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

István Kádár

Department of Software Engineering
University of Szeged

Szeged, 2017

Supervisor:

Dr. Rudolf Ferenc

Summary of the Ph.D. thesis submitted for the degree of Doctor of Philosophy

of the University of Szeged

University of Szeged

PhD School in Computer Science

Introduction

It is a big challenge in software engineering to produce huge, reliable and robust software systems. In industry, developers typically have to focus on solving problems quickly. The importance of code quality in time pressure is frequently secondary. However, software code quality is very important, because a too complex, hard-to-maintain code results in more bugs, and makes 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 to make 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 activity*. 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). The main usages of symbolic execution are generating inputs of program failure and high-coverage test cases. It 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 were not detected 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. One of the greater challenges in symbolic execution is the very large number of possible execution paths, which increas exponentially.

Our research proposes approaches to 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. The concept of refactoring was introduced by Martin Fowler. He originally proposed that detecting 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 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 subjected the dataset to an analysis of the effects of code refactoring on source code metrics and maintainability, which are primary quality attributes in software development.

The thesis consists of four thesis points organized into two parts. In this booklet, we summarize the results of each thesis point in a separate section.

Part I
Advances in Symbolic Execution for Runtime Error Detection

Background of Symbolic Execution

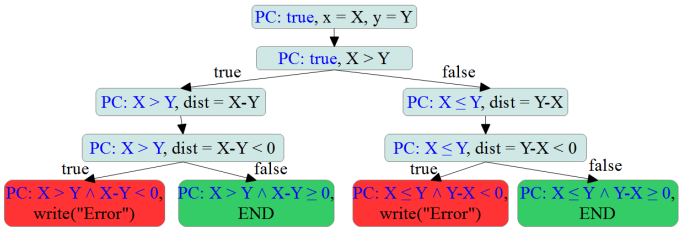
Symbolic execution [15] assumes that programs are operated on symbolic variables instead of specific input data, and the output is a function of these symbolic variables. A symbolic variable is a set of possible values of a concrete variable in the program, thus a symbolic state is a set of concrete states. When 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 true or false. The execution continues on both branches accordingly. This way, we can simulate all possible execution branches of the program.

During symbolic execution, we maintain a so-called *path condition (PC)*. 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 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.

All 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 1.

```
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 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

The symbolic execution of the code is illustrated on Figure 1 (b) with the corresponding symbolic execution tree. We handle x and y symbolically, their symbols are X and Y respectively. 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:

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

In line 8, the execution branches similarly.

Thesis Point 1: A Method-level Symbolic Execution Technique for Runtime Error Detection in Real-world Software Systems

Most runtime failures of a software system can only be revealed during test execution only, which has a very high cost. 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 automatically and finding dangerous parts in the source code, that could behave as time-bombs during further development.

We use symbolic execution [15] to implement the approach that is able to explore the possible execution paths of the program. We used Symbolic PathFinder [21] (SPF) as the symbolic execution engine, which is an extension of the Java PathFinder [13] (JPF) execution environment (a special Java Virtual Machine) that aims to verify Java programs.

The runtime environment we implemented on the top of Symbolic PathFinder starts the symbolic execution for every method of an arbitrary Java system keeping the state space reduced. If 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.

The concept of the developers of Symbolic PathFinder was to start running the program in normal mode like in a real life environment, then switch to symbolic execution mode at given points, e.g. at more complex or problematic parts in the program [22]. 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, as it requires prior knowledge or preliminary work. Another disadvantage is that you have to run the program manually to make the control reach the methods which will be analyzed symbolically.

By contrast, the tool we have developed is able to execute symbolically an arbitrary method or all methods of a program. 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 its parts separately. The approach also makes it possible to analyze libraries that do not have a *main* method such as log4j logging library. One of the major disadvantages is 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 all methods separately. 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.

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. We implemented this algorithm by extending the Symbolic PathFinder using its notification interface.

Besides small example codes, we evaluated our algorithm on three open source systems and found

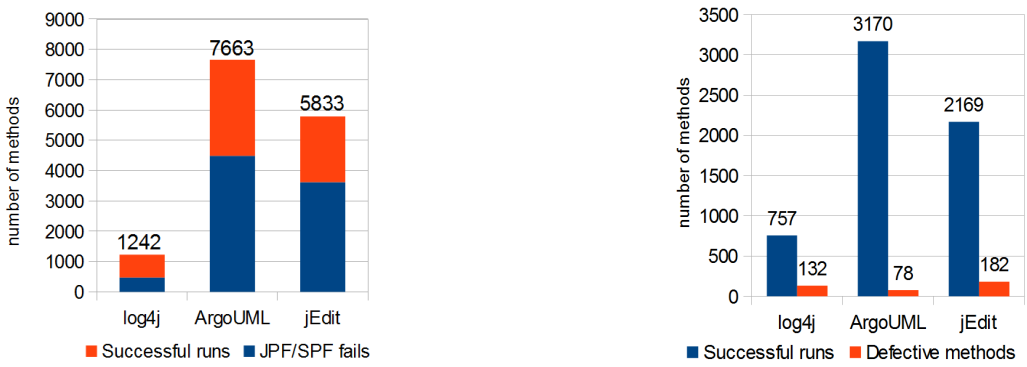


Figure 2. (a) The number of methods examined in the programs and the number of JPF or SPF faults (b) The number of successfully analyzed methods and the number of defective methods

multiple runtime issues. The real-world subject systems we analyzed are the log4j logging library, the ArgoUML modeling tool, and the jEdit text editor program. We found multiple errors in the log4j system that were also reported as real bugs in its bug tracking system.

Figure 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 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 2 (b) shows the number of successfully analyzed methods and the methods with one or more runtime exceptions.

It is important to note, that not all indicated exceptions are real errors. This is because the analysis was performed in an artificial execution environment which might have introduced false positive hits.

We prove the validity of the detected exceptions with the bug reports found in the bug tracking systems of these projects that describe program faults caused by the runtime exceptions that were also found by our tool. Here we provide the list of these bugreports:

- The first affected bug reports the termination of an application using log4j version 1.2.14 caused by a NullPointerException.

  URL: 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 explains that the runtime exception causing the halt comes from method *org.apache.log4j.NDC.remove()*, at line 377.

  URL: https://issues.apache.org/bugzilla/show_bug.cgi?id=45335

- Also in log4j version 1.2.15 we found an error 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.

  URL: https://issues.apache.org/bugzilla/show_bug.cgi?id=46271

The author's contributions to the results 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 with the bug reports found in the bug tracking systems of the analyzed projects was also the author's task. He contacted the authors of Symbolic PathFinder several times to report some blocker issues that held back the research.

Thesis Point 2: A Constraint Building Mechanism for Symbolic Execution to Improve Runtime Error Detection Accuracy

Symbolic Checker, the symbolic execution engine developed at the Software Engineering Department of 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 (PC).

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 as well. Thus, we can track the dependencies of the symbolic variables extending the original principles of path condition building.

The main idea behind the developed constraint building mechanism is that if during analysis the program sets up conditions (constraints) that unambiguously determine the value of one or more symbolic variables, we can then convert these symbols into concrete values and the symbolic execution can be continued on the actual path using these concreted variables. Since these variables are handled as concrete data, it is possible to detect errors that could not be revealed with a conventional constraint building mechanism, because in symbolic execution we fire an issue that is the effect of concrete values. If a variable is a symbol it means that its value is doubtful, not known. For example, 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.

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 contains the dependencies of the symbolic variables too 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 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. This way, false positive defects can be eliminated.

The code snippet in Listing 1 shows an example where the extended PC has some gains.

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

Executing the code in Listing 1 symbolically 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 and the dependencies of symbol *a*; in other words, those constraints that are given by the assignments that defines variable *a*. After satisfying this constraint system, it can be obtained that *a* can only be 9, which implies that the value of b and c symbols are unambiguous too: $b = 0.0$ and $c = -2$. In the example above, if the execution continues with the b=0.0 value at line 8 a division by zero error can be detected. As long as symbol *b* would not be included in the PC, and if its unambiguous value would not be used, the detection of division by zero would fail.

In real-life programs, big constraint systems are be built, which contain lots of symbols. It is easy to see that by satisfying such a large set of constraints as a whole, has a low probability for only one possible solution. To overcome the problem above, we decompose the path condition into connected components. That is, we take apart the constraint sets that are independent, in terms of they do not contain the same variables. The connected components can be satisfied individually and if some of them determine a variable unambiguously then 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.

To build and satisfy the constraint systems, we used the open-source Gecode constraint satisfaction tool-set [11].

As a result, runtime errors can be detected that would not be possible using a conventional symbolic execution tool. 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 from a conventional symbolic execution tool and Symbolic Checker without using the novel constraint building mechanism. We also

found potential runtime errors in large, real-life systems that would not be possible with the traditional constraint building mechanism.

The author's contributions to the results 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.

Thesis point 3: Novel Search Strategies for Symbolic Execution and Empirical Investigation of State Space Limitations

Our symbolic execution engine at the Department of Software Engineering at the University of Szeged went through further developments and has 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. 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 was 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 faster.

The empirical investigations on three open-source Java systems (ArgoUML, JetSpeed and JFreeChart) showed that adjusting the maximum number of states for the symbolic execution trees directly impacts execution time but not the number of found issues. The number of executed states is highly correlated to the analysis time. The Pearson-correlation coefficients are all above 0.99, which is a strong positive correlation meaning that high state limit goes with high run-time. The results are significant at $p < 0.05$. On the other hand, the correlation coefficients between the number of states and the number of found issues are spread from 0.3 to 0.8 indicating a weaker relationship, moreover, the results are not significant in many cases at $p < 0.05$. Hence, the number of states determines the execution time, but the number of errors probably depends on other factors too.

To understand the role of maximum depth, we run RTEHunter with different depth limits, on each depth limit level with different maximum state sizes, which enables the investigation of how the results change by increasing the analysis time. The investigated depth limits are 50, 100, 200, 400, 600 and 800, the maximum state numbers differ from 200 to 10,000 for each. The results are depicted with line diagrams such as in Figure 3, which shows how the number of errors grows in time on each depth limit level. 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.

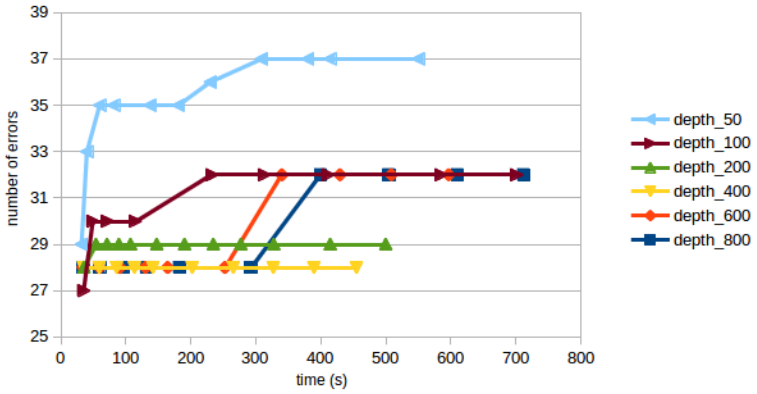
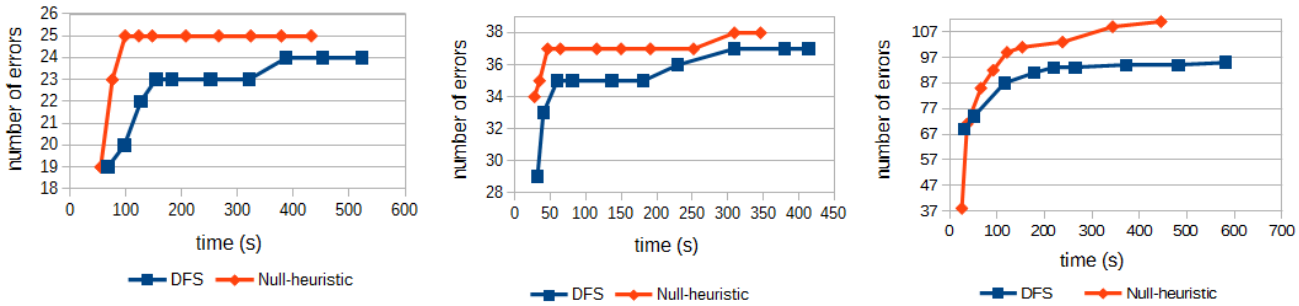


Figure 3. The growth of the error number upon analysis time using different depth limits in Jetspeed



(a) (b) (c)

Figure 4. The efficiency of null-heuristic search compared to the default depth-first search on ArgoUML (a), Jetspeed (b), JFreeChart (c)

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.

Null-heuristic search strategy intends to drive the traversal to find more null pointer dereference issues. For each symbolic state, we summarize the number of reachable reference-type values (variable values, literals, function return values, etc.) with *null* value at the current symbolic state of the analyzed Java program. To continue the traversal, the engine chooses the state with the highest number of null values attributing higher probability to finding possible null pointer dereferences in that state and in the sub-tree available from it. The null-heuristic search strategy performs better than the default depth-first search by finding upto 16 % more errors within the same time frame. The results are depicted in Figure 4.

We also developed a search strategy that supports the detection of not only 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, and durring the traversal it always chooses the state with the highest score. The score is the estimated number of runtime issues that might have been detected in the sub-tree reachable from the state.

The training data of the linear regression model contains one training example for each state from symbolic execution trees that are traversed before by the engine. The label (i.e. the supervisory signal) to each example is the number of runtime issues that were detected in a previous RTEHunter run in the sub-tree under the state that the example belongs to.

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

We defined five attributes as predictors for each state:

1. *The depth of the state in the symbolic execution tree.* If a significant part of the faults tends to occur in a given depth, the information will be coded into the model.
2. *The number of null values in the state*
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.*
4. *The Logical Lines of Code (LLOC) metric of the method that the state belongs to.*
5. *The cyclomatic complexity metric [17] of the method that the state belongs to.*

In the evaluation of the linear regression-based search strategy, we used 10-fold cross-validation on each system. This strategy also outperforms DFS. It detects more than twice as many errors in ArgoUML and Jetspeed as seen in Table 1.

The author's contributions to the results 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 found issues by running RTEHunter many times on three open source systems. He also performed many experiments to find the optimal depth limits in case of each system and revealed the depth level where most of the errors are found. The idea of the two search strategies for the state space exploration, their implementation and the entire evaluation are the author's work.

Part II
Investigation of Refactoring Activities Based on a New Dataset

Thesis Point 4: Assessment of Refactoring Activities on Classes and Methods Based on a New Public Dataset

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 [10] and IT practitioners

today 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 [8, 19, 23], 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 with the intention 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 [14] and more than 50 types of source code metrics for 37 releases of 7 open-source systems at class- and method-level. In addition to source code metrics, the dataset includes the relative maintainability indices of source code elements, calculated by the *QualityGate*[1] tool, an implementation of the *ColumbusQM quality model* [9]. 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.

We applied the dataset to 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?*

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. Since we cannot assume anything about the distribution of the maintainability indices nor the number of refactorings, we performed a Spearman rank correlation analysis.

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 ran a Mann-Whitney U test [16], which is a non-parametric statistical test to analyze whether the distribution of the values differs significantly between two groups. The p-value of the test helped us judge whether there is significant difference in the metric decreases between the sourcecode entities subjected to refactoring and the entities unaffected by refactoring.

The result of this test gave us a hint on what are the 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 [12]. Cliff's δ measures how often the values in one distribution are larger than the values in a second distribution. Simply put, 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 in the case of negative value the metric value differences are higher in the non-refactored elements.

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. Considering method-level, we also found that methods with poor

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

maintainability are subject to more refactorings in practice than methods with higher technical quality. Clone coverage, size metrics and number of outgoing invocations decreased most intensively in the methods subjected to frequent refactorings, which might indicate that performing code refactoring in practice indeed mitigates unwanted code characteristics such as clones, size, or coupling, and result in more maintainable software systems.

According to the authors of Ref-Finder, the precision of the tool is 79% [20], however, our analysis showed 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. This contains one manually validated release for each of the 7 systems from which we expect better quality results. The number of all and manually validated refactorings with precision information for each subject system is shown in Table 2.

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

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 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.

The possible utilization of the assembled datasets goes way beyond our investigations. We made them publicly available at the PROMISE data repository [18] to encourage the research community to reveal more complex phenomena in connection with practical refactorings.

The author's contributions to the results 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.

Summary

In this thesis, we were concerned 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 we made it more efficient by a novel constraint building mechanism, and investigated the possible state space limitations while giving new algorithms to prioritize the paths to explore. As a result of this research work, we developed methods and a tool that are 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, and the connection with other quality attributes (source code metrics) is also 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 way beyond the 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.

In Table 3 we summarize the publications related to each thesis point.

$\mathcal{N}o.$	[4]	[2]	[5]	[1]	[6]	[7]	[3]
1	•	•					
2			•				
3				•			
4					•	•	•

Table 3. Thesis contributions and supporting publications

Acknowledgments

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 considered 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 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.

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

István Kádár, oktober 2017

The Author's Publications

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

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

[3] **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.

[4] **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.

[5] **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.

[6] **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.

[7] **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.

References

[8] 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.

[9] 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.

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

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

[12] 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.

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

[14] 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.

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

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.