyek detektálása a konvencionális feltételrendszer építési mechanizmus használatával nem lett volna lehetséges.

A szerző hozzájárulása az eredményekhez. A szerző részt vett a Symbolic Checker szimbolikus végrehajtó motor fejlesztésében vezető fejlesztőként. Ő dolgozta ki a bemutatott feltételrendszer építő mechanizmust, megvalósította és integrálta azt a Symbolic Checker-be. A módszer kiértékelése, összehasonlítása a konvencionális megközelítéssel, valamint a példakódokon és a valós rendszereken történő tesztelemzések futtatása szintén a szerző nevéhez fűződik.

A tézisponthoz tartozó publikáció:

- **I. Kádár**, P. Hegedűs, R. Ferenc. Adding Constraint Building Mechanisms to a Symbolic Execution Engine Developed for Detecting Runtime Errors. In *Proceedings of the International Conference on Computational Science and Its Applications – ICCSA*, volume 9159 Lecture Notes in Computer Science (LNCS), pages 20–35, Springer International Publishing, 2015.

3. **Új bejárási algoritmusok és az állapottér korlátozások empirikus vizsgálata a szimbolikus végrehajtásban.**

   Ez a tézispont a szimbolikus végrehajtás futásidejű hibadetektálásra való alkalmazásához tartozik és a 5. fejezetben tárgyaltuk részletesen.

   A szimbolikus végrehajtó motor, amit a Szegedi Tudományegyetem Szoftverfejlesztés Tanszékén készítettünk komolyabb fejlesztéseken esett át, és az új *RTE-Hunter* nevet kapta. A szimbolikus végrehajtás elméletének megfelelően, az RTE-Hunter felépíti a lehetséges végrehajtási utak fáját, amit szimbolikus végrehajtási

fának nevezünk. Futásidejű problémákat úgy detektál az eszköz, hogy bejárja ezt a fát végrehajtva a szóban forgó útvonalakat, és amennyiben egy bizonyos feltétel teljesül, hibát jelez. A végrehajtási utak száma azonban exponenciálisan növekszik az elágazási pontok számával, vagyis a teljes végrehajtási fa bejárása lehetetlen a gyakorlatban. Ez a probléma azzal kezelhető, hogy különböző módokon korlátozzuk a fa méretét. Például a fában lévő szimbolikus programállapotok száma, a fa mélysége, vagy az elemzés időtartalma limitálható. Az RTEHunter-ben a maximális állapotok száma, a fa maximális mélysége, valamint a fabejáráshoz használt keresési stratégia konfigurálható. A fent leírt álltoprobbanás miatt a szimbolikus végrehajtás állapotterének korlátozása releváns kutatási téma, ha a gyakorlatban is alkalmazni kívánjuk a módszert. A célunk az, hogy megtaláljuk az RTEHunter optimális paraméterezését, vagyis azt, hogy mi az maximális állapotszám, maximális fa mélység, és mi az a bejárási stratégia, amellyel a lehető legtöbb futásidejű hibát meg tudjuk találni minél rövidebb idő alatt.

A három nyílt forrású Java rendszeren végzett empirikus vizsgálatok azt mutatják, hogy a szimbolikus végrehajtási fában lévő állapotok maximális száma közvetlenül befolyásolja az elemzési időt, de a futásidejű hibajelzések számát nem. Ugyanakkor a beállított maximális mélység nagyobb mértékben van hatással a talált hibák számára. Az optimális mélységkorlát különbözik a vizsgált rendszereken, de összességében azt a következtetést vonhatjuk le, hogy a 0 és 60-as mélységkorlát között gyakrabban fordulnak elő hibák, mint a fa mélyebb szintjein, habár mindez nagyban függ az alkalmazott bejárási stratégiától is. Kifejlesztettünk két olyan fabejárási stratégiát, amelyekkel a szimbolikus végrehajtást a hibára hajlamosabb kódrészek felé irányítjuk statikus és dinamikus információkat egyaránt felhasználva. Az úgynevezett null-heurisztikát használó bejárási stratégia 16 %-kal több hibát talál ugyanannyi idő alatt, mint az alapértelmezett mélységi bejárás. A lineáris regressziót használó stratégiával pedig több, mint kétszer annyi potenciális futásidejű hibát találtunk az ArgoUMl és a Jetspeed rendszerekben, mint az alapértelmezett bejárás.

A szerző hozzájárulása az eredményekhez. A szimbolikus végrehajtási fa maximális állapotszáma, az elemzés időtartama és talált futásidejű hibajelzések száma közötti kapcsolat empirikus úton történő megtalálása az RTEHunter számtalan futtatásával a vizsgált rendszereken a szerző munkája. Ő végzett el számos kísérleti elemzést annak érdekében, hogy megtaláljuk azt az optimális mélységkorlátot a vizsgált rendszereken, ahol a futásidejű hibák többsége detektálható. A bemutatott két fabejárási stratégia ötlete, azok megvalósítása és kiértékelése szintén a szerző érdeme.

A tézisponthoz tartozó publikáció:

- **I. Kádár.** The Optimization of a Symbolic Execution Engine for Detecting Runtime Errors. In *Acta Cybernetica*, pages 573–597, 2017.

II. rész - Refaktoring tevékenységek vizsgálata egy új refaktoring adatbázis alapján.

A disszertáció második részében arra keressük a választ, hogy a szoftverfejlesztők hogyan alkalmaznak refaktoring műveleteket a gyakorlatban. A kutatás elvégzéséhez

létrehoztunk egy adatbázist, amely nyílt forrású Java rendszereben talált refaktoringokat gyűjt össze. Az adatbázist nem csak azzal a céllal hoztuk létre, hogy a saját tanulmányunkhoz felhasználjuk. Publikusan elérhetővé tettük, támogatva ezzel a terület jövőbeli kutatásait.

4. **Refaktoring tevékenységek vizsgálata osztály és metódus szinten egy új publikus adatbázis felhasználásával.**

   Ezen tézispont kontribúciói a refaktoring műveletek vizsgálatához kapcsolódnak. A tézispont teljes kifejtése a 6. fejezetben található.

   A forráskód refaktoring egy népszerű és hatékony technika a forráskód belső minőségének javítására. A refaktoring fogalmát Fowler [29] vezette be, és napjainkban a forráskód refaktorálása lényeges szerepet tölt be a fejlesztésben az IT szakemberek egybehangzó véleménye szerint. Annak ellenére, hogy a refaktoring használata széles körben elfogadott az ipari szoftverfejlesztésben, több tanulmány is kimutatta, hogy a fejlesztők a Fowler által javasolt módszertől eltérően alkalmaznak refaktoring műveleteket a gyakorlatban. Emellett kevés olyan tanulmány lelhető fel, amely azt vizsgálja, hogy mik a refaktoring műveletek valós hatásai, és hogy a fejlesztők hogyan alkalmazzák azokat a gyakorlatban.

   Létrehoztunk egy olyan adatbázist, amely 7 nyílt forrású Java rendszer számos verziójára tartalmaz forráskódmetrikákat és verziók között elvégzett refaktoring műveleteket. Az adatbázis célja, hogy támogassa a refaktoring tevékenységek kutatását. Az adatbázis a Ref-Finder nyílt forrású refaktoring detektáló eszközzel [52] megtalált refaktoringokról tartalmaz pontos információkat, valamint több mint 50 féle forráskód metrikát 7 nyílt forrású Java rendszer 37 verziójára osztály és metódus szinten. Az adatbázis előállításához továbbfejlesztettük a Ref-Finder eszközt, hogy az a verziókezelő rendszer alapján több egymást követő verzió között automatikusan is képes legyen a refaktoringok meghatározásra. Ezen felül, azt is megvalósítottuk benne, hogy a refaktoringokat a pontos forráskód pozíciójukkal együtt exportálja meghatározott struktúrájú CSV (Comma Separated Values) fájlokba.

   Az adatbázis felhasználásával a karbantarthatóság és az elvégzett refaktoring műveletek közötti kapcsolatot kerestük, valamint megvizsgáltuk, hogy a refaktorálás milyen hatással van az egyes forráskódmetrikákra osztály és metódus szinten. Azt találtuk, hogy az alacsony karbantarthatósági mutatóval rendelkező osztályokat többször refaktorálták a gyakorlatban, mint jobb minőségi mutatóval rendelkezőket. A forráskódmetrikákat tekintve a klónok száma, a komplexitás és a csatoltságot jellemző metrikák javultak refaktorálás hatására, a kommentezettséget mérő metrikák azonban romlottak. Ezen felül, a méret alapú metrikák is jelentős csökkenést mutatnak. A metódusok szintjén, szintén azt állapíthatjuk meg, hogy az alacsonyabb karbantarthatóságú metódusokat jellemzően több refaktoring érinti a gyakorlatban. A klón lefedettség, a méret, és a metódusokból kimenő hívások száma csökkent a legintenzívebben a refaktoring által érintett metódusokban, ami arra utal, hogy a refaktoring műveletek ténylegesen mérséklik a negatív mutatójú forráskódjellemzőket, és karbantarthatóbb szoftverrendszert eredményeznek a gyakorlatban.

   A fenti vizsgálatokon felül, elvégeztük az említett refaktoring adatbázis manuális validációját, amivel kiszűrtük a Ref-Finder refaktoringnak nem tekinthető találá-

tait. Az így pontosított adatbázis felhasználásával számos aspektusát megvizsgáltuk a refaktoring műveletek által érintett és egyáltalán nem érintett osztályoknak és metódusoknak. A manuálisan validált adathalmazon kapott eredmények összhangban vannak az eredeti adatbázis felhasználásával kapottakkal, de azok konzisztensebbek és több rendszeren kaptunk szignifikáns eredményt. Ez azért lehet, mert eredeti adatbázisban meglévő hamis pozitív refaktoringok eltorzították a korábbi eredményt. A validált adatbázison végzett vizsgálatok azt mutatják, hogy a refaktoring által érintett forráskód elemek átlagos karbantarthatósága sokkal alacsonyabb volt a refaktorálás előtti verzióban, mint a refaktoring utáni kiadásban, azokhoz az osztályokhoz és metódusokhoz képest, amiket egyáltalán nem érintettek refaktoring műveletek. A méret, a komplexitás és a csatolás metrikák eloszlása mutatja a legnagyobb különbséget a refaktorált és a nem refaktorált elemek között az új adatbázis szerint.

A bemutatott refaktoring adatbázisok felhasználásában rejlő lehetőségek jelentősen túlmutatnak az általunk elvégzett vizsgálatokon, emiatt publikusan elérhetővé tettük azokat a PROMISE adattárházban [64] támogatva ezzel a komplexebb összefüggések felfedezésére irányuló jövőbeli kutatásokat.

A szerző hozzájárulása az eredményekhez. A szerző fejlesztette tovább a RefFinder refaktoring kereső eszközt, hogy az a verziókezelő rendszer alapján számos egymást követő verzió között automatikusan is képes legyen a refaktoringok meghatározásra, valamint arra, hogy a megtalált refaktoringokat a pontos pozícióinformációval együtt megfelelő struktúrában CSV fájlokba exportálja. A szerző az egyes refaktoring példányok kódelemekhez rendelésével is hozzájárult az adatbázis létrejöttéhez, valamint a manuális validáció felét is ő végezte el. A szerző részt vett az elemzési módszertan kidolgozásában és az eredmények kiértékelésében. A tézispont eredményeit alátámasztó publikációk:

- **I. Kádár**, P. Hegedűs, R. Ferenc, T. Gyimóthy. A Code Refactoring Dataset and Its Assessment Regarding Software Maintainability. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, pages 599–603. IEEE Computer Society, 2016.
- **I. Kádár**, P. Hegedűs, R. Ferenc, T. Gyimóthy. Assessment of the Code Refactoring Dataset Regarding the Maintainability of Methods. In *International Conference on Computational Science and Its Applications*, volume 9789 Lecture Notes in Computer Science (LNCS), pages 610–624. Springer International Publishing, 2016.
- **I. Kádár**, P. Hegedűs, R. Ferenc, T. Gyimóthy. A Manually Validated Code Refactoring Dataset and Its Assessment Regarding Software Maintainability. In *Proceedings of the The 12th International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 10–14. ACM, 2016.
- P. Hegedűs, **I. Kádár**, R. Ferenc, T. Gyimóthy. Empirical Evaluation of Software Maintainability Based on a Manually Validated Refactoring Dataset. Second revision submitted to journal *Information and Software Technology*. 44 pages Elsevier B.V.

Az B.1 táblázatban újra összefoglaljuk az egyes tézispontokat alátámasztó publikációkat.

$\mathcal{N}o.$	[96]	[94]	[97]	[93]	[98]	[99]	[95]
1	•	•					
2			•				
3				•			
4					•	•	•

B.1. táblázat. A tézispontokat alátámasztó publikációk

Bibliography

[1] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Commun. ACM*, 19(3):137–, March 1976.

[2] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970.

[3] Giuliano Antoniol, Massimiliano Di Penta, and Ettore Merlo. An Automatic Approach to identify Class Evolution Discontinuities. In *Proceedings of the 7th International Workshop on Principles of Software Evolution (IWPSE 2004), 6-7 September 2004, Kyoto, Japan*, pages 31–40. IEEE Computer Society, 2004.

[4] Roberta Arcoverde, Alessandro Garcia, and Eduardo Figueiredo. Understanding the Longevity of Code Smells: Preliminary Results of an Explanatory Survey. In *Proceedings of the 4th Workshop on Refactoring Tools*, WRT '11, pages 33–36, New York, NY, USA, 2011. ACM.

[5] T. Bakota, P. Hegedűs, P. Körtvélyesi, R. Ferenc, and T. Gyimóthy. A Probabilistic Software Quality Model. In *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 243–252, Sept. 2011.

[6] T. Bakota, P. Hegedüs, I. Siket, G. Ladányi, and R. Ferenc. QualityGate SourceAudit: A Tool for Assessing the Technical Quality of Software. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE 2014, Antwerp, Belgium, February 3-6, 2014*, pages 440–445, 2014.

[7] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba. An Experimental Investigation on the Innate Relationship Between Quality and Refactoring. *Journal of Systems and Software*, 107:1 – 14, 2015.

[8] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT – a Formal System for Testing and Debugging Programs by Symbolic Execution. In *Proceedings of the International Conference on Reliable Software*, pages 234–245, New York, NY, USA, 1975. ACM.

[9] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 183–198, New York, NY, USA, 2011. ACM.

[10] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 443–446, Washington, DC, USA, 2008. IEEE Computer Society.

[11] D. Bushnell, D. Giannakopoulou, P. Mehlitz, R. Paielli, and Corina S. Păsăreanu. Verification and Validation of Air Traffic Systems: Tactical Separation Assurance. In *Aerospace Conference, 2009 IEEE*, pages 1–10, 2009.

[12] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[13] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 322–335, New York, NY, USA, 2006. ACM.

[14] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic Execution for Software Testing in Practice: Preliminary Assessment. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1066–1071, New York, NY, USA, 2011. ACM.

[15] G. Ann Campbell and Patroklos P. Papapetrou. *SonarQube in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2013.

[16] Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, and George Candea. Selective Symbolic Execution. In *5th Workshop on Hot Topics in System Dependability (HotDep)*, 2009.

[17] E. Choi, N. Yoshida, and K. Inoue. What Kind of and How Clones Are Refactored?: A Case Study of Three OSS Projects. In *Proceedings of the 5th Workshop on Refactoring Tools*, WRT '12, pages 1–7, New York, NY, USA, 2012. ACM.

[18] E. Choi, N. Yoshida, and K. Inoue. An Investigation into the Characteristics of Merged Code Clones during Software Evolution. *IEICE Transactions on Information and Systems*, 97(5):1244–1253, 2014.

[19] Edmund Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of c and verilog programs using bounded model checking. In *Proceedings of the 40th annual Design Automation Conference*, pages 368–371. ACM, 2003.

[20] S. Counsell, X. Liu, S. Swift, J. Buckley, M. English, S. Herold, S. Eldh, and A. Ermedahl. An Exploration of the 'Introduce Explaining Variable' Refactoring. In *Proceedings of the XP2015 Scientific Workshop*, XP '15 workshops, pages 9:1–9:5, New York, NY, USA, 2015. ACM.

[21] P. David Coward. Symbolic Execution Systems – a Review. *Software Engineering Journal*, 3(6):229–239, November 1988.

[22] Cppcheck: Static source code analysis tool for c and c++ code. `https://sourceforge.net/projects/cppcheck/`. Accessed: 2017-08-26.

[23] Christoph Csallner and Yannis Smaragdakis. JCrasher: an Automatic Robustness Tester for Java. *Software Practice and Experience*, 34(11):1025–1050, September 2004.

[24] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Finding Refactorings via Change Metrics. *SIGPLAN Not.*, 35(10):166–177, October 2000.

[25] Thomas Dudziak and Jan Wloka. Tool-supported Discovery and Refactoring of Structural Weaknesses in Code. *Unpublished doctoral dissertation, Technical University of Berlin, Germany*, 2002.

[26] R. Ferenc, Á. Beszédes, M. Tarkiainen, and T. Gyimóthy. Columbus – Reverse Engineering Tool and Schema for C++. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM'02)*, pages 172–181. IEEE Computer Society, IEEE Computer Society, oct 2002.

[27] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for esc/java. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, FME '01, pages 500–517, London, UK, UK, 2001. Springer-Verlag.

[28] F. A. Fontana and S. Spinelli. Impact of Refactoring on Quality Code Evaluation. In *Proceedings of the 4th Workshop on Refactoring Tools*, WRT '11, pages 37–40, New York, NY, USA, 2011. ACM.

[29] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Longman Publishing Co., Inc., 1999.

[30] Xiang Fu and Kai Qian. SAFELI: SQL Injection Scanner Using Symbolic Execution. In *Proceedings of the 2008 Workshop on Testing, Analysis, and Verification of Web Services and Applications*, TAV-WEB '08, pages 34–39, New York, 2008. ACM.

[31] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification*, CAV'07, pages 519–531, Berlin, Heidelberg, 2007. Springer-Verlag.

[32] Xi Ge, Quinton L. DuBose, and Emerson R. Murphy-Hill. Reconciling Manual and Automatic Refactoring. In Martin Glinz, Gail C. Murphy, and Mauro Pezzè, editors, *Proceedings of the International Conference on Software Engineering*, pages 211–221. IEEE Computer Society, 2012.

[33] Gecode Tool-set. `http://http://www.gecode.org/`.

[34] Adnane Ghannem, Ghizlane El Boussaidi, and Marouane Kessentini. Model Refactoring Using Examples: a Search-based Approach. *Journal of Software: Evolution and Process*, 26(7):692–713, 2014.

[35] Patrice Godefroid. Compositional dynamic test generation. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 47–54, New York, NY, USA, 2007. ACM.

[36] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. *SIGPLAN Not.*, 43(6):206–215, June 2008.

[37] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[38] Michael W. Godfrey and Lijie Zou. Using Origin Analysis to Detect Merging and Splitting of Source Code Entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, February 2005.

[39] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. Search based software engineering: A comprehensive analysis and review of trends techniques and applications. *Department of Computer Science, King's College London, Tech. Rep. TR-09-03*, 2009.

[40] P. Hegedűs, T. Bakota, G. Ladányi, Cs. Faragó, and R. Ferenc. A Drill-Down Approach for Measuring Maintainability at Source Code Element Level. *Electronic Communications of the EASST*, 60, 2013.

[41] Melinda R Hess and Jeffrey D Kromrey. Robust Confidence Intervals for Effect Sizes: a Comparative Study of Cohen's d and Cliff's delta Under Non-normality and Heterogeneous Variances. In *Annual Meeting of the American Educational Research Association*, pages 1–30, 2004.

[42] M. I. Hoque, V. N. Ranga, A. R. Pedditi, R. Srinath, M. A. A. Rana, M. E. Islam, and A. Somani. An Empirical Study on Refactoring Activity. *ACM Computing Research Repository*, abs/1412.6359, 2014.

[43] David Hovemeyer and William Pugh. Finding bugs is easy. *ACM Sigplan Notices*, 39(12):92–106, 2004.

[44] Christian Igel, Verena Heidrich-Meisner, and Tobias Glasmachers. Shark. *Journal of Machine Learning Research*, 9:993–996, 2008.

[45] Petri Ihantola. Test Data Generation for Programming Exercises with Symbolic Execution in Java PathFinder. In *Proceedings of the 6th Baltic Sea Conference on Computing Education Research*, Baltic Sea '06, pages 87–94, New York, 2006. ACM.

[46] Konrad Jamrozik, Gordon Fraser, Nikolai Tillman, and Jonathan Halleux. Generating Test Suites with Augmented Dynamic Symbolic Execution. In *Tests and Proofs*, volume 7942 of *Lecture Notes in Computer Science*, pages 152–167. Springer Berlin Heidelberg, 2013.

[47] Java PathFinder Tool-set. `http://babelfish.arc.nasa.gov/trac/jpf`.

[48] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya. A Quantitative Evaluation of Maintainability Enhancement by Refactoring. In *Proceedings of the International Conference on Software Maintenance*, pages 576–585, 2002.

[49] Foutse Khomh, Massimiliano Di Penta, and Yann-Gael Gueheneuc. An Exploratory Study of the Impact of Code Smells on Software Change-proneness. In *Proceedings of the 16th Working Conference on Reverse Engineering*, pages 75–84. IEEE, 2009.

[50] Yit Phang Khoo, Bor-Yuh Evan Chang, and Jeffrey S. Foster. Mixing type checking and symbolic execution. In Benjamin G. Zorn and Alexander Aiken, editors, *PLDI*, pages 436–447. ACM, 2010.

[51] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. Generalized Symbolic Execution for Model Checking and Testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'03, pages 553–568, Berlin, Heidelberg, 2003. Springer-Verlag.

[52] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit. Ref-Finder: a Refactoring Reconstruction Tool Based on Logic Query Templates. In *Proceedings of the 18th ACM SIGSOFT international symposium on Foundations of software engineering (FSE'10)*, pages 371–372, 2010.

[53] James C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[54] Yasemin Kosker, Burak Turhan, and Ayse Bener. An Expert System for Determining Candidate Software Classes for Refactoring. *Expert Systems with Applications*, 36(6):10000–10003, August 2009.

[55] Ted Kremenek. Finding software bugs with the clang static analyzer. *Apple Inc*, 2008.

[56] Kiran Lakhotia, Phil McMinn, and Mark Harman. An empirical investigation into branch coverage for c programs using {CUTE} and {AUSTIN}. *Journal of Systems and Software*, 83(12):2379 – 2391, 2010.

[57] Hui Liu, Qiurong Liu, Yang Liu, and Zhouding Wang. Identifying Renaming Opportunities by Expanding Conducted Rename Refactorings. *IEEE Transactions on Software Engineering*, vol. 41(9):887–900, 2015.

[58] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. Directed symbolic execution. In *Proceedings of the 18th International Conference on Static Analysis*, SAS'11, pages 95–111, Berlin, Heidelberg, 2011. Springer-Verlag.

[59] Rim Mahouachi, Marouane Kessentini, and Mel Ó Cinnéide. Search-based Refactoring Detection using Software Metrics Variation. In *International Symposium on Search Based Software Engineering*, pages 126–140. Springer, 2013.

[60] Rupak Majumdar and Ru-Gang Xu. Directed test generation using symbolic grammars. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 134–143, New York, NY, USA, 2007. ACM.

[61] H. B. Mann and D. R. Whitney. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *Ann. Math. Statist.*, 18(1):50–60, 03 1947.

[62] Mika Mantyla, Jari Vanhanen, and Casper Lassenius. A Taxonomy and an Initial Empirical Study of Bad Smells in Code. In *Proceedings of the 2003 International Conference on Software Maintenance, ICSM 2003.*, pages 381–384. IEEE, 2003.

[63] Thomas J. McCabe. A complexity measure. In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, pages 407–, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[64] T. Menzies, R. Krishna, and D. Pryor. The Promise Repository of Empirical Software Engineering Data, 2015.

[65] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33(1):2–13, 2007.

[66] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 181–190, New York, NY, USA, 2008. ACM.

[67] Frederick Mosteller. Association and estimation in contingency tables. *Journal of the American Statistical Association*, 63(321):1–28, 1968.

[68] A. Murgia, R. Tonelli, M. Marchesi, G. Concas, S. Counsell, J. McFall, and S. Swift. Refactoring and its Relationship with Fan-in and Fan-out: An Empirical Study. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 63–72, March 2012.

[69] E. Murphy-Hill, C. Parnin, and A. P. Black. How We Refactor, and How We Know It. *IEEE Transactions on Software Engineering*, 38(1):5–18, Jan 2012.

[70] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson, and Danny Dig. A Comparative Study of Manual and Automated Refactorings. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 552–576, Berlin, Heidelberg, 2013. Springer-Verlag.

[71] P. Oman and J. Hagemeister. Metrics for Assessing a Software System's Maintainability. In *Proceedings of the International Conference on Software Maintenance*, pages 337–344. IEEE CS Press, 1992.

[72] R. Peters and A. Zaidman. Evaluating the Lifespan of Code Smells using Software Repository Mining. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 411–416, March 2012.

[73] Pmd/java. `https://pmd.github.io/`. Accessed: 2017-03-21.

[74] K. Prete, N. Rachatasumrit, N. Sudan, and K. Miryung. Template-based Reconstruction of Complex Refactorings. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance (ICSM)*, pages 1–10, Sept 2010.

[75] Corina S. Păsăreanu and Neha Rungta. Symbolic PathFinder: Symbolic Execution of Java Bytecode. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 179–180, New York, NY, USA, 2010. ACM.

[76] Corina S. Păsăreanu, Neha Rungta, and Willem Visser. Symbolic execution with mixed concrete-symbolic solving. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 34–44, New York, NY, USA, 2011. ACM.

[77] Corina S. Păsăreanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing NASA Software. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 15–26, New York, NY, USA, 2008. ACM.

[78] Jacek Ratzinger, Thomas Sigmund, and Harald C. Gall. On the Relation of Refactorings and Software Defect Prediction. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, MSR '08, pages 35–38, New York, NY, USA, 2008. ACM.

[79] Barbara G. Ryder, Donald Smith, Ulrich Kremer, Michael Gordon, and Nirav Shah. A Static Study of Java Exceptions Using JESP. In *Proceedings of the Ninth International Conference on Compiler Construction*, volume 1781 of *Lecture Notes in Computer Science*, pages 67–81. Springer-Verlag, 2000.

[80] Koushik Sen and Gul Agha. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-checking Tools. In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, pages 419–423, Berlin, 2006. Springer-Verlag.

[81] Daryl Shannon, Daiqian Zhan, Sukant Hajra, Alison Lee, and Sarfraz Khurshid. Abstracting symbolic execution with string analysis testing. In *Academic and Industrial Conference Practice and Research TechniquesMUTATION, 2007. TAICPART-MUTATION*, 2007.

[82] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. Why We Refactor? Confessions of GitHub Contributors. *CoRR*, abs/1607.02459, 2016.

[83] Saurabh Sinha, Hina Shah, Carsten Görg, Shujuan Jiang, Mijung Kim, and Mary Jean Harrold. Fault Localization and Repair for Java Runtime Exceptions. In *Proceedings of the 18th International Symposium on Software Testing and Analysis*, ISSTA '09, pages 153–164, New York, NY, USA, 2009. ACM.

[84] Gustavo Soares, Bruno Catao, Catuxe Varjao, Solon Aguiar, Rohit Gheyi, and Tiago Massoni. Analyzing Refactorings on Software Repositories. In *Proceedings of the 25th Brazilian Symposium on Software Engineering, SBES, Sao Paulo, Brazil, September 28-30*, pages 164–173, 2011.

[85] Gustavo Soares, Rohit Gheyi, Emerson Murphy-Hill, and Brittany Johnson. Comparing Approaches to Analyze Refactoring Activity on Software Repositories. *Journal of Systems and Software*, 86(4):1006 – 1022, 2013.

[86] Quinten David Soetens, Javier Pérez, and Serge Demeyer. An Initial Investigation into Change-Based Reconstruction of Floss-Refactorings. In *Proceedings of the International Conference on Software Maintenance*, pages 384–387. IEEE Computer Society, 2013.

[87] JaeSeung Song, Tiejun Ma, Cristian Cadar, and Peter Pietzuch. Rule-Based Verification of Network Protocol Implementations Using Symbolic Execution. In *Proceedings of the 20th IEEE International Conference on Computer Communications and Networks (ICCCN'11)*, pages 1–8, 2011.

[88] Matheus Souza, Mateus Borges, Marcelo d'Amorim, and Corina S. Păsăreanu. CORAL: Solving Complex Constraints for Symbolic Pathfinder. In *Proceedings of the Third International Conference on NASA Formal Methods*, NFM'11, pages 359–374, Berlin, Heidelberg, 2011. Springer-Verlag.

[89] Reinout Stevens, Coen De Roover, Carlos Noguera, and Viviane Jonckers. A History Querying Tool and Its Application to Detect Multi-version Refactorings. In Anthony Cleve, Filippo Ricca, and Maura Cerioli, editors, *Proceedings of the 17th European Conference on Software Maintenance and Reengineering*, pages 335–338. IEEE Computer Society, 2013.

[90] Gábor Szőke, Csaba Nagy, Lajos Jenő Fülöp, Rudolf Ferenc, and Tibor Gyimóthy. FaultBuster: an Automatic Code Smell Refactoring Toolset. In Michael W. Godfrey, David Lo, and Foutse Khomh, editors, *SCAM*, pages 253–258. IEEE Computer Society, 2015.

[91] Ladan Tahvildari and Kostas Kontogiannis. A Metric-based Approach to Enhance Design Quality Through Meta-pattern Transformations. In *Proceedings of the 7th European Conference on Software Maintenance and Reengineering*, pages 183–192. IEEE, 2003.

[92] Kunal Taneja, Danny Dig, and Tao Xie. Automated Detection of Api Refactorings in Libraries. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 377–380, New York, NY, USA, 2007. ACM.

[93] **István Kádár**. The optimization of a symbolic execution engine for detecting runtime errors. *Acta Cybernetica*, 23(2):573–597, 2017.

[94] **István Kádár**, Péter Hegedűs, and Rudolf Ferenc. Runtime exception detection in java programs using symbolic execution. *Acta Cybernetica*, 21(3):331–352, 2014.

[95] **István Kádár**, Péter Hegedűs, Rudolf Ferenc, and Tibor Gyimóthy. A Manually Validated Code Refactoring Dataset and Its Assessment Regarding Software Maintainability. In *Proceedings of the 12th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE 2016, pages 10:1–10:4, New York, NY, USA, 2016. ACM.

[96] **István Kádár**, Péter Hegedűs, and Rudolf Ferenc. Runtime exception detection in java programs using symbolic execution. In *Proceedings of the 13th Symposium on Programming Languages and Software Tools, SPLST'13*, page 215–229, Szeged, 2013. University of Szeged, University of Szeged.

[97] **István Kádár**, Péter Hegedűs, and Rudolf Ferenc. Adding constraint building mechanisms to a symbolic execution engine developed for detecting runtime errors. In *International Conference on Computational Science and Its Applications*, pages 20–35. Springer, 2015.

[98] **István Kádár**, Péter Hegedűs, Rudolf Ferenc, and Tibor Gyimóthy. A Code Refactoring Dataset and Its Assessment Regarding Software Maintainability. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 599–603. IEEE, March 2016.

[99] **István Kádár**, Péter Hegedűs, Rudolf Ferenc, and Tibor Gyimóthy. *Assessment of the Code Refactoring Dataset Regarding the Maintainability of Methods*, chapter Computational Science and Its Applications – ICCSA 2016: 16th International Conference, Beijing, China, July 4-7, 2016, Proceedings, Part IV, pages 610–624. Springer International Publishing, 2016.

[100] Nikolai Tillmann and Jonathan De Halleux. Pex: White Box Test Generation for .NET. In *Proceedings of the 2nd International Conference on Tests and Proofs*, TAP'08, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag.

[101] Tom Tourwe and Tom Mens. Identifying Refactoring Opportunities Using Logic Meta Programming. In *Proceedings of the 7th European Conference on Software Maintenance and Reengineering*, pages 91–100, 26-28 March 2003.

[102] N. Tsantalis, V. Guana, E. Stroulia, and A. Hindle. A Multidimensional Empirical Study on Refactoring Activity. In *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '13, pages 132–146, Riverton, NJ, USA, 2013. IBM Corporation.

[103] E. van Emden and L. Moonen. Java Quality Assurance by Detecting Code Smells. In *Proceedings of the 9th Working Conference on Reverse Engineering*, pages 97–106, 2002.

[104] Filip Van Rysselberghe and Serge Demeyer. Reconstruction of Successful Software Evolution Using Clone Detection. In Tommi Mikkonen, Michael W. Godfrey, and Motoshi Saeki, editors, *Proceedings of the International Workshop on Principles of Software Evolution*, pages 126–130. IEEE Computer Society Press, 2003.

[105] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with Java PathFinder. *SIGSOFT Softw. Eng. Notes*, 29(4):97–107, July 2004.

[106] Markus von Detten. Towards Systematic, Comprehensive Trace Generation for Behavioral Pattern Detection Through Symbolic Execution. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, PASTE '11, pages 17–20, New York, NY, USA, 2011. ACM.

[107] Wei Wang and Michael W Godfrey. Recommending Clones for Refactoring Using Design, Context, and History. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 331–340. IEEE, 2014.

[108] Westley Weimer and George C. Necula. Finding and Preventing Run-time Error Handling Mistakes. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 419–431, New York, NY, USA, 2004. ACM.

[109] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'05, pages 365–381, Berlin, Heidelberg, 2005. Springer-Verlag.

[110] Zhenchang Xing and Eleni Stroulia. Refactoring Detection based on UMLDiff Change-Facts Queries. In *Proceedings of the Working Conference on Reverse Engineering*, volume 6, pages 263–274. Citeseer, 2006.

[111] Aiko Fallas Yamashita and Leon Moonen. Do Developers Care about Code Smells? An Exploratory Survey. In *Proceedings of the 2013 Working Conference on Reverse Engineering, WCRE*, volume 13, pages 242–251, 2013.