MŰEGYETEM 1782

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

On Supporting Automatic Test Generation

Master's Thesis

Author
Dávid Honfi

Advisor
Zoltán Micskei, PhD
András Vörös

2015. május 24.

Contents

Kivonat

Manapság a szoftverek egyre komplexebb felépítése új kihívások elé állítják a mérnököket. A verifikációs módszerek egyre fontosabb szerephez jutnak a tervezés és implementáció fázisában, hiszen a növekvő méret és komplexitás nagyobb valószínűséggel előforduló hibákhoz vezethet. A fejlesztés során végzett átfogó tesztelés kulcsfontosságú a szoftverminőségben, habár a megfelelő bemenetek kiválasztása a teszteléshez meglehetősen nehéz feladat valós környezetekben.

Az automatikus tesztbemenet-generálás megkönnyítheti ezen bemenetkiválasztási feladatot. A generálási folyamat építhet egy viselkedési modellre vagy akár magára a forráskódra is. A dinamikus szimbolikus végrehajtás technikája ez utóbbiakhoz tartozik: a kódot szimbolikus változók segítségével értelmezi és vezérli, majd az ezekből kapott eredményeket konkrét végrehajtások segítségével finomítja. A módszer hatékonyan képes bonyolult forráskódok végrehajtási útvonalait is feltérképezni. Mindezek ellenére a módszer ipari alkalmazása a komplexitásából és a tesztgenerálást végző személy számára rendelkezésre álló információk kis mennyisége miatt nehézkes válik. A dolgozatban két saját módszert mutatok be a szimbolikus végrehajtás alapú tesztbemenet-generálás ipari alkalmazásának megkönnyítéséhez.

Az első általam kidolgozott technika a szimbolikus végrehajtás vizualizációja, amely képes a végrehajtás egyes lépéseit egy bejárási fa csomópontjaihoz társítani segédinformációk hozzáadásával (pl. forráskódhely, útvonalfeltétel). A technika a futás átláthatóságát 1) az útvonalfeltételek szétbontása, 2) a végrehajtási útvonalak osztályozása és a 3) tesztelt komponensből való kilépések érzékelése segítségével biztosítja. A technika implementációjaként elkészült egy eszköz is, amely nyílt forráskódú és online is elérhető.

A második kidolgozott technika az automatikus izolációs környezet generálása, amely automatikusan képes a tesztelés alatt álló egység külső hívásait helyettesíteni. A dolgozatban bemutatott módszer észleli a vizsgált egységből kifele történő hívásokat, majd több szempont alapján megvizsgálja azokat. Az ebből kapott eredmények segítségével pedig forráskódot generál az izolációs környezet felépítéséhez a külső hívások helyettesítésére.

Mindkét kidolgozott technikához elkészült egy-egy prototípus implementáció, amelyeket megvizsgáltam egyszerű és valós példákon keresztül is.

Abstract

Nowadays, complex software systems pose new challenges for engineers. Verification techniques are getting an important role during the design and implementation phase, since the increasing size and complexity of software leads to increased possibility of defects. Thorough testing during development is a key for the improvement of software quality, however choosing the appropriate test inputs can be a rather complex task in real-world environments.

Automation of test input generation can support the task of selecting the proper inputs. The test input generation procedure can be based on a behavioral model or on the source code itself. Dynamic symbolic execution is one of the latter techniques: it executes the source code symbolically (with symbolic variables and values) and refines the results with constraints derived from concrete executions. The technique is able to effectively discover execution traces of complicated software. However, its industrial application is hindered by the complexity of the technique and the lack of provided information for the test engineer. In this thesis, I present two techniques that are able to alleviate the work with automated test input generation based on symbolic execution.

The first presented technique is the visualization of symbolic execution, which maps the steps of the execution to nodes of an execution tree and enriches it with metadata information (e.g. source code locations, path conditions). The technique supports the better understanding of the execution by 1) decomposing the path conditions of each trace, 2) classifying the execution paths and 3) identifying situations when a trace exits from the unit under test. A tool was implemented and it is open-source and available online.

The second technique to support the industrial usage of symbolic execution is the automated isolation environment generation. Thus, it can automatically replace the external dependencies with doubles. The presented technique discovers the calls going outside from the unit under test, while analyzing them from several aspects. From the results of this analysis, the technique is able to generate source code for unit isolation.

Prototype implementations of both of the techniques were developed and examined through simple and also real-world examples.

Introduction

Nowadays, testing of software is a basic concept of development processes and software lifecycles. Testing is a verification technique that is able to discover defects that otherwise would probably stay hidden. The need for quality and fault-tolerance required to enhance techniques in this area. The current trends in software testing show that test automation is getting more and more important due to the increasing complexity of software and the costs of their development. Automation is spreading in different areas of testing, one of them being test input generation from source code, where several methods and techniques are available [1]. One of the techniques is *symbolic execution* that is a program analysis technique first appeared in the 80s [27] and greatly enhanced in the last decade (e.g. [7, 8]).

Symbolic execution starts from an arbitrary entry point in the source code. During the analysis, the technique interprets each statement while it also gather constraints. These constraints can be passed to a solver in order to get concrete input values, which will execute exactly the path, in which the constraint was gathered. A typical use case of this technique is generating inputs for white-box testing to achieve high code coverage.

Several tools have been presented in the past decade that are able to generate test inputs for white-box tests by using symbolic execution. Microsoft Pex [38] is one of them being, which uses the enhanced variant called dynamic symbolic execution (DSE) that combines concrete executions with the symbolic ones.

Challenges There are many challenges in the industrial adoption and application of symbolic execution (e.g. problem identification of executions, efficient usage on large-scale software) [35, 13, 4, 39]. We experimented previously with Pex in this area and our previous experiences also confirmed these challenges. We generated test inputs for a model checker tool developed at our research group and a content management system from one of our industrial partners. Generating tests in complex software without any guidance for the generator tool may result in low coverage at the first execution. Commonly, this is caused by reaching timeouts, calls to external libraries and resources, or conditions that cannot be solved by the solver. Identification of the root cause of these problems may prove to be effort-intensive and may lead to deep analysis of generated logs. This analysis could be a difficult and time-consuming task for developers and test engineers without academic background. Thus currently, there is a gap between the academic practice of test input generation and its industrial application. To overcome this gap and alleviate the challenges occurred in large-scale software, I applied two techniques.

Contributions First is the idea of *visualizing symbolic execution.* Based on our previous work and related research, I defined a visual representation that may be able to help engineers identifying problems appeared during symbolic execution. The representation is called a symbolic execution tree and I also defined other, related information that should be attached to the tree visually and textually. The nodes provide information about the test generation (e.g. path conditions, source code locations, selected inputs). The visualized tree itself serves as an overview of the execution and may help enhancing the test generation process. I implemented the visualization technique in a prototype tool called SEViz (Symbolic Execution VisualiZer) that is able to collect information from monitoring the executions and is able to interactively visualize the symbolic execution tree with source code inspection. A thesis on this tool was written for the Conference of Association of Scientific Students [22]. A tool paper was also published on the International Conference of Software Testing (ICST) [21], which is one of the largest conferences in research of software testing. The implemented tool is open source and available on GitHub with documentation and demos [1].

The second idea to alleviate the industrial adoption of symbolic execution-based test generation is the *automated isolation environment generation.* Our experiences confirmed that one of the most effort-intensive task during test generation is isolation of the unit under test from external units and libraries. If the source code is not designed to be testable, this task gets more difficult and time-consuming due to the large number of dependencies to handle. The idea uses the parameterized unit test that is the entry point of a test input generation process. During the symbolic execution, the interpreted statements are collected and analyzed by using code analysis dynamically in runtime. From the results of the analysis, two actions are made: 1) the parameters of the unit test are extended, 2) code of the isolation environment is generated that use the extended parameters. I implemented the idea in an extension for Microsoft Pex, which is currently capable of generating isolation code using Microsoft Fakes.

Structure of the thesis The thesis is organized as follows. Chapter 1 presents the important and related notions in software testing to help understanding the rest of my work. Chapter 2 describes the motivation of the work in detail and presents my previous experiences and the related research. In Chapter 3, the visualization technique of symbolic execution and the implementation details of SEViz are presented along with the occurred challenges, which is followed by the evaluation of the tool being introduced with some demonstrational use cases and experimental results. Chapter 4 presents the idea of the automated isolation environment generation in detail and the lessons learned during the implementation, which is followed by the evaluation and experimental results of the technique and the implemented tool. Finally, Chapter 5 concludes my contribution to the addressed challenges and presents the future work.

[1] `http://ftsrg.github.io/seviz`

Chapter 1

Background

Nowadays, the size and complexity of software is growing, thus errors are more easily made during the development phase. This is one of the reasons, why software development cycle models are applying a large variety of testing and verification techniques.

In the first part of this chapter, the basic notions and techniques of software testing are introduced, which is followed by a deeper inspection of unit testing and its role in software quality. The second part of this chapter introduces the symbolic execution-based test input generation technique and Microsoft Pex, a tool that implements a variant of the technique.

1.1 Software testing

This section gives a general overview of the basic definitions, notions and applied techniques of software testing.

1.1.1 Overview

In terms of software anomalies, this thesis applies the IEEE (*Institute of Electrical and Electronics Engineers*) terminology [23], that is the following.

- *Defect:* The software does not meet its requirements and specification.
- *Error:* A human-made mistake during the development.
- *Fault:* Anomaly in a program, which is caused by an error.
- *Failure:* Anomaly in the function of a program, that leads to a failure state.

However, these anomalies cover only a part of the potential problems around software quality. Deficiencies such as inaccurate functionality are also included in this topic.

Notions and definitions in the area of software testing is yet not well standardized, although there are some joint efforts to throughly sum up and standardize concepts, that have connection to software testing. In the list below, two different definitions of software testing are presented. The first one is made by IEEE, while the other one is connected to ISTQB (*International Software Qualification Board*).

- *IEEE:* Testing is an activity, in which the examined system or component is executed, and the results from the execution are analyzed in several aspects [24].
- *ISTQB:* Software testing is a static or dynamic process, that exists in all software development phase, and relates to the design, implementation and evaluation of the software product, which makes it able to decide if the software meets its requirements and goals. Testing is responsible for finding the defects in the software product [26].

Consequently, software testing can be applied in any phase of development. The involved methods and techniques can also be presented in other aspects as in the following section.

1.1.2 Methods of testing

In this section, dynamic testing techniques are presented based on the definition of ISTQB. These methods execute the code.

Test design: There are three main groups of dynamic design techniques, which are the following.

- *Specification-based:* Testing techniques, that are based on the specification, are able to help deciding if the requirements are met. They are commonly called *black-box* techniques due to the fact that the source code itself is seen as a black box.
- *Structure-based:* These techniques use only the source code as their basic knowledge, thus their common name is *white-box* techniques. One of their main motivation is to cover full source code in various aspects (e.g. statements, decision).
- *Experience-based:* These are less formal techniques, and their effectiveness largely depends on the experience and knowledge of the person, who design them.

Observed attributes: The possibly observed attributes divide the techniques into two groups that are well-isolated.

- *Functional attributes:* These describe the behavior and function of the analyzed software. In general, the expected results are given with concrete values.
- *Non-functional attributes:* The characteristics of the functionality are described, such as scalability and robustness. However, in several cases, concrete measurement values can not be given here as expectation.

Level of testing: The currently wide-spread software development processes use testing in almost every phase, thus techniques should also be grouped by the level of testing. Models like the V-Model are good examples, because each step has its own verification step, that verifies the assembled product. In this thesis, unit testing and its related techniques play a main role. Unit test is the verification process of the detailed design phase in the V-Model.

1.2 Unit testing

According to the ISTQB, a unit is the smallest, yet testable part of a program. Consequently, unit testing is an activity to verify the behavior of a unit in a program.

In several cases, it can be hard to find the differences among unit testing and integration testing. The latter one tries to verify the collaboration between units, however defining the units with high granularity may obscure the borders between the two. Thus, it is necessary to accurately define the units and their scope.

1.2.1 Isolated unit

A well-defined unit of a program should have its well-isolated external dependencies, which reduces or even prevents the influence of errors from external components. The next formalized definitions define the isolated units of an object-oriented program.

Definition 1. Let use the following:

- Let $C \in \mathcal{C}_\mathcal{P}$ be a class, $\mathcal{C}_\mathcal{P}$ and $\mathcal{P}$ is a set of classes found in the source code.
- Let $M_C \in \mathcal{M}_C$ be a method, where $\mathcal{M}_C$ is the methods of an arbitrary $C \in \mathcal{C}_\mathcal{P}$ class
- Let $\mathcal{D}_{E,I} \subseteq \mathcal{D}_E$ be a set of dependencies between elements $E, I \in \mathcal{M} \cup \mathcal{C}_\mathcal{P}$ $(E \neq I)$ (class or method) so that $E \to I$, namely E depends on I. A very common example of a type dependency is a usage in the parameters in of the methods. ▪

Definition 2. Let the **unit under test** be a set $\mathcal{U} \subset \mathcal{M} \cup \mathcal{C}_\mathcal{P}$, so that every $U \in \mathcal{U}$ element fulfills for any I: every $D \in \mathcal{D}_{U,I}$ dependency is isolated. ▪

Example 1. *An example system $\mathcal{P}$ consists of classes A, B, $C \in \mathcal{C}_\mathcal{P}$, that are the following:*

- **A:** $\mathcal{M}_A :=$ {m1(),m2()}, $\mathcal{D}_A := \{\emptyset\}$
- **B:** $\mathcal{M}_B :=$ {m3()}, $\mathcal{D}_{B,A} :=$ {m3() $\to$ m2()}
- **C:** $\mathcal{M}_C :=$ {m4(),m5()}, $\mathcal{D}_{C,A} :=$ {m4() $\to$ m2()}, $\mathcal{D}_{C,B} :=$ {m5() $\to$ m3()}

Let there be chosen class C and $m2()$ method of class A as the $\mathcal{U}$ unit under test, thus $U :=$ { C, $m2()$ }. Consequently the only dependency to be isolated is $m5() \to m3()$, as shown on Fig 1.1, where the unit under test is marked with green, while the dependency to be isolated is shown with a red cross.

1.2.2 The importance of unit testing

The importance lies in the well-known fact, that unit testing plays a key role in the quality assurance of software by reducing the number of bugs (e.g. unhandled exceptions or erroneous runs) in a very early stage of development.

The basic principle of unit testing is choosing the smallest unit size as possible. However it is not a trivial decision due to the various complexity of the components of systems, since some components can be very straightforward, but large in size and vice versa. This

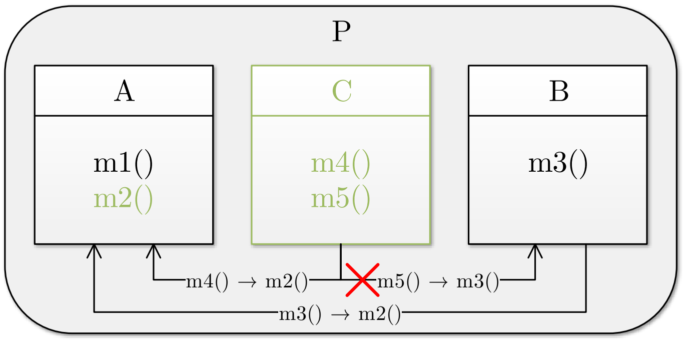


Figure 1.1: Overview of Example 1.

principle ensures more deeper analysis of the unit under test, because basic level bugs (e.g. uncaught exceptions, wrong condition in branching) may remain uncovered with higher granularity levels of units.

In summary, unit testing has several advantages, when it is applied in the correct time and place. These non-exhaustive advantages are the following.

- *Number of discovered anomalies:* Due to the lower level of granularity (e.g. compared to integration testing), an anomalies can be discovered earlier, which may reduce the costs of development.
- *Deeper analysis:* Some external application programming interfaces (APIs) may restrict access to the inner logic of the unit without a maintenance interface. Unit testing is applied within the unit, thus may interact with the logic without using any programming interface.
- *Improvement of maintainability:* Unit testing requires well-defined components and dependencies yet in the early stages of development, thereby it helps the developer to write well-structured, clear source code.
- *Reduction of costs:* Discovering anomalies in earlier stages reduces the costs of their correction. Unit testing is applied in parallel or even before (e.g. in Test Driven Development – TDD) the implementation phase, which can be thought as an early stage in the whole software development process.
- *Possibility of parallelization:* With a well-built structure of the source code, unit testing may be applied in parallel in each of the isolated components. This can reduce the time and thereby the costs of quality assurance.

1.3 Symbolic execution

Symbolic execution is a program analysis technique, that uses symbolic variables as the inputs of the program instead of concrete values. During this process, the program itself is not executed, but a symbolic interpreter interprets each statement, then evaluates the effect of the statement of the symbolic variables. This process goes until every feasible path is analyzed or a predefined boundary is reached. It is important to emphasize that in general case, the feasibility of a path is undecidable in general. [18].

The interpreter forms constraints over the symbolic variables, that are commonly called path conditions (PC). They indicate the feasibility of each discovered and interpreted path and its statements. Example 2. shows the main steps of the whole symbolic execution technique.

Example 2. *Let Foo be a method with an integer return value and two input parameters. The method returns a value based on the two inputs as shown below:*

```
int Foo(bool a, int b) { return a ? (b > 0? 1 : 0) : -1; }
```

The logic of the code itself is two branches nested into each other. The symbolic interpreter firstly substitutes variables a *and* b *with symbolic variables $\tilde{a}$, $\tilde{b}$. Then, the interpreter starts the analysis from statement to statement and if it discovers a branching, it decomposes the symbolic execution for each branch. In this concrete case, the steps are the followings:*

1. *Discovering the first branch, where a condition for variable* a *is present. Here, the symbolic execution goes into two different execution with the conditions $\neg\tilde{a}$ and $\tilde{a}$.*
2. *The arbitrary chosen branch to be executed is now $\neg\tilde{a}$. The interpreter finds the return statements, which indicates the end of the path (because we have no external methods here). Thereby the newly discovered path has the condition $\neg\tilde{a}$.*
3. *The interpreter chooses the next branch to be executed, which is the only remaining: $\tilde{a}$. Here, a new branching is discovered, that also divides the execution into to branches: $\tilde{b} \leq 0$ and $\tilde{b} > 0$.*
4. *Here, another branch is selected to be executed symbolically, however in this special example, both of them terminates with a return statement. Thus, their path conditions are the following: $\tilde{a} \wedge \tilde{b} > 0$, $\tilde{a} \wedge \tilde{b} \leq 0$.*

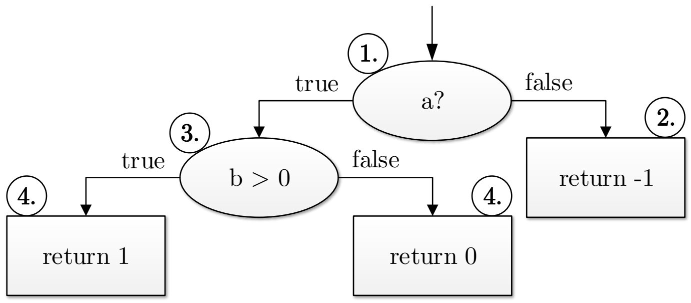


Figure 1.2: A possible visual representation of the symbolic execution.

These path conditions can not be used directly for creating tests, however with the application of a constraint solver the test generation may be carried out. Constraint solvers are able to solve these conditions and they provide concrete values to fulfill them. These values are used as test inputs, which can be extended to become test cases.

Several logical formulas are described by SAT (Boolean SATisfiability Problem), though their solution use only true or false values. Symbolic execution however collects constraints, that are interpreted over variables, thus they need to have values from the value set of its type. This leads to a more general problem called SMT (Satisfiability Modulo Theories). An SMT problem is an extended form of boolean satisfiability where background theories are used to define problems encapsulated into a SAT.

The rapid improvements of computational capacity in the past decade and the development of novel algorithms enabled to solve practical SMT problems, like path conditions collected during symbolic execution. This improvement affected the research around symbolic execution, which led to e.g. an effective collaboration between an SMT solver and the symbolic execution interpreter. This architecture is able to automatically generate test inputs based only on the source code.

1.4 An automated test input generator

Due to the rapid improvement described in the previous section, more and more tools using symbolic execution. Nowadays, several promising tools (e.g. [38, 17, 5, 34]) are under continuous development. The goals of these tools are common: getting into the phase of industrial applicability. One of the most promising tools is Pex developed by Microsoft Research. Currently, Pex is included in the release candidate version of Visual Studio 2015 with the name *IntelliTest*.

Pex is an automated white-box test generation tool for .NET. The tool uses dynamic symbolic execution (DSE, see 1.4.2), an enhanced version of symbolic execution, to collect the path constraints. The input values are passed to parameterized unit tests (see 1.4.1.), that are able to derive concrete test cases with extended implementation. Pex uses the Z3 theorem prover [11] to efficiently solve the path constraints and generate concrete input values from them.

1.4.1 Parameterized unit tests

Parameterized unit tests (PUTs) provides input the symbolic execution engine of Pex. The whole idea is described in [10].

A concrete representation of a PUT is a method with a predefined list of parameters, thus it can be seen as a wrapper interface for the test inputs. PUTs not only contain the calls to the unit under test, but initialization, isolation and assertion logic are also placed here.

Example 3. *The following code snippet shows an example Parameterized Unit Test.*

```
class TestClass
{
  // Parameters corresponding to the method under test
  int PutTestMethod(bool a, int b)
  {
    A testObj = new A();           // Arrange - instantiating the unit under test
    int result = testObj.Method(); // Act - calling the method under test
    Assert.IsTrue(result > 0)      // Assert - assertions on the result
    return result;
  }
}
```

1.4.2 Dynamic symbolic execution

DSE is a technique (also applied by Pex), that combines symbolic execution with concrete execution. Executions run in parallel, which enables the support and guidance of symbolic execution by the information gained from concrete executions.

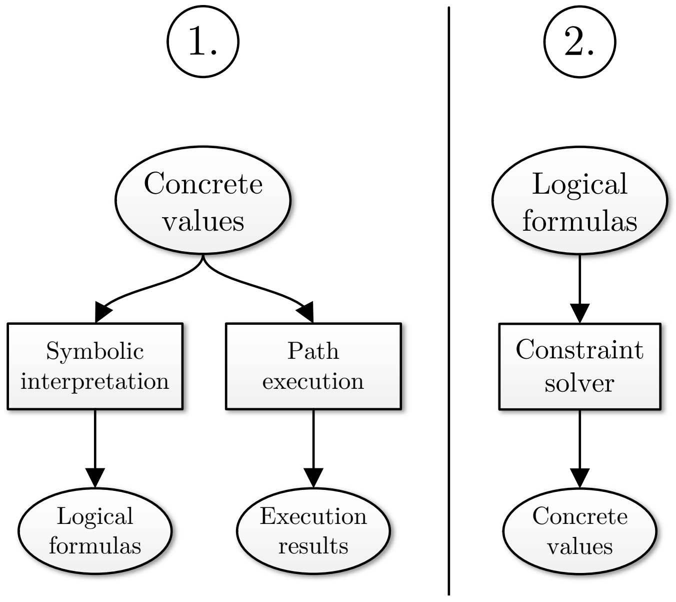


Figure 1.3: A simple overview of the steps applied in DSE.

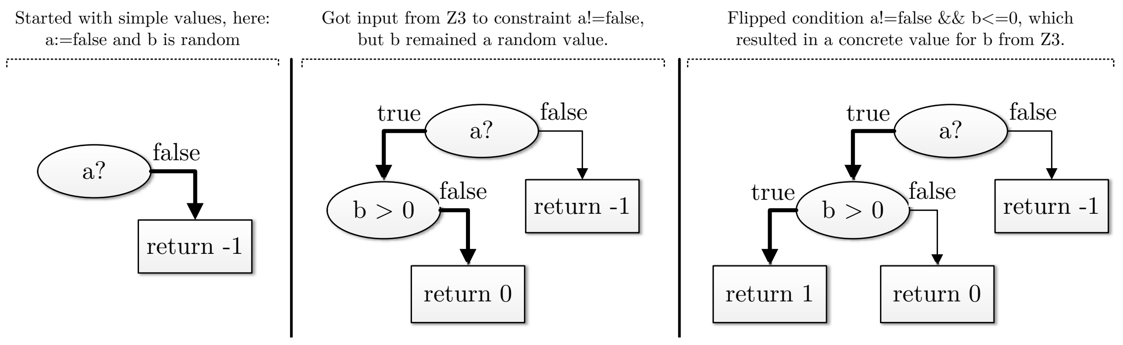
DSE starts from simple, concrete inputs (e.g. zero in case of integers), then collects the constraints along this execution path. In the next step, a search strategy chooses a branching to unfold: the algorithm negates or extends the constraint and passes it to a constraint solver to gain concrete input values. These values provide a new execution path, which is executed next. The algorithm continues this loop until new execution paths are reachable (the new constraint is satisfiable) or the execution reaches a predefined boundary (e.g. time, memory or even the number of discovered branches).

1.4.3 Internal functionality

Pex uses the term *exploration* for the analysis of exactly one PUT. In each step of the exploration, as described in the previous section, the tool starts a concrete execution called *runs*. During the whole exploration process, the tool stores an execution tree, which represents the discovered paths built from code blocks. Code blocks are the smallest part of source code that are handled by Pex, because it uses the intermediate language of .NET during the interpretation. A node in this tree represents a block of program statements or a branching in the source code. Every node is labeled with its path condition called *prefix*, which holds the constraints that have to be fulfilled to reach the node.

The tool selects a node to be unfolded according to a search strategy. When a node is selected, Pex analyzes its prefix and conjugates it with the negated disjunction of the conditions of all discovered outgoing branches. This step is called *flipping*. The newly formed constraint is passed to Z3, which provides concrete values if the SMT formula is satisfiable. It is important to mention, that Pex has an abstract representation for every type in .NET to form SMT problems from the collected path conditions, thereby Z3 is able to process them. Figure 1.4 shows an example (based on Figure 1.2) of the functionality.

In the third step of the example, the search strategy has two nodes to choose from ($a?$ and $b > 0$), however Pex immediately discovers that $a?$ is fully discovered, thus it chooses to the only remaining node ($b > 0$).



Started with simple values, here: a:=false and b is random
a?
false
return -1
Got input from Z3 to constraint a!=false, but b remained a random value.
true
a?
false
b > 0
false
return -1
return 0
Flipped condition a!=false && b<=0, which resulted in a concrete value for b from Z3.
true
a?
false
true
b > 0
false
return -1
return 1
return 0

Figure 1.4: Example of the Pex functionality.

It can be seen that the search strategy could be the key for effective functionality in complex programs. Choosing an inappropriate strategy may easily lead to reaching a boundary of the execution. To overcome this, Pex uses a so called meta-strategy, which changes the used strategy in each step to adapt the environment and avoid reaching boundaries of symbolic execution.

1.4.4 Analysis of extensibility

The default extensibility mechanism of Pex is working with the help of .NET attributes provided by the framework built around the tool. The tool publishes three main abstract attributes that can be implemented. Besides, several other interface exist, however the lack of documentation makes the development very difficult. The three attributes are called by Pex in different points of time, therefore they have different kind of information to process.

- *Execution:* Components that use this attribute are working during the whole time when Pex is running (even when multiple PUTs are analyzed).
- *Exploration:* These components are active, when Pex is making an exploration of a PUT. They are recreated when another PUT is executed.
- *Run:* These components are activated before and after a concrete run of Pex.

Chapter 2

Motivation

In this chapter, the motivation of the work is presented. The first section introduces the motivation and results of related researches, which is followed by our own results based on previous case studies. In the end, the potential improvement areas are discussed, then the research questions close this chapter.

Nowadays source code of industrial software get more and more complex, while the development costs are also high. This principle may lead to quality defects, because testing is one of the most expensive tasks during development, and the only one, which could be skipped. For this reason, today many software lack of testing and have defects, which could be eliminated with testing. As it was mentioned in Section 1.2.2, unit tests may provide at least a partly solution for early detection of severe software defects.

Although unit tests may play a key role in software quality and therefore can reduce costs, the testing process itself could be also expensive and take large amount of time (even more than the 50% of development time) if done exhaustively. Besides this, automated testing in complex, industrial-sized software has several challenges, due to their severe complexity, that must be solved in order to apply testing effectively. This motivated the early researches for the automation of software testing.

2.1 Related work

Bertolino has described the dream of 100% automated testing in [3]. In her paper, she mentioned that this dream would be a fully automatic test environment, in which the source code is deployed and testing would begin immediately without the guidance and supervision of the developer. This testing environment would take care of every related task, like the development of test drivers and mocks too. This dream for unit testing has a promising research track, including general techniques like the XUnit frameworks, the Parameterized Unit Tests and technologies in the field of test input generation. The paper also mentions the idea of "concolic" testing, the combination of symbolic and concrete execution (which was mentioned before with the notion DSE in 1.4.2).

A testing research travelogue from 2014 by Orso and Rothermel [30] summarizes the research around automated test input generation in detail. The paper starts from symbolic execution and mentions DSE as one of the most successful variants of symbolic execution, however the authors state that "practical impact of these techniques are still unclear". Symbolic execution has several heavy limitations to come across, for example in programs with complex structures, large-sized source code and environment interaction. These limitations must be eliminated in order to improve the usability of symbolic execution in industrial software.

Other researches have also found symbolic execution as one of the most promising techniques in the area of test generation [1]. Several tools exist, that implement symbolic execution, therefore some research address the problem of selecting among them [6, 7, 35]. Xiao *et al.* also discussed [35] the task overhead of using symbolic execution in industrial software, in which they mentioned the isolation of dependencies and the pursuit for proper search strategies in individual situations.

In terms of Microsoft Pex, the research is still active. Tillmann *et al.*, the developers of the tool recently analyzed the industrial usage and the transfer from research to practice [39]. The first challenge they met was the problem of early acceptors [1], which needs large efforts from both the developers and the users. The paper also notices that a successful case study in a large-scale software does not mean that the technique is ready for widespread usage, thus it is an important task to bring closer the research and the engineering application of a novel technique, like symbolic execution-based test input generation.

2.2 Our previous case studies

Our previous results in this area have showed the same as the related bibliography, namely the problems during the industrial application of symbolic execution-based test generation. We experienced with two, near industrial-sized software. The first one is a Petri net modeling tool developed at our research group, the other one is a content management system of one of our industrial partners. In this section, the experiences from these case studies are presented.

2.2.1 Petri net modeler

This modeler tool developed at our research group is able to model Petri nets and verify different attributes on them. We examined the second version of this tool, which is currently under development and not available publicly yet. During the test generation process for the modeler tool (ca. 3000 LOC), the following use cases of Pex were applied.

- *Exploration:* The simplest use case of Pex, that allows minimal intervention to the test generation process, thus it can be applied only in a couple of situations.
- *Parameterized Unit Tests:* Gives the ability to fill in the body of the parameterized unit tests, thus concrete test cases can be created and saved for later usage. Furthermore, this use case allows creating the isolated environment around the unit under test, and also allows implementing factory methods for creating complex objects, that are not trivial to instantiate.
- *Extended Parameterized Unit Tests:* By applying this use case, the ability of functional test generation is provided due to the special statements of Pex, which can guide and steer the exploration into predefined directions (e.g. specific inputs for functional testing based on the specification).

One of the most important aspects during the test generation was to create a basic test case set for the developers that they can extend later as the development goes further.

[1]A group in innovation acceptance theory, who comprehensively use the novelty, unlike the innovators.

We managed to generate 371 test cases, which covered 99% of the code blocks. 100 out of them caused exceptions to be thrown, and – based on the feedback of the developers – around half of them were false defects. The other half contained defects that were caused by lack of defensive programming, namely the unprepared code for malicious input values. Moreover, we were able to discover inconsistent modifications throughout the code, which were unreachable during any concrete execution.

We created ten test cases manually for functional testing based on the specification of the modeler. Then, we wrote an extended PUT to cover these manual tests. Pex was able to generate 9 out of the 10 cases, which was a great motivation for further research in the area of applying test generation in industrial-scale software.

However, it is important to emphasize that the first steps of the test generation process took a considerable amount of time due to the complex dependencies among the components, which had to be isolated. We also invested huge efforts to find the reason for the lack of test in some parts of the system. Pex did not provide proper information about the failing of the test generation.

2.2.2 Content management system

One of our industrial partners provided the possibility to examine their software. We selected an application, which is a server-side part of a huge content management system. This software component (ca. 10000 LOC) served as the second case study in our research. We gathered very useful and valuable experiences during the analysis and testing process with the usage of Pex.

Since the source code was unknown for us, the preparation steps for the test generation took more time and effort. Moreover, it needed far more time, than we expected based on the previous case study. We divided the analysis into two main phases.

1. *Analysis without size restriction:* We ran Pex without any isolation environment, thus this was not a real unit testing.
2. *Analysis with small-sized units:* We defined units with only a few classes per unit, which makes the whole testing problem easier and also alleviates the test generation process due to the restricted number of test inputs.

The goal of the first analysis was to unfold the weaknesses and boundaries of Pex in such complex and large-sized software. In the first few runs of Pex, only a little number of test cases were generated, that provided very low statement coverage. A severe hindering factor was that the explorations of Pex was practically chaotic and could not be grasped by the human eye, because each run could run through even 10 method calls. As it was in the previous case study, in this case it also required huge efforts to precisely identify the reasons of reaching the boundaries of the symbolic execution-based test generation.

Despite we have managed to get relatively clear explorations during the second phase, it was also not trivial to identify why Pex failed generating tests sometimes. The discovered defects in the examined methods included a functional bug between the data access and the business logic layer, that resulted in an exception. Furthermore, several defects were discovered around parameter handling and a potential security leak in the system was also recognized.

In the second phase, we had to take the same effort and time, as in the previous case study, to isolate the unit from the environment by creating mocks. The identification of all exiting method calls from the unit is a very effort-intensive and time-consuming task, thus it could be a potential area of automation.

2.3 Potential improvement areas

The previous two sections summarized the research and their potential problems to be solved for symbolic execution-based test generation.

Firstly, the related researches have found out that *complex data structures* and *large-scale source* code may hinder the effectiveness and usability of the technique. Researches has emphasized the *problem of environment interaction* and *isolation* as a closely related task. The former hinders the execution and may lead to reaching boundaries, while the latter is a time-consuming task. Some researches mention the fine-tuning of the symbolic execution as a problematic task, because it depends on the system under test and should be reviewed in every new environment (e.g. the selection of the search strategy). The problem of the learning curve during the industrial adoption was also mentioned in these researches.

Secondly, our case studies showed two hindering factors, that certainly negatively affects the test generation process. The first one was the precise problem identification of reaching boundaries during the symbolic execution, since there was no useful information about the problems and no usable guidelines how to solve them. The other problem was caused by the interaction with the environment and the effort-intensive task of building the mock environment around the unit under test.

It can be seen, that our findings of the hindering factors and the problems mentioned by related researches greatly overlap. The problem identification task and the complex data structure and large-scale source code problem derives from a common deficiency, which is the *lack of perspicuity* of the symbolic execution itself. The other common problem was the *environment interaction and mock implementation.*

2.4 Research questions

The related researches and our earlier results also clearly show that symbolic execution and its related techniques are getting nowadays to the adoption phase in industrial environments. If the common problems mentioned in the previous section would have a solution or would be alleviated at least, the symbolic execution could be more easily used in the industrial practice and large-scale software. Currently, one of the most important task in the area of symbolic execution-based test generation is to find the balance between industrial usage and the effectiveness, in order to solve the early adoption problem.

Based on these, the answers and solutions of the following main- and subquestions are presented in the remaining chapters of the thesis.

How can the industrial application of symbolic execution-based test generation be supported?

1. What type of representation can help reduce the lack of perspicuity?
2. What kind of information can be obtained during symbolic execution?
3. What type of information can be represented for the explorations?

4. How the problem of isolation can be alleviated or solved in order to support test generation?

   To answer the questions above, I propose the following extensions of Pex.

   - Visual representation of the exploration process to help the lack of perspicuity.
   - More precise problem identification for the test engineers in order to diagnose the reason of execution boundaries.
   - Representation of the path conditions to identify the unsolvable SMT problems.
   - Automated isolation, that can automatically generate isolation environment for the unit under test in order to spare time and effort during testing.

Chapter 3

Visualization of symbolic execution

In this chapter, a method is presented, that is able to support symbolic execution during industrial usage.

3.1 Methodology

I have created a visualization technique extended with several metainformation, that may be able to alleviate the problems, which occur during the industrial application of symbolic execution-based test generation. Furthermore, the method serves as a basis of the automated isolation technique (see Chapter 4).

Visualization may be a proper procedure to help test engineers, who have less knowledge about applying symbolic execution in practice. However, this requires such information attached, that is able to increase the perspicuity. Figure 3.1 introduces the basic notions of the technique, where my contribution is filled with orange color.

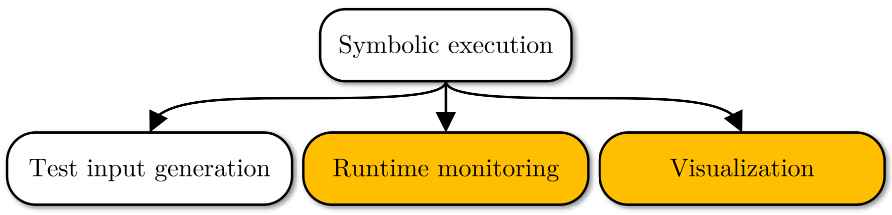


Figure 3.1: Basic notions of the approach.

The working of the technique is the following. First, it monitors the symbolic execution during runtime and collects all the related information. Then, this collected data can be visualized in a clear form which helps engineers to have a general overview of the execution and to identify defects of the execution. Thus, the technique also supports the decision process of further steps during testing.

The following sections presents the background of the methodology from the representation to the concrete visualization.

3.2 Representation

The required information for visualization must be collected during the symbolic execution in a usable representation. This section presents its details.

3.2.1 Nodes and edges

Branches and branchings were already mentioned when symbolic execution was introduces (1.3). These two notions helped to have an idea for the representation of the execution. I chose a tree-structured graph as it is well-known by engineers and ensures a clear overview. However, from the description of symbolic execution, the meaning of nodes and edges may be not obvious. The following two definitions clear the ambiguities.

Definition 3. Let the **symbolic execution graph** be a $G = (V, E)$ directed tree[1], so that it represent a set of run from an analysis of an arbitrary unit $\mathcal{U}$ of any program $\mathcal{P}$.

- The set of vertices V may represent the followings.
  - *Branching:* Any branching in the source code should be a branching in the tree, e.g. a simple `if-else`, or even the branching in the condition analysis of a `for` loop.
  - *Start of a code block:* Every start of a source code block should be represented with a vertex in the tree to vindicate the executions.
  - *End of a code block:* Similarly, every finish of a source code block should be a vertex.
- An edge exists between two arbitrary $v_1, v_2 \in V$ vertex in G, if they were consecutive steps in a run during the symbolic execution. This ensures the correct shape of the tree, which clearly represents the characteristics of execution. ▪

However, the definition does not restrict, that every source code element listed above should be represented in the symbolic execution tree. Thus, for example a method call may not be shown as an individual node in the tree, although the first branching in that method is represented.

The nodes are collected and stored in parallel with the symbolic execution, and done as follows.

1. Discovery of a new node
2. Attachment of a unique identifier and its depth
3. Determining the path condition
4. Assignment of source code mapping
5. Storage with attached data by their unique identifier

Clear visualization moreover requires the detailed information about each run, which is able to increase perspicuity. The following data should be collected during runtime in order to present the required information.

[1]A graph, that has only one path between each pair of its vertices, thus it is acyclic and connected.

- Ordinal number of the run
- Identifiers of the nodes that are visited during a run
- Reason of the run termination (e.g. passed, erroneous, timeout).

3.2.2 Appearance of the nodes

The attached information of each node is required for the simple usage and application in industry. The next sections present two main attributes, but before that, the following definitions give clear description of the visualized attributes.

- The shape of node depends on calls to the constraint solver. If there was a call during the execution of the code block or the branching, the node is ellipse shaped, otherwise it is rectangular.
- If the node has concrete source code mapping, the node has doubled border, otherwise it has single.
- The default background of nodes is white.
- A node has orange background if it is found at the end of a run, but no concrete test case was derived from it.
- A node has green background if it is found at the end of a run and there is a passed test derived from it.
- A node has red background if it is found at the end of a run and there is a failed test derived from it.

3.2.3 Source code mapping

The mapping between the source code and the symbolic execution tree is required to help having a clear overview of the execution. The two-way mapping can be defined in two levels.

- *Line-level:* A node has concrete source lines attached, which has the advantage of precision and accuracy. The disadvantage is that, in cases of intermediate languages[2] it is difficult or even not even possible to implement it.
- *Block-level:* A node in the execution tree has only a block of code attached. The advantage is the simplicity of the implementation, however the representation of the execution is more abstract, than with line-level mapping.

In general, the source mapping should be implemented as precisely as possible in order to increase perspicuity of the symbolic execution tree. Furthermore, every node should have the fully qualified name of their method attached.

[2]E.g. Java bytecode or the IL of .NET.

3.2.4 Path conditions

Definition 4. Let $G = (V, E)$ be a symbolic execution graph, where a path condition in a $v \in V$ vertex is represented by constraints that are collected during the execution. The conjunction in a node $v \in V$ can be divided into a set of individual formulas PC_v. ▪

Taking a look back to Example 2, the path conditions of the two leaf nodes in the fourth step are $\tilde{\mathsf{a}} \wedge \tilde{\mathsf{b}} > 0$ and $\tilde{\mathsf{a}} \wedge \tilde{\mathsf{b}} \leq 0$.

The path conditions should be mapped to each node in the symbolic execution tree in order to trace the executions. The formulas are written in first-order logic, however the operators should be replaced with operators known from programming languages in order to bring closer the visualization and the source code. It is also important to identify the variables in path conditions with the same name as in the source code.

However, deep traversal of source code may build huge path conditions that is hard to interpret for humans. One of the goals was to help the work of test engineers, thus in order to alleviate this hindering factor, I proposed the following solution.

Definition 5. Let $G = (V, E)$ be a symbolic execution graph. The path condition of a vertex $v \in V$ can be built as follows.

- *Full mapping:* Every node in the graph has its own, full PC_v set of individual formulas.
- *Incremental mapping:* In every $v_1, v_2 \in V$ pair of nodes, where v_2 is the parent of v_1, v_2 has only a $PC_{v_2} \setminus PC_{v_1}$ subset of path conditions attached. This set can be thought as a ΔPC compared to its parent node. ▪

Example 4. *The symbolic execution of the source code below, should produce a symbolic execution tree very similar to the tree depicted in Figure 3.2.*

Consider the following object-oriented code snippet.

```
class A
{
  // method Foo
  int Foo(bool a, int b)
  {
    if(a) return Bar(b);
    else return -1;
  }

  // method Bar
  int Bar(bool b)
  {
    if(b) return 1;
    else return 0;
  }
}
```

It can be seen, that the path conditions are mapped to the nodes both in full and incremental form. The unique identifier of the nodes are their traverse order numbers. In the figure, PC denotes the full form, while ΔPC is the incremental form. The figure also shows the name of the methods.

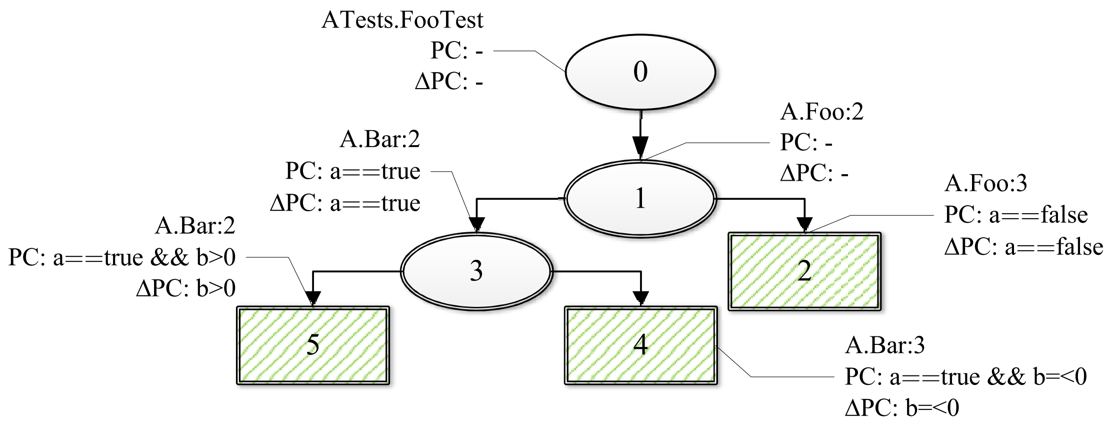


Figure 3.2: The symbolic execution tree extended with path conditions.

3.2.5 Indicating unit isolation

During sections 1.2.2 and 1.2.1 the importance of isolated unit testing was introduced. This isolation can also be used with symbolic execution in order to reduce the complexity of the unit under test, however it is a rather complex task.

The visualization of symbolic execution can also help test engineers determining the borders of the unit to be isolated. To support this, the symbolic execution can be extended as follows.

Definition 6. Let there be a symbolic execution tree $G = (V, E)$ derived from the symbolic execution of an arbitrary unit under test U. Every node $v_1, v_2 \in V$ for the following holds:

- a $v_1 \to v_2$ directed edge exists,
- v_1 is derived from a method found in U,
- v_2 is derived from a method outside U,

then the $v_1 \to v_2$ edge should be highlighted in the visualization, thus it can indicate an exit point from the unit. ▪

Using this technique, the test engineer can identify the method to be mocked. The following example introduces the function of the unit border marking.

Example 5. *Consider the code snippet in Example 4. Let the $\mathcal{U}$ unit under test consist only of method* `Foo`*, formally* $\mathcal{U} := \{$`Foo(int,int)`$\}$ *and* $\mathcal{D}_{\mathcal{U}} := \{$`Bar(bool)`$\}$*. Thus, only method* `Bar` *is an external dependency.*

Then a symbolic execution started from a PUT, where method `Foo` *is called, the following symbolic execution tree can be obtained. Figure 3.3 clearly shows the red directed edge, which is a call to an external dependency.*

3.3 Implementation

This section introduces the problems appeared and their solutions during the implementation of the visualization technique.

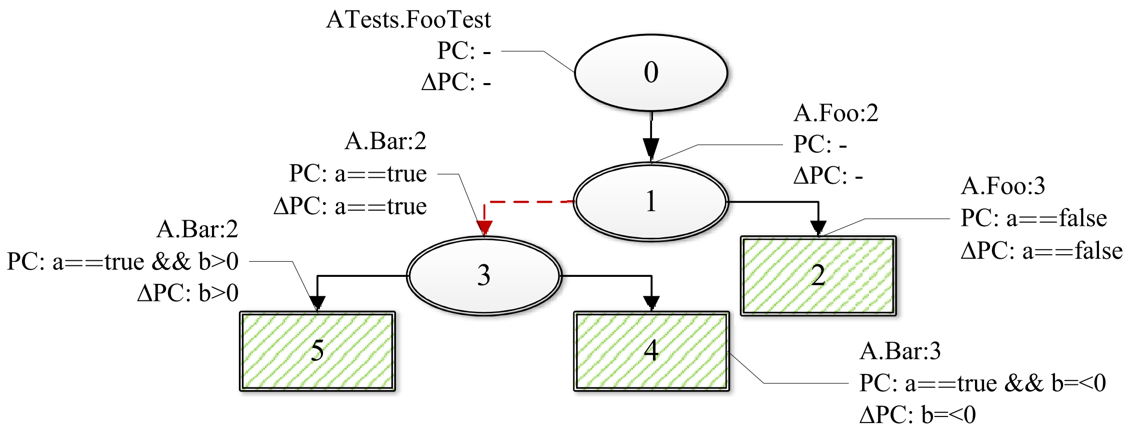


Figure 3.3: An example of marking the exit points from the unit in a symbolic execution tree.

3.3.1 Tool architecture

I created a tool, called SEViz (Symbolic Execution VIsualiZation), that currently works with data collected from Pex, however the intention was to design a tool, which could be used with a wide range of symbolic execution-based tools. Thus,vI separated the architecture into three loosely-coupled components. Each of them has well-defined tasks to support maintainability. Fig. 3.4 shows an overview of the structure of the tool.

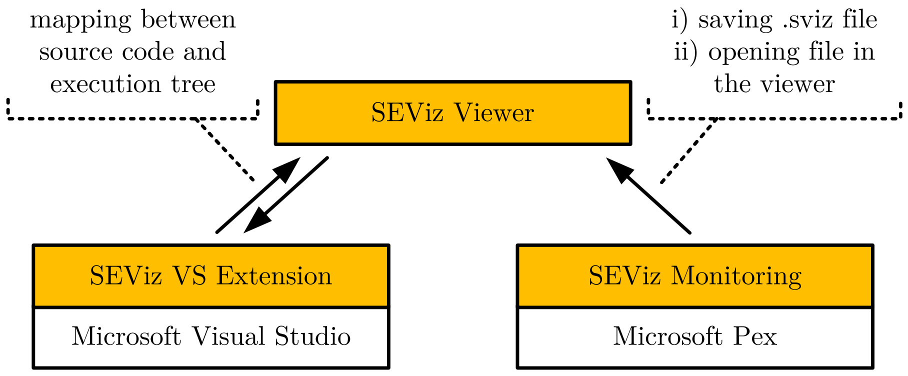


Figure 3.4: Architecture of the SEViz tool

3.3.2 SEViz Monitoring

The component collects the information from the executions of Pex through the provided Pex extension API. Several extension points are present in Pex, thus I had to decide, that which one affects the executions the least. I gathered ideas for the implementation from existing, open-source Pex extensions (e.g. [9, 42, 43]).

Pex uses search frontiers during the symbolic execution to choose which code location (execution node) is going to be explored next. The tool allows the definition of search frontiers in order to alter the strategy of the code exploration. A monitoring search frontier, based on the default one, would have been a straightforward solution. However, during the implementation of the monitoring search frontier, I discovered that Pex only attaches the metadata to the nodes after the executions, which cannot be reached through this API. Thus, I searched for another solution.

The architecture type of Pex offers extensibility in each of its layer (which is execution, exploration and path). The path layer defines callbacks before and after each execution path. Firstly, I analyzed these path-based operations with test extensions to determine if the layer fits the requirements. After an execution of a path, Pex provides the list of the executed nodes with all the required metadata attached. Nevertheless, other information is needed, which can only be extracted after the whole exploration (e.g. generated tests, Z3 [11] calls).

Algorithm 1 Collecting the nodes and edges.

```
1: for each node in nodesInPath.ExcludeFirst() do            ▷ iterating without root
2:     prevNode := nodesInPath[node.Index -1];                   ▷ getting the previous
3:     if !prevNode.KnownSuccessors.Contains(node) then
4:         PrintEdge();                                            ▷ printing the edge
5:         prevNode.KnownSuccessors.Add(node);                   ▷ now we know about it
6:     end if
7:     nodeStorage.Add(node);                                      ▷ storing the node
8: end for
```

This monitoring component stores the execution nodes in an internal storage (Algorithm 1), which plugs itself into Pex. After the exploration ends, SEViz flushes the collected nodes into a GraphViz [15] file and all of its related metadata into data files. Then, SEViz calls GraphViz to compile its source to create a layout for the graph. Finally, the generated files are zipped into a SEViz (.sviz) file, which can be opened with the SEViz Viewer.

Constraint solver calls The implementation of the monitoring component however had several challenges to solve. Firstly, the calls to the constraint solver have to be tracked in order to show them on the symbolic execution tree. After a deep search in the API of Pex, I discovered an event, which is fired, when a logical formula is passed for solution, thus I used this event by attaching an event handler to indicate the calls.

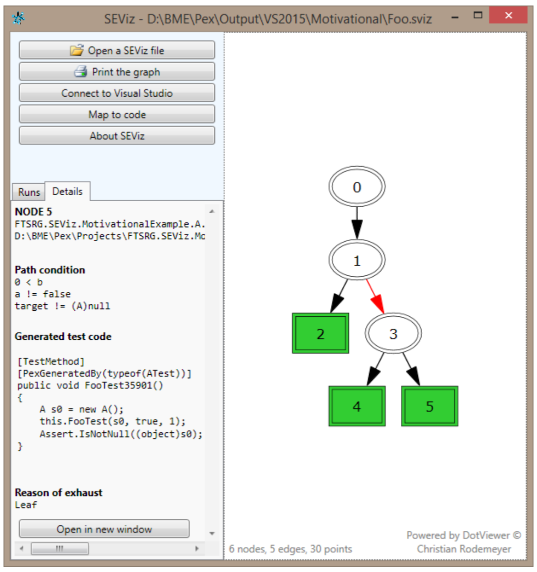
Source code mapping The implementation of the source code mapping was another huge challenge. In the first step, it should be verified that a node in the tree has information about its place in the code, since external libraries lack this information. Next, it should be checked that the source code file exists in the file system that was retrieved before. The mapping can be carried out only if this file is found. The functionality of the mapping is shown in Algorithm 2. Method `MapToFile` and `MapToLine` was implemented by us. The former maps to the file in the system, while the latter maps .NET IL to source code lines using the so called sequence points.

Path conditions Each node has its path condition attached, which is one of the most valuable information for a test engineer for problem identification. Pex uses its own representation to store the formulas, thus when the Monitoring component extracts them it needs to present it in a clear, readable form. This transformation is called *pretty printing*, and I implemented the technique using the services provided by Pex, which were discovered with reverse engineering tools.

Algorithm 2 Algorithm of source code mapping.

```
 1: function MapToSourceCode(ExecutionNode node)
 2:     if node.Location then                                   ▷ if source info exists
 3:         file := MapToFile(node.Location);                    ▷ mapping to file
 4:         if Exists(file) then                                ▷ if file exists
 5:             line := MapToLine(node.Location.Offset);         ▷ mapping to lines
 6:             return file + line;                        ▷ return with file and lines
 7:         end if
 8:     end if
 9:     return null;                                   ▷ otherwise return nothing
10: end function
```

Figure 3.5: Symbolic execution trees in SEViz


SEViz - D:\BME\Pex\Output\VS2015\Motivational\Foo.sviz
Open a SEViz file
Print the graph
Connect to Visual Studio
Map to code
About SEViz
Runs
Details
0
1
2
3
4
5
Open in new window
6 nodes, 5 edges, 30 points
Powered by DotViewer © Christian Rodemeyer

NODE 5

```
FTSRG.SEViz.MotivationalExample.A.
D:\BME\Pex\Projects\FTSRG.SEViz.Mc
```

Path condition

```
0 < b
a != false
target != (A)null
```

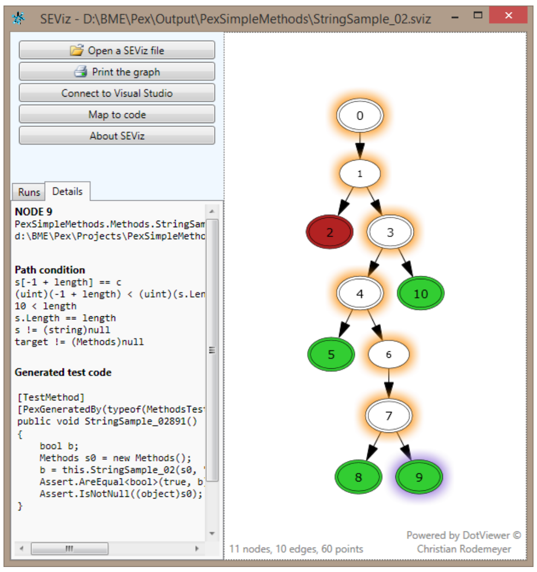
Generated test code

```
[TestMethod]
[PexGeneratedBy(typeof(ATest))]
public void FooTest35901()
{
    A s0 = new A();
    this.FooTest(s0, true, 1);
    Assert.IsNotNull((object)s0);
}
```

Reason of exhaust

```
Leaf
```

(a) Visualization example


SEViz - D:\BME\Pex\Output\PexSimpleMethods\StringSample_02.sviz
Open a SEViz file
Print the graph
Connect to Visual Studio
Map to code
About SEViz
Runs
Details
0
1
2
3
4
10
5
6
7
8
9
11 nodes, 10 edges, 60 points
Powered by DotViewer © Christian Rodemeyer

NODE 9

```
PexSimpleMethods.Methods.StringSam
d:\BME\Pex\Projects\PexSimpleMethc
```

Path condition

```
s[-1 + length] == c
(uint)(-1 + length) < (uint)(s.Len
10 < length
s.Length == length
s != (string)null
target != (Methods)null
```

Generated test code

```
[TestMethod]
[PexGeneratedBy(typeof(MethodsTes
public void StringSample_02891()
{
    bool b;
    Methods s0 = new Methods();
    b = this.StringSample_02(s0, '
    Assert.AreEqual<bool>(true, b
    Assert.IsNotNull((object)s0);
}
```

(b) String example

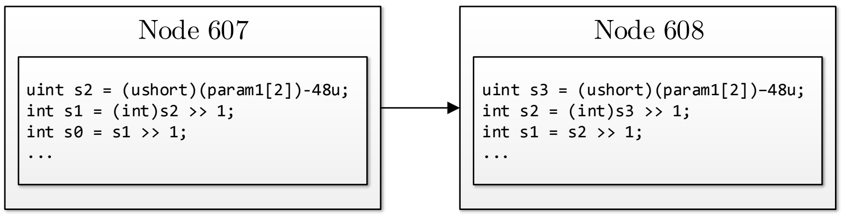
SEViz Monitoring is compatible with the public version of Microsoft Pex (v0.94.51023.0), and also successfully collaborates with the recently released IntelliTest integrated in Visual Studio 2015 RC.

3.3.3 SEViz Viewer

The main component of the architecture is based on GraphViz files. I used an open-source project (DotViewer [36]) to visualize the compiled graph data. The Viewer opens SEViz (.sviz) files, extracts its content, and builds up the graph. The DotViewer WPF control enables to zoom in and out, and also allows to select one or multiple nodes. The main features besides the visualization are the following. Figure 3.5 shows two examples opened in the Viewer.

- *Zoom and scroll:* Users are able to zoom into large graphs to search for details, and it is also allowed to scroll in two dimensions.
- *Node selection:* Users can select one or multiple nodes for analysis. These are highlighted in the visualized graph.

- *Node details:* Detailed metadata of the nodes are presented in two forms. 1) When the cursor hovers a node, then a tooltip appears with the incremental path condition, the method of the node and the code location mapping (if it exists). 2) When a node is selected, all of its metadata is presented in the node details section of the UI, which can be opened in a new window too. This enables users to compare the data of different nodes.
- *Run selection:* SEVIZ enables users to select a run from a list and the tool highlights the corresponding nodes in the visualized graph.



```
uint s2 = (ushort)(param1[2])-48u;
int s1 = (int)s2 >> 1;
int s0 = s1 >> 1;
...
```

```
uint s3 = (ushort)(param1[2])-48u;
int s2 = (int)s3 >> 1;
int s1 = s2 >> 1;
...
```

Figure 3.6: Example of the incremental path condition problem.

Incremental path conditions The Viewer component is responsible for dividing the full path conditions into a clearer, incremental form. The implementation however had some difficulties to come across. When Pex explores complex programs and structures, the tool introduces variables in the path conditions, which makes it more difficult to read. When this happens, the path condition of a node consists of two parts: 1) initialization, 2) conjuction. The former includes the declaration and initialization of logical variables, while the latter contains the conjucted set of conditions. The main difficulty is that Pex names these variables in a non-deterministic way[3] for each node, thus the conditions in a children-parent pair of nodes can be mapped into each other, however the textual representations do not match. Figure 3.6 introduces an example of this problem. There are two possibilities to solve this.

1. Comparing the two logical formulas using a constraint solver, which can results in the difference if the constraints are well-composed.
2. Comparing the textual representation of the formulas using text-based pattern matching, which can resolve the variable name problem.

The first alternative may seem the obvious one, however it must be emphasized, that the path conditions obtained from Pex must be transformed back to the original representation or into SMT-LIB [2] to pass them for Z3. This would raise multiple, complex problems and may reduce the effectiveness of Pex due to the increased computational need.

Thus, I applied the second alternative to implement the incremental path conditions. The algorithm of the obtaining technique for the increment is presented in Algorithm 3. The algorithm assumes that number in the variable names have a difference not greater than three in each consecutive pair of nodes. I derived this assumption from a large number of observations.

[3]The variables have *si* form, where *i* is a non-deterministic number.

Algorithm 3 The incremental path condition algorithm

```
function GetIncrementalPathCondition(ExecutionNode node)
    remainedPC[ ];
    nodePC[ ] := SplitByLine(node.PathCondition);              ▷ splitting into lines
    OrderByLetter(nodePC);                                      ▷ ordering the conditions
    prevNode := FindParent(node);                               ▷ search for parent
    prevNodePC[ ] := SplitByLine(prevNode.PathCondition);       ▷ splitting into lines
    OrderByLetter(prevNodePC);                                  ▷ ordering the conditions
    if !HasInitializations(prevNodePC) then                     ▷ no initialization exist
        for each s in NodePC do
            if !prevNodePC.Contains(s) then
                remainedPC.Add(s);              ▷ adding, if previous did not contained it
            end if
        end for
    else
        remainedPC := nodePC;                                   ▷ otherwise processing it
    end if
    for i = 1 to 3 do                                           ▷ assumption for difference
        for each line in prevNodePC do
            incrementedLine := IncrementVariableName(line,i);
            remainedPC.Remove(incrementedLine);                 ▷ removing it if contained
        end for
    end for
end function
```

3.3.4 SEViz Visual Studio Extension

The third component, an extension for Visual Studio, is responsible for ensuring the two-way mapping between the source code to visualized nodes in SEViz Viewer.

The component uses pipes to implement inter-process communication with the Viewer. I created a pipe server in the Viewer and also in the Visual Studio extension for ensuring two-way communication. Thus, when a user selects a source code line the corresponding node is selected in the symbolic execution graph. Furthermore in the other direction, when a user selects a node in the graph, the corresponding source code line is selected in a predefined Visual Studio instance.

The difficulty of the implementation was that Pex uses IL, the intermediate language of .NET, for symbolic execution. To implement this mapping, I relied on the internal services of Pex, which are made available through its API. First, I get the instrumentation info about the method of the currently analyzed node. Then, the offset of the node in this method is extracted, which results in a sequence point. Finally, this can be used in the translation to the source code. However, this multi level mapping does have some potential inaccuracy, which can be optimized in future works.

3.4 Introductory examples

In this section I introduce three examples with their code snippets and symbolic execution trees in order to present a better overview of the technique.

Example 6. *In this example, an array and an index is given to the method under test. The logic of the method is the following: it return true if the array contains the number 99 at the index 1, otherwise it returns false. The source code is the following.*

```
public bool ArraySample_02(int[] array, int index)
{
  var element = array[index];
  if(index == 1)
  {
    if (element == 99)
    {
      return true;
    }
  }
  return false;
}
```

Pex generated five test cases, that are detailed in the table below. SEViz *generated the symbolic execution tree seen on Figure 3.7. It can be seen, that tree has each of the test cases represented including the two failing one. It is also interesting that it was the last run, which covered the* `return true` *statement.*

array	**index**	**Result**	**Summary**
null	0	-	`NullReferenceException`
{}	0	-	`IndexOutOfRangeException`
{0}	0	false	-
{0, 0}	1	false	-
{0, 99}	1	true	-

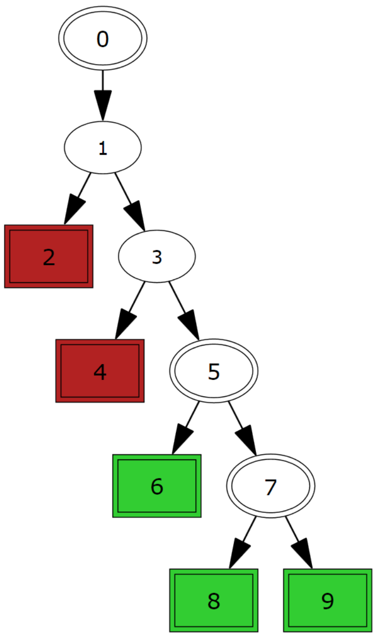


Figure 3.7: Symbolic execution tree of method ArraySample_02.

Example 7. *The example method gets three inputs: a string and two integer values. The functionality of the method is the following. The input string should contain the substring "ThisIsATest" and after from the index given by the first integer, characters are deleted* *from the string, where the number of deleted characters is determined by the second integer input. If this string remains "ThisIs" and the original contained "ThisIsATest", then the method return true, otherwise it returns false. The source code of the method is the following.*

```
public bool StringSample_05(string s, int index, int count)
{
  var temp = s.ToString();
  s = s.Remove(indexToDelete, count);
  if (temp.Contains("ThisIsATest") && s == "ThisIs")
  {
    return true;
  }
  return false;
}
```

Pex generated eight test cases, from where two fail with exception. It can be also seen how Pex tried to increase the coverage by changing the value of the input string. The generated symbolic execution tree can be seen on Figure 3.8.

s	index	count	Result	Summary
null	0	0	-	`NullReferenceException`
""	0	0	false	-
new string('\0', 11)	0	11	false	-
"ThisIsATest"	0	11	false	-
""	1	0	-	`ArgumentOutOfRangeException`
"\0"	0	0	false	-
"\0\0\0"	2	1	false	-
"ThisIsATest"	3	6	true	-

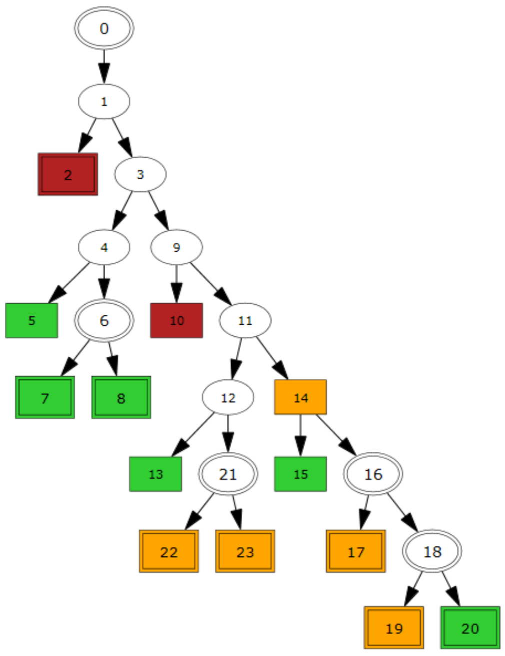


Figure 3.8: Symbolic execution tree of method StringSample_05.

Example 8. *The last example is based on a method, which calculates the Fibonacci numbers by using a loop instead of recursion. These kinds of structures are well-known hindering factors of effective symbolic execution. The implementation of the method is the following.*

```
public int[] ForLoop_Fibonacci(int length)
{
  int a = 0;
  int b = 1;
  int c = 0;
  int[] numbers = new int[length];
  numbers[0] = a;
  numbers[1] = b;
  for (int i = 2; i < length; i++)
  {
    c = a + b;
    numbers[i] = c;
    a = b;
    b = c;
  }
  return numbers;
}
```

Pex generated six test cases, and three of them failed with exception. One of them raised an `OverflowException` that can point to potential defects of the source code, which is not inspected by the engineer in detail. The tree of symbolic execution is able to point to these parts and with the help of the branchings (e.g. finding a missing branch), the test engineer may be able to uncover new cases too. The upper part of the tree is shown on Figure 3.9.

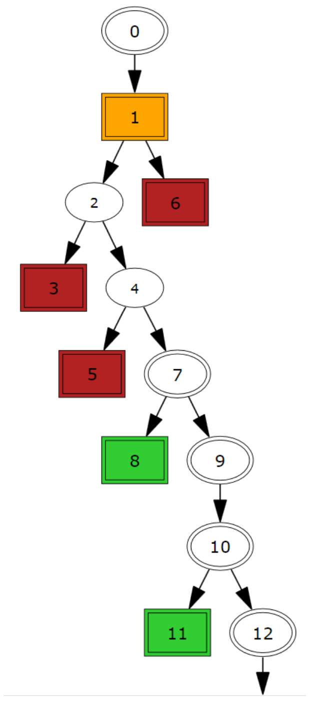


Figure 3.9: Symbolic execution tree of method ForLoop_Fibonacci.

length	Result	Summary
0	-	`IndexOutOfRangeException`
1	-	`IndexOutOfRangeException`
`int.MinValue`	-	`OverflowException`
2	{0, 1}	-
3	{0, 1, 1}	-
7	{0, 1, 1, 2, 3, 5, 8}	-

3.5 Evaluation

I used the tool to generate visualization for complex programs, where test generation was hindered by problems, that are difficult to identify. In this section, some examples are presented for use cases of the tool, which is followed by the experiments that were made to evaluate the effectiveness of the tool. Finally, the limitations of the approach and the tool are summarized.

3.5.1 Example use cases

This section presents the two main use cases of the tool. First, by visualizing the symbolic execution of simple programs with basic language elements (branches, loops, etc.), users can get familiar with the technique. Secondly, it is able help test engineers and developers to make the correct decisions to enhance test generation.

The typical usage workflow of SEViz is rather straightforward. Currently, it collaborates with Microsoft Pex, therefore I introduce the workflow using it.

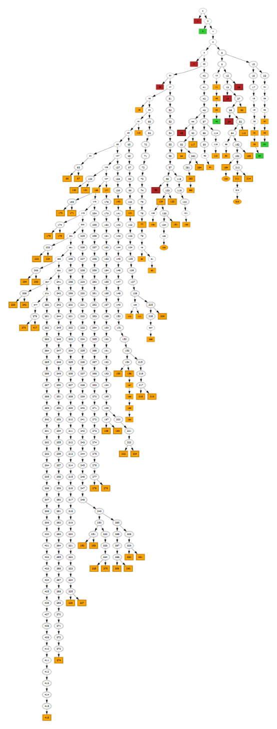
1. *Attributing:* The user should specify two attributes for the unit test method in order to start monitoring during the execution of Pex. These attributes define the output file location and the unit under test.
2. *Execution:* During the execution of Microsoft Pex, the monitoring is active and data is collected.
3. *Analysis:* When the execution finishes, SEViz creates a file, which contains all the collected information. This can be opened with the tool and a symbolic execution tree is drawn from the data.
4. *Mapping:* Each node, which has source code metadata, can be mapped to a source code file and line in a user specified Visual Studio instance.

Educational and Training Usage

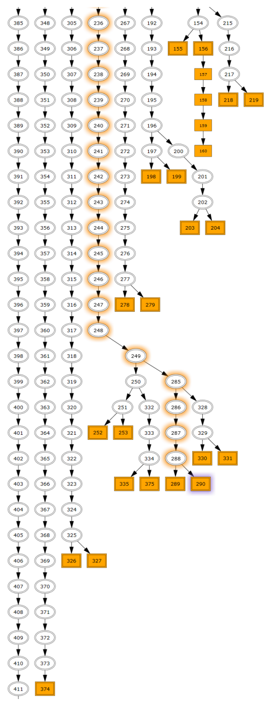
The motivation of this work was to support engineers using symbolic execution for test generation in large-scale, complex software cases. However, SEViz, the implemented tool has another use case, which is to support the learning curve of the usage of symbolic execution. From simple to rather complex cases, the implemented tool is able to visualize the executions, therefore it is able to help understanding the working of symbolic execution and SEViz to support the teaching of symbolic execution based test generation.

Users can easily extract the path conditions and observe the executions as the contained constraints are incremented with literals from node to node (by using the incremental

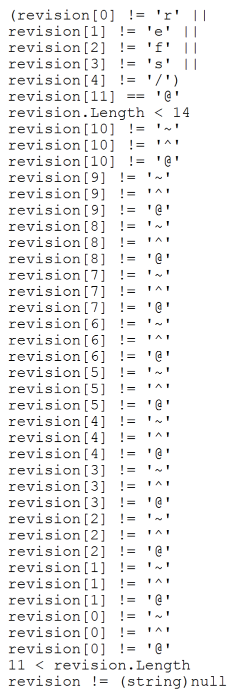
Figure 3.10: Analysis steps of the GitSharp example in SEVIz



(a) Overview of the tree



(b) Loop identification



```
(revision[0] != 'r' ||
revision[1] != 'e' ||
revision[2] != 'f' ||
revision[3] != 's' ||
revision[4] != '/')
revision[11] == '@'
revision.Length < 14
revision[10] != '~'
revision[10] != '^'
revision[10] != '@'
revision[9] != '~'
revision[9] != '^'
revision[9] != '@'
revision[8] != '~'
revision[8] != '^'
revision[8] != '@'
revision[7] != '~'
revision[7] != '^'
revision[7] != '@'
revision[6] != '~'
revision[6] != '^'
revision[6] != '@'
revision[5] != '~'
revision[5] != '^'
revision[5] != '@'
revision[4] != '~'
revision[4] != '^'
revision[4] != '@'
revision[3] != '~'
revision[3] != '^'
revision[3] != '@'
revision[2] != '~'
revision[2] != '^'
revision[2] != '@'
revision[1] != '~'
revision[1] != '^'
revision[1] != '@'
revision[0] != '~'
revision[0] != '^'
revision[0] != '@'
11 < revision.Length
revision != (string)null
```

(c) Path condition of a selected node

path conditions). Especially, cases with complex objects are challenging, since the path condition constructs are can grow into rather complex constraints.

Source code mapping and the overall view of the symbolic execution tree can help understanding and comparing the search strategy used during the execution. For example, as the sequence numbers inside the nodes show the order of execution, the user can experiment with different strategies (e.g. BFS, random, fitness-guided) to understand the traversal of different constructs in the code for example loops or recursion.

Let us see the example source code from the previous section along with the manually drawn symbolic execution tree. Microsoft Pex was run on the code, and SEVIz monitored the execution. Fig. 3.6a shows the symbolic execution tree. SEVIz shows how Pex explored the code. The shape of the nodes illustrate that the constraint solver was called three times (node 0, 1 and 3). Pex selected three test inputs (node 2, 4 and 5). The edge from node 1 to 3 is colored, because that edge corresponds to calling the `Bar` method, and I specified method `Foo` as the unit under test. By selecting the nodes, all the path conditions can be examined in the details panel.

Fig. 3.6b presents a real-world example from a method that manipulates strings. The node with purple selection has its details opened, while the orange nodes show the path, which should be traversed to get there.

Engineering Usage

The implemented tool supports test engineers, who are familiar with test input generation and use it on complex software. Their workflow (generate tests for code, examine generated test inputs and coverage, improve generation) can be supported by visualization.

In general, the visualization of the symbolic execution with the tool can help identifying problems, which prevent achieving higher code coverage. The following list summarizes the most important characteristics to analyze in common situations.

- *Shape of the tree:* It shows the effectiveness of the traversal strategy. Too deep branches could mean that the search strategy was not efficient or a loop or recursive structure was not properly handled.
- *Generated tests:* If there are many leaf nodes with no corresponding generated tests, those can be unnecessary runs. Either they should be cut off to avoid repeat traversal or guidance should be provided to help selecting test more relevant inputs.
- *Path constraints:* Complex constraints blocking the progress of symbolic execution can be identified by analyzing the path conditions.
- *Unit borders:* Those traversals which pass through the boundary of a unit reached code regions, which are not in scope of the testing. Indicating these exit points supports finding proper position for isolation.

Let us introduce a real-world example from the project called GitSharp [12] to show how visualization can be used to increase coverage. With SEViz the issues can be more quickly identify (e.g. reaching boundaries of the execution, lack of coverage) in the generation and the detailed analysis of the reports and logs can be avoided using the following steps. My first steps with this GitSharp example showed that Pex was not able to produce the correct format of a string in a resolver method.

1. *Execution identification (Fig. 3.11a):* By just looking at the tree at a high level, the test inputs can be identified that were only selected in the runs at the right side of the tree, and there is a large, deep portion of the tree where Pex spent considerable time, but was not able to select any tests.
2. *Loop identification (Fig. 3.11b):* In the next step, I took a look at the source code mappings of the nodes in this deep branch. I discovered, that the nodes are mapped to a loop, which causes the tree to be deep. Now it should be investigated why the execution did not quit from the loop.
3. *Path condition identification (Fig. 3.11c):* The answer lies in the path conditions of the nodes. It can be seen, that Pex could not guess the correct condition to exit the loop, it tried to add the same characters until reaching its execution boundary.

By going through these three steps, the problem (format of the string) can be discovered just by working with the data presented by SEViz, and there is no need for further analysis or debugging.

3.5.2 Experiments

The basic conditions of the visualization technique and tool are: 1) execution time should not be greatly affected by the visualization monitoring, 2) the solution should be scalable to larger, complex source codes. This section introduces the quantitative measurement results by following these two guidelines.

Besides efficiency, usability is an important factor in engineer application. However, measurement of usability requires preplanned, step-by-step process with at least ten test engineers involved. Fraser *et al.* elaborated a similar process to measure the usability of their test generator tool [13]. However, during my work, I had no opportunity to conduct a similar process to evaluate the technique and tool.

I executed the measures in the following environment: Intel Core i7 @ 3,1 GHz, 8 GB DDR @ 1333 MHz, Windows 8.1 Pro, .NET v.4.5.51641, Microsoft Visual Studio Ultimate 2013 v12.0.30501 Update 2, Microsoft Pex v0.94.51006.1.

Analysis of execution time

I divided the measurement into two parts: 1) artificial examples, 2) real-world examples. Thus, the tool can be analyzed in two aspects, on one hand the tool can be verified against its minimal requirements (e.g. visualization of basic structures), on the other hand the effectiveness in the intended usage environment of the tool can be verified. I measured the following attributes during the analysis.

- Execution time
- Execution time with visualization
- Number of traversed nodes
- Deepest node
- Lines of code

Pex has a built in watch to measure execution times, however it counts needless times, like generating the log file. In order to have accurate measurements, I implemented a component for Pex, that uses a stopwatch for this functionality.

In this thesis, I only present the results of the real-world examples and the summary of the execution time analysis. The rest of the results can be found in [22].

I chose nine methods from different open-source projects and one method inspired by the CMS system presented in Section 2.2.2. Five out of them are from projects, that have been analyzed with Pex before [41], therefore they also serve as a good basis for evaluation of the visualization [12, 32]. Two of the four remaining methods come from a project, that implements biological algorithms and data structures [33], while the other two is from the .NET version of the Couchbase NoSQL database management system. I selected the methods with variable complexity, however each of them may be a challenge for the tool.

I used the same steps for the two analysis, thus I examined the relation between the ratio of execution times (with/without visualization), the height of the deepest node, the number of nodes and the lines of code. The results of the experiment is summarized in the table below, where T denotes the time without visualization, TV denotes the time with visualization, R is the ratio of the two times, N is the number of nodes and D is the height of the deepest node.

Project	Method	T [s]	TV [s]	R	N	D	LOC
Couchbase	AppendData	11,994	54,082	4,509	676	52	49
Couchbase	Compare	6,102	47,527	7,789	989	32	54
CMS	CreateUser	9,991	135,954	13,61	2104	238	5
Bio	MapAcid	244,678	377,82	1,544	717	149	11
Bio	Compare	2,697	4,033	1,495	65	14	8
DSA	IsPrime	3,114	10,135	3,255	36	16	9
DSA	Avl.Remove	2,491	8,668	3,48	21	13	6
DSA	Heap.Remove	0,529	6,478	12,25	40	13	20
GitSharp	Merge	173,493	1334,22	7,69	7322	1073	62
GitSharp	MyersDiff	37,965	440,4	11,6	3197	488	23

Let us look at Figure 3.11, which presents the connection between the height of the deepest node and the execution times. It can be seen, that several ratio of execution times went over five, which is due to the large number of loops and iterations on data structures. Some executions had the same deepest node, but the times were different. The reason of this is that one of the two had more, similarly deep branches than the other, which slows down the process.

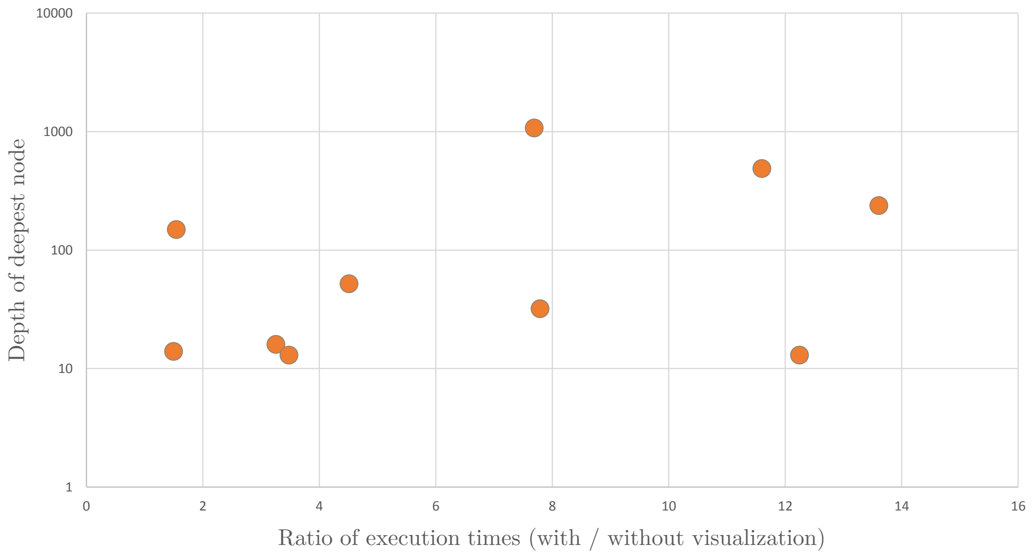


Figure 3.11: Relation between depth and execution time.

Figure 3.12 presents the results for the analysis between number of nodes and execution times. It can be discovered, that while the number of nodes grow from ten to tens of thousands, the execution times may only grow two times larger. Taking the average execution times into account, these scale of growth can be acceptable in industrial environments.

I also experimented with the connection between the lines of code and the execution time, however the results showed no correlation, thus I do not discuss it here.

Based on the results, I can state the the visualization greatly affect the execution time, however there are two reasons I must mention: 1) long loops, 2) hash functions. Both of them are the consequences of the symbolic execution approach, thus I do not discuss these in detail.

In summary, the results of real-world examples did not contain extremely high increase of execution time, because there were no unbounded loops or recursions. Moreover, the

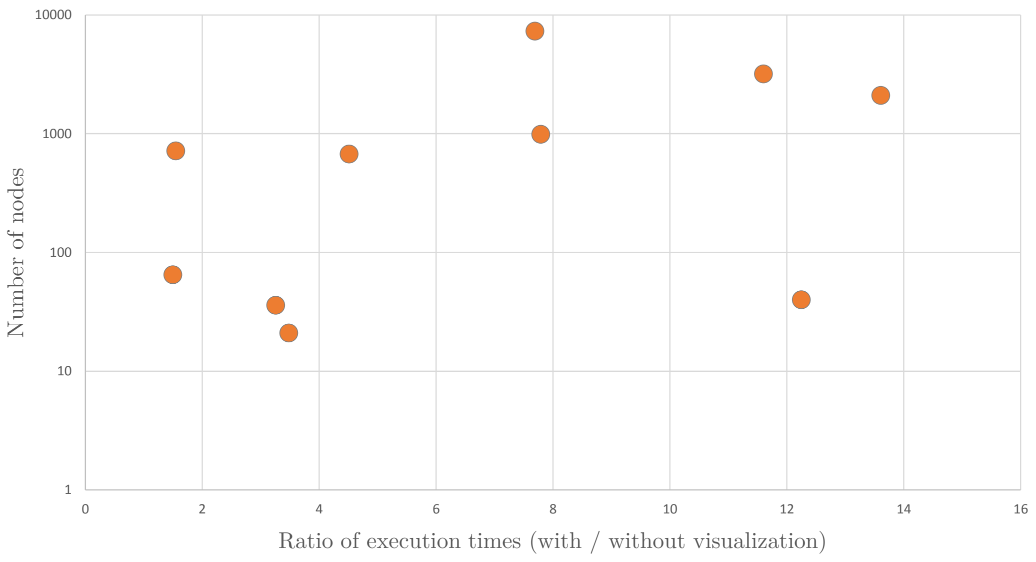


Figure 3.12: Relation between number of nodes and execution time.

experienced growth of execution time may be acceptable in engineering usage compared to their average length (can be measured in seconds or some minutes).

Analysis of scalability

I implemented a parameterizable code generator, in order to influence the depth and width of the symbolic execution tree, which can help evaluating the scalability quantitatively. The generated code contains a `switch` and `for` loops in each of the branches. This supports the analysis process, because the structure of the tree remains the same, however its size (width and depth) increase. I analyzed the following attributes in connection with scalability: depth of the tree, width of the tree, execution time with visualization, number of nodes. I set the depth between 5 and 250, while the width was set between 5 and 25. This resulted in 25 measurements.

Let us take a look at Figure 3.13, which presents the connection between execution time and the width of the tree in cases of different depths. The results show that the degree of growth of execution time depends on the depth, which can be seen in cases of depth 100 and 250. An interesting observation is that with 100 node depth (which could mean 2500 nodes), the execution time stays under 200 seconds.

Figure 3.14 shows the connection between depth and execution times with different widths. Based on the results, less deeper trees don't have extremely increased execution times in different widths, however with 250 depth there is a big jump. In this case, the width also greatly affect the execution times, since there is a large number (width) of very deep (height) branches. This explains why did the execution times grow in previous cases, where the depth and the number of nodes stayed the same.

Based on these results, an interesting experiment is the relational analysis between the number of nodes and the execution time, which is shown in Figure 3.15. As it can be easily seen, the values of the two variables are strongly correlated, since there is a quadratic polynomial trend line, that fits really well on the data. In this case, the quadratic scalability factor could be good in real-world applications.

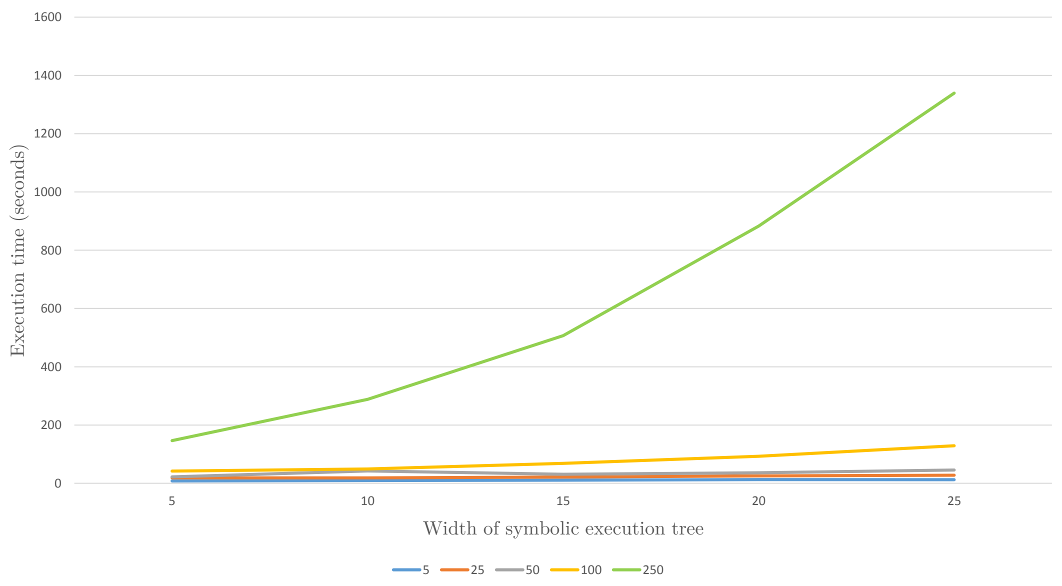


Figure 3.13: Relation between width (x) and execution time (y) in different depths (color).

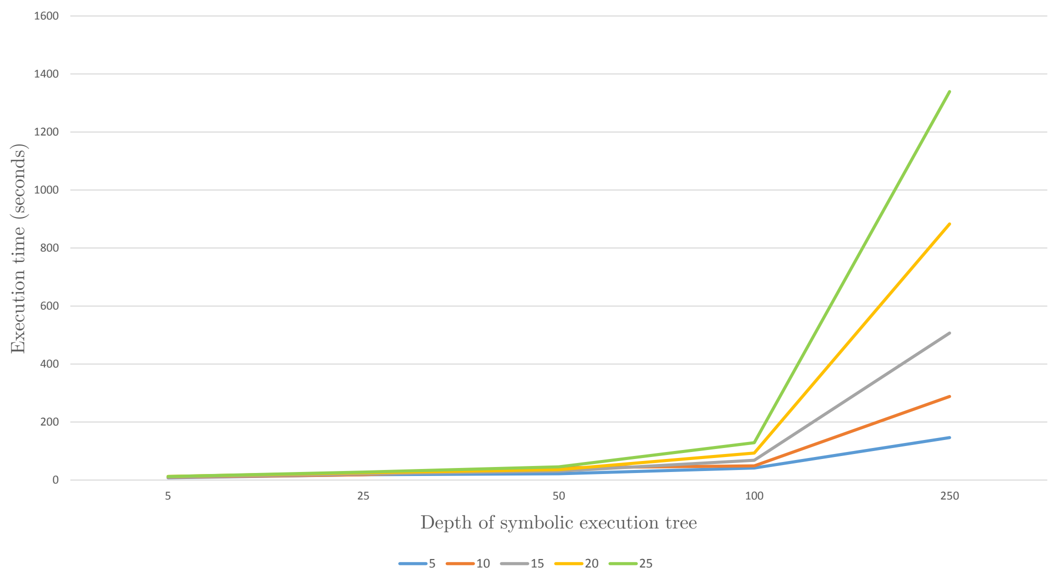


Figure 3.14: Relation between depth (x) and execution time (y) in different widths (color).

Summary of experiments

Two statements can be made from the results: 1) the visualization component affects the execution times in a variable manner, 2) the component scales very good even for large trees.

Unacceptable, extremely high growth of execution times were only experienced in cases of unbounded loops, recursions and hash functions. A solution could be to handle these branches with abstraction: detecting them during the execution, stopping the analysis until they are ended, then present it visually with abstraction (e.g. collapsed branch).

The quadratic scalability factor could be acceptable in real-world examples, since unit testing restricts the space to explored, and symbolic execution trees larger than 7000 nodes are less likely to be generated.

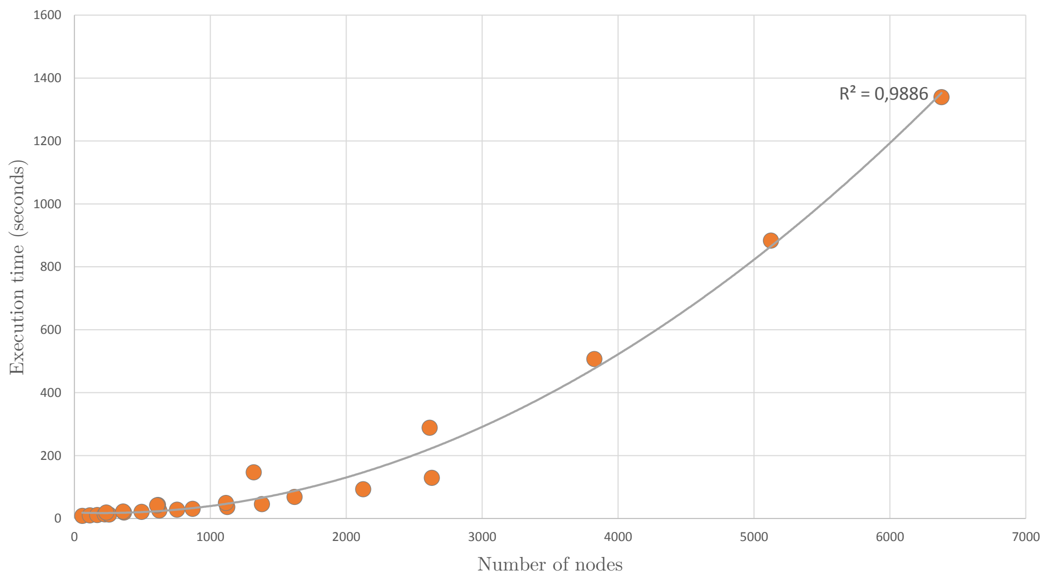


Figure 3.15: Scalability relation between number of nodes and time.

It is important to have the threats to validity of the experiments discussed, which could be the following.

- The experiments were manually, however I repeated each of them three times and presented their average, in order to have valid results.
- The artificial examples used in the experiments with execution time, are from an other research, thus its errors could affect also the results. To prevent this, I scrutinized the source code of each selected method.
- Although the cautious selection of real-world examples, they have the possibility to represent cases, that are not common in real-world applications. Therefore, I chose the examples from projects that were used by also other researches related to Pex.
- The low number of real-world examples may have the possibility, that a different set of selected methods could produce different results. That is the reason why I chose example methods from open-source projects beside the above mentioned ones.

3.5.3 Limitations of the approach

In this section, the potential limitations of the tool are collected and presented.

In its current form, the visualization technique cannot recognize and abstract loops, recursions and repeated statements. Thus, the symbolic execution can grow very huge and may have very deep branches, which can hinder the effective usage of the tool (there is a related research that addresses this problem).

Another limit of the usability of to tool is the integration into the testing process, since the ineffective use of the tool may also hinder and slow down the process instead of supporting it.

There are other hindering factors, that are related to the symbolic execution itself. Most important of them are the following [8].

- Handling multi-threaded environments
- Interaction with environment
- Handling external calls (e.g. external libraries)
- Handling floating point number representations
- State space explosion
- Optimization of constraint solvers

My implementation is currently lacking of optimization. The measurements also confirmed the hypothesis, that the monitoring of symbolic execution affects its execution time, thus the solution may requires deep analysis of the internal functionality of Pex.

Another limitation could be, that the Monitoring component only knows about the information provided by Pex through its API, which leads to three disadvantages: 1) hugely depends on Pex API and its further development, 2) it has restricted access to the executed source code, thus mapping is not always precise as possible, 3) new versions of Pex may break the interfaces used by the tool.

3.6 Related work

Several research have addressed the visualization of symbolic execution, however the motivation was different in each. Hentschel *et al.* gave a solution [20] for unbounded symbolic execution by abstracting the branches in cases of unbounded loops and recursions. Their tool (SED) is a symbolic execution-based debugger, that is able to visually represent the steps during tracing. They also mention the clear relation between control flow graphs and symbolic execution trees. The motivation of their research however was to increase the usability of their debugger.

Another, earlier research also approaches the visualization in the field of debugging. Hähnle *et al.*, precisely described [19] the visualization and the meaning of each attributes. Nevertheless, their motivation was the extension of their tool, in spite of my motivation that was the support of test generation in complex environments.

In summary, the research around is the visualization of symbolic execution is active, but not well-known. The motivation of the researches could be different, however the solutions from different aspects can help each other to support the applicability.

Chapter 4

Automated isolation environment generation

In this chapter, I present the idea of automated isolation environment generation. Firstly, the basic notions are presented, which are needed to understand the idea. Then, the technique is introduced in detail including an overview and its three main steps. Finally, during the evaluation, several examples and limitations are presented along with the related research.

4.1 Basic notions

To throughly understand our idea of automated isolation, I introduce the most important notions around this topic first.

4.1.1 Unit test isolation

As it was mentioned in Section 1.2.1, unit testing should be done in isolation, thus all the external dependencies of the unit should be removed or replaced. Naturally, removal of a call to a dependency is not an option, since it would alter the function of the code. Another solution could be to replace the external dependency with a *replacement object*, and a call is made into it. Decades ago, this idea and the increasing importance of unit testing led to a whole new area in software test engineering called *test doubles.*

Definition 7. *Test doubles* is the common name of static or dynamic objects, that can be used as a replacement of real objects during test executions, in order to handle the problem of isolation in unit testing. ▪

There are many types of test doubles, however the naming conventions can be different across publications. To overcome this, Meszaros wrote a summarizing book [29], that assesses the notions and patterns around unit testing, including test doubles too. I also applied the notions of this book in the area of test doubles, which are the following.

- *Dummy:* An object that only created to be passed, but has no effect on the results of the test execution, no values are calculated from it and its methods are called if they do not affect any result. A good example for this is a simple object that is passed as an argument for a method call that is outside of the testing scope.

- *Stub:* A replacement object with methods that return constant values for usage during test execution, however the values should not affect the success of the test case.
- *Mock:* Their return values are included in the test input data, therefore they can influence the success of a test case. One of their main property is that the calls (and their parameters and count) made to the object can be verified against the expected.

Test doubles can be formally defined as rules with LHS (left-hand side) and RHS (right-hand side). However, at first a relation between two methods must be defined to express the rules.

Definition 8. Let m_1 and m_2 two arbitrary methods. We say $m_1 \sim m_2$ (one is similar to the other), if the signatures of the methods are equal to each other. ▪

Now, the rule form of a test double can be defined.

Definition 9. Let A be an arbitrary class and F_A be its test double containing all the $m \in \mathcal{M}_A$ methods ($m_{F_A} \in \mathcal{M}_{F_A}$). When F_A is used instead of A during testing, then the following exists for any call to method m, if $m \sim m_{F_A}$:

$$m_A \Rightarrow m_{F_A},$$

that is any call to method m (LHS) is *detoured* to m_{F_A} (RHS), so the latter will be called transparently with getting all the argument values of the call, that can be used for verification. ▪

Testing with doubles may seem very easy, but many challenges exist in complex software environments, which must be solved in order to have effective unit testing. The most important factor is the *design for testability*, which enables easy implementation of test doubles. Let us take a look at the following example that introduces the problem of lack of testability in the design.

Example 9. *Let A be a class with method $m1(int) : int$, while let B be a class with method $m2() : bool$. The following source code shows a testability problem.*

```
class A
{
  private B b;

  A(int a)
  {
    b = new B(a);
  }

  int m1(int a)
  {
    if(b.m2()) return 10*a;
    else return 0;
  }
}
```

```
class B
{
  private int a;

  B(int a)
  {
    this.a = a;
  }

  bool m2()
  {
    if(a > 10) return true;
    else return false;
  }
}
```

The problem occurs when the unit U is defined as $U := \{ A \}$, thus the external dependency that must be isolated is $A.m1() \rightarrow B.m2()$. This means that a double F_B is needed, that replaces the object B used in class A and $m2 \Rightarrow m2_{F_A}$ is true.

It can be seen, that the tester of class A has no access to the field b. Moreover, the field has no possibility to be set or instantiated from outside, thus the object cannot be replaced with a double from outside. This is one of the biggest problems among testing of industrial software, since design for testability is not a common directive. Below, I introduce several solutions that can alleviate this problem.

Application of dependency injection pattern

Dependency injection is a software architecture design pattern, which ensures that a dependency of a software module can be replaced with another. This may be useful when working with replaceable units (e.g. different version of components or other functionality), moreover this pattern introduces maintainability, thus also testability to the source code.

In concrete implementation, as in the example, the problem occurs when a dependency is created (either with instantiation or using a factory) inside the component, and is not passed to it. This pattern solves this by requiring the dependencies to be passed through parameters or fields that are visible from outside. Certainly, this invokes several questions, like security in an API. The following idea solves this problem too.

Creating testing and maintenance interface

Instead of providing the ability to inject dependencies via parameters, an internal interface should be created for this purpose for each component. This interface can be only visible to testers in order to inject the dependencies and replace the desired objects during testing.

The concrete representation of a maintenance interface would be one or a set of internal (visible only inside the library) methods, that are able to alter the behavior of the component. This introduces some implementation overhead, although there are almost no security risks of publishing the rest of the component as an API.

Nonetheless, there are several situations when the source code cannot be altered, or the user of the library does not have access into it. In these cases, the third and most powerful solution comes into view.

Replacing in runtime

An interesting approach is replacing the dependencies and detouring the calls in a lower layer during runtime without altering the original source code. This method does not introduce any implementation overhead, does not have any security risks, however it may make developers feel comfortable, which can lead to have testability disrespected.

The concrete implementation of this idea raises some questions in detail due to its functionality. In managed environments like .NET and Java, it is possible to create detours runtime with their special programming interfaces. On the other hand, when one has to work in unmanaged environment, the source code is compiled directly to machine code, this solution will not work (due to e.g. accessing to running machine code, altering binary code).

4.1.2 Isolation frameworks

The implementation of test doubles is a time consuming task which is not always acceptable during development. Therefore, several supporting tools exist, one of them is isolation frameworks. These are software, that are able to automatically create double objects from analyzed methods. The user of the tool should only define the logic for the doubles (e.g. what to constantly return, what to assert), and in most of the tools, the user should also take care of injecting this object into the unit under test. Some isolation frameworks however provide the function of replacing objects in runtime. According to this functionality of the test framework, they can be categorized into two main groups.

Proxy-based

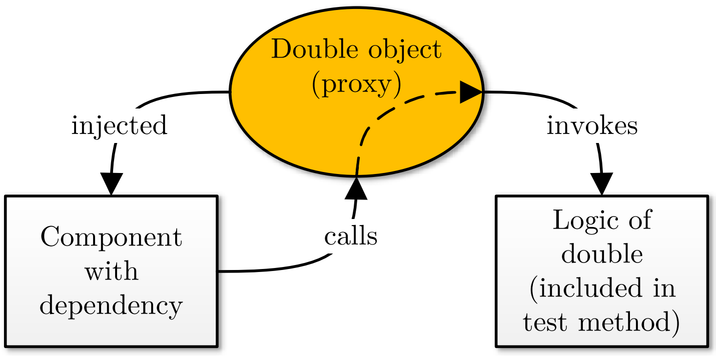

Double object (proxy)
injected
invokes
Component with dependency
calls
Logic of double (included in test method)

Figure 4.1: Overview of proxy-based isolation technique.

The proxy-based isolation frameworks are built on the idea of dependency injection. A double object is used to replace the original dependency, however the framework itself ensures that the user-specified logic can be defined anywhere, and is executed when the double object is called. In terms of implementation, the usage of the proxy pattern is the common technique to handle and detour the calls made to the double.

Most of the commonly used isolation frameworks are proxy-based, due to the relatively easy implementation and efficiency.

Profiler-based

Profiler-based frameworks use runtime detours in order to execute logic implemented inside double objects. Thus, the injection is not needed, however the implementation of this technique is far from trivial, since a lower layer is needed during runtime, where calls to methods can be caught and analyzed for further processing. This is can be relatively easily achieved in environments like Java and .NET, where the managed runtime is available for access. Unmanaged code however must be detoured in the lowest levels of runtime, which may not worth the effort due to the complexity of the problem.

In managed environments, a concrete solution could be to watch the stack frames, that are put onto the call stack. When an external dependency is called, it is detected and the call is detoured (the stack frame is popped and a fake is pushed onto). Certainly, there are other implementation possibilities, but I do not consider them here.

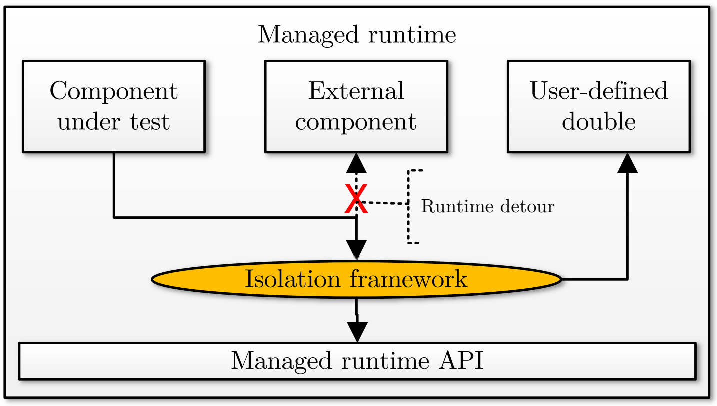


Figure 4.2: Overview of profiler-based isolation technique.

There are only a few isolation frameworks, that are profiler based, because the implementation requires deep knowledge of the environment in focus. However these tools are the most powerful, since they can be used to test legacy code or used to double all created instances of a type. Fakes is one of them, developed by Microsoft and is part of specific versions of Visual Studio.

4.1.3 Microsoft Fakes

Microsoft Fakes uses two main concepts, that should be clarified first.

- *Stub:* Stubs are lightweight double objects, that are used like an injected dependency. It requires testable structure of the source code and an interface for each objects that is used as dependency. The stub will implement this interface in order to replace the original object. In this case, Fakes works as a proxy-based framework. A great advantage is that, this technique works really fast compared to the other one.
- *Shim:* Shims are powerful doubles, which detour the code during runtime. They work with the CLR profiler API, and are able to fake almost everything, including internal and private methods. The main advantage of this approach is the ability to fake all instances of a type.

The usage of Fakes is very straightforward. If a double of an object is needed, a "fake" assembly should be created first from the original. Here, Fakes creates stub interfaces and shim types that can be used for making test doubles.

The next step is to implement a stub or a shim. Here, I introduce a simple example, where the methods of Example 9 are used.

Example 10. *As in the previous example, the unit U is defined as $U := \{ A \}$, thus the external dependency that must be isolated is $A.m1() \rightarrow B.m2()$. This can not be achieved by using dependency injection, so shims are needed with using Fakes.*

```
class ATest
{
  public void m1Test(A targetToTest, int a, bool m2Return)
  {
    // Definition of the sim with C# lambda expression
    ShimB.AllInstances.m2 = () => { return m2Return; };
    // Calling the method under test
    targetToTest.m1(a);
  }
}
```

In this example, a parameterized unit test is used, where I extended the list of parameters with the return value of the shim of method `m2()`*.*

4.2 Methodology

In order to support symbolic execution-based test generation, my idea is to generate the isolation environment automatically. The process uses the collected data from the symbolic execution. This section firstly gives an overview of the technique, then each step is described in detail.

4.2.1 Overview

The techniques build on top of parameterized unit tests in order to have test doubles, that can give back values, that are relevant to the component under test. A quick overview of the approach is presented in Figure 4.3.

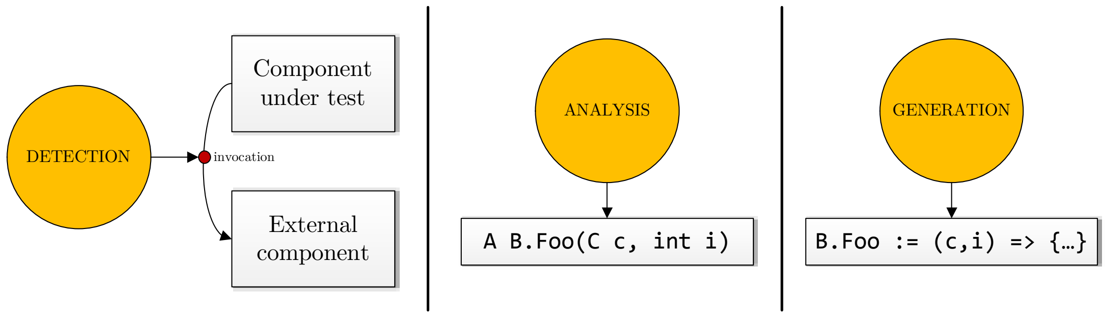


Figure 4.3: Overview of the automated isolation technique.

The automated generation of isolation environment relies on an analysis process, which is conducted when an invocation to a predefined external dependency is reached during the symbolic execution. Then, based on the results of the analysis, the generation step creates double objects that are able to replace the original ones.

4.2.2 Detection

Detection is the first phase of the isolation process. Firstly, the test engineer defines the unit or namespace under test with giving its fully qualified name (FQN). During the symbolic execution, this FQN is used for detecting an external call.

When an external invocation is detected, all the information regarding this call is collected and stored and used by the analysis step. The most important data are the followings.

- *Name of method:* The fully qualified name in order to be able to create the double method with the correct name.

- *Source of class:* Based on the FQN of the method, the source code of the container class should be gathered for later analysis. This is only done, when the class is unknown, otherwise the following is done.
- *Known methods of class:* If a new method is detected for an already known class, then the list of contained methods is extended with it.

The following algorithm gives an overview of the detection and storage step.

Algorithm 4 External call detection and data collection algorithm

```
function DetectionAndDataCollection(string unitUnderTest, Call call, KB kb)
    if !call.Method.StartsWith(unitUnderTest) then
        var document := GetSourceDocument(call.Method);
        if document then
            if kb.Contains(document) then
                kb[document].Methods.Add(call.Method);
            else
                kb.Add(document, new MethodList(cal.Method));
            end if
        end if
    end if
end function
```

4.2.3 Analysis

The analysis step is the most important of the three, since the decisions are made here, which decide how the double objects are generated. This step can be divided into three substeps, that are the *analysis of return value*, the *analysis of parameters* and the *assessment* of the scrutiny.

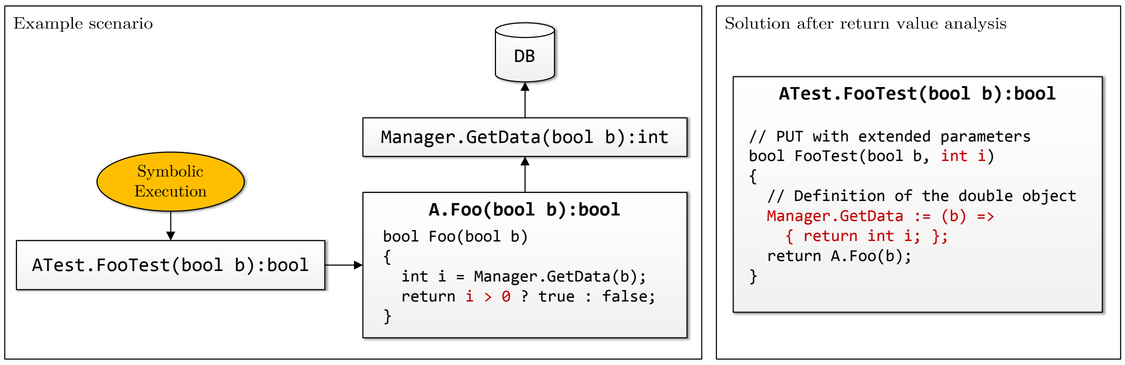
Return value analysis

In the first step, the return value of the invoked external method is analyzed, which is one of the most important information in a double object. The return value can be used in the unit under test as a branch condition or other statements, thus execution paths exists, which rely on this value. In order to cover these paths, the correct value must be selected.

If a path relies on the variable created from the return value, the symbolic execution interprets it as a term in the path condition. Problems occur, when the analysis can not provide proper inputs through this dependency (e.g. not yet implemented, gets value from database or file system), thus the coverage does not depend on it. This can be alleviated if the solver of symbolic execution can give concrete values for the variable that represents the return value. By this way, arbitrary values can be assigned to this variable and the coverage criteria (e.g. an execution path) can be satisfied that depends on the variable. The arbitrary values can be passed to the concrete execution through the parameters of the unit test. Example 11 shows a simple scenario of the idea.

Example 11. *Let us look at Figure 4.4, where an example scenario of the analysis is introduced. Let there be a class* `A` *and the class* `Manager`*, which reaches a database (DB). Method* `Foo` *of class* `A` *invokes* `GetData` *of the* `Manager`*. In this example, the U unit under test is* $U := \{A\}$*, thus the dependency to be isolated is* $A.Foo(bool) \rightarrow Manager.GetData(bool)$*.*

The symbolic execution starts from a parameterized unit test (`FooTest`) in a test class called `ATest`.


Example scenario
DB
Manager.GetData(bool b):int
Symbolic Execution
ATest.FooTest(bool b):bool
A.Foo(bool b):bool
Solution after return value analysis
ATest.FooTest(bool b):bool

```
bool Foo(bool b)
{
  int i = Manager.GetData(b);
  return i > 0 ? true : false;
}
```

```
// PUT with extended parameters
bool FooTest(bool b, int i)
{
  // Definition of the double object
  Manager.GetData := (b) =>
    { return int i; };
  return A.Foo(b);
}
```

Figure 4.4: An example scenario of return value analysis.

As described previously, a problem occurs during symbolic execution, when a value from an external dependency affects the coverage in the unit. In this example, the problematic branching is marked with red, since it relies on the value returned from the database (DB).

My idea provides a solution with the analysis of the return value. Two actions are done.

- *The parameters of the PUT is extended with the return value of the external dependency (here `int i`, marked with red).*
- *A double is created (also marked with red) in order to replace the behavior of the original `Manager` class. In the body of the double, the extension of the PUT parameter is returned, which gives the ability to symbolic execution to handle it as a free variable that can have arbitrary values.*

Parameter analysis

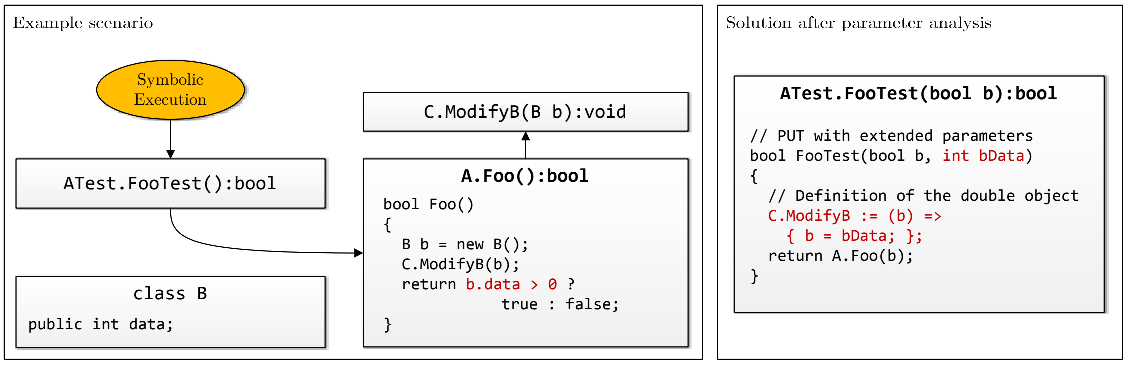
The analysis of parameters is the second step of my idea, however not all types of parameters are in focus. Method parameters can be primitive or complex types. In the two popular managed environments (.NET, Java) every complex type is handled as reference and the parameters are passed by value by default. Thus, when using reference type parameters, the reference itself is passed to the method as value, which means it is copied and refers to the same object. This enables the called method to modify the pointed object, which modifications can be also seen in the caller. However, the original reference cannot be modified. Note that .NET also provides the ability to pass the reference itself by using the `ref` keyword.

The reference type parameters lead to a problem in isolation scenarios, when the called method is an external dependency, because the passed object can be modified inside the dependency and therefore it can affect the coverage (as seen in the case of return values) in the unit under test. Example 12 shows a scenario for this problem.

Example 12. *The example shows the complex type parameter problem when the state of the object may be altered in the external method `ModifyB`. Then, the part of the object is used in a branching inside the unit which may not be covered during symbolic execution (due to e.g. file, database or cloud service access).*

```
class A
{
  public void Foo()
  {
    B b = new B();
    C.ModifyB(b);
    return b.data > 0 ? true : false;
  }
}
```

My idea to alleviate this is similar as in the case of return values, but the scenario is more complex. The first step is the same: extending the parameters of the PUT with the complex type parameter under analysis and handle it in the created double object. However, due to the complex type, there are numerous possibilities to modify the state of the object outside the unit. The idea is to explore the publicly available fields and properties of the object and use them to alter its state. By this way, I can simulate the actions made inside the external dependency that can be required to increase the coverage inside the unit under test. Figure 4.5 shows an overview of our idea.



```
public int data;
```

```
bool Foo()
{
  B b = new B();
  C.ModifyB(b);
  return b.data > 0 ?
            true : false;
}
```

```
// PUT with extended parameters
bool FooTest(bool b, int bData)
{
  // Definition of the double object
  C.ModifyB := (b) =>
    { b = bData; };
  return A.Foo(b);
}
```

Figure 4.5: An example scenario of parameter analysis.

Assessment

During the last step of the analysis all the collected information about the return values and parameters are filtered for duplications, then stored, which is used for double generation. Every method should contain the information that describe what to emit, when they are in the focus of code generation. This also includes the doubles of complex type parameters.

4.2.4 Generation

The generation is the last step of processing an external dependency, which can be divided into substeps. Firstly, the newly created parameters of the parameterized unit tests is emitted and appended to the original one. Then, the double of the method is assembled and emitted into the body of the PUT. This emission includes the name of the double method, which can be specific to isolation frameworks and also includes the inner body that can include setting of state modification for parameters and verification too. Finally, the doubles of the complex type parameters are generated that are property or field setter methods.

4.3 Implementation challenges

In order to implement the idea described in the previous section, I needed technologies that are able to traverse and analyze source code for exploring the external dependencies. Since I wanted the implementation as an extension for Microsoft Pex and Fakes, I searched for tools in .NET environment. Currently the Roslyn project also developed by Microsoft, is available for code analysis of .NET languages and it is considered as one of the most suitable tools for this purpose. The project itself recently went from community preview to a renewed open-source version and got the name .NET Compiler Platform. The platform itself consists of two main parts: 1) compiler as a service, 2) APIs for code analysis. Both of them were used to implement the idea of automated generation of isolation environment.

4.3.1 Extending Microsoft Pex

During the symbolic execution, the detection phase needs access to the currently invoked methods to decide whether it is an external call or not. Thus, I created an exploration extension package for Pex (as described in Section 1.4.4). Before the exploration I attach a handler to the event, which indicates that the symbolic execution entered into a new stack frame. This is a data structure that is put onto the call stack and contains information about the currently called method and its arguments.

When the event is fired, the required information is stored about the current stack frame, which includes the name of the method and its class, their source code document and the list of known methods. However, getting these information is not trivial, since file system access is needed. Thus, I used the services provided by the API of Pex to implement these functions. The process of obtaining the source code is very similar to the source code mapping described in Chapter 3 and Algorithm 2.

After the exploration is finished, the collected information is sent to analysis with the .NET Compiler Platform, then the desired code is generated from the assessed information.

4.3.2 Using .NET Compiler Platform

Naturally, the largest part of the implementation is related to Roslyn due to need for deep code analysis. In the following algorithms, the information assessment is described in detail.

The code analysis in Algorithm 5 contains many important steps. Firstly, the inclusion in the unit is examined. If the namespace of the currently visited source document is included in the unit, then a syntax tree is created with the help of Roslyn. This is followed by a runtime compilation in order to get the semantic model, which contains type information of the types included in the source code. By this way it can be decided whether a type is a primitive or reference type. Then, the algorithm iterates through the parameters and inspects if it has formerly stored information from the analysis phase. If yes, the source code is parsed and walked through to get the required information for the generation. Finally, the whole set of fakes along with the extension parameters to the PUT are generated. It can be seen that the two most important methods are the method- and property-walkers that are analyzing the source. Algorithm 6 and Algorithm 7 shows the data collection procedures for the methods and properties of complex type parameters respectively.

Algorithm 5 Overview algorithm of the code analysis process.

```
function AnalyzeCode(KB knowledgeBase)
    for each doc in knowledgeBase do
        if IsIncludedInUnit(doc) then
            var syntaxTree := Parse(doc);
            var semanticModel := syntaxTree.Compile().GetSemanticModel();
            var methodData := WalkMethods(syntaxTree, semanticModel);
            for each method in methodData do
                var propertyGetters := new[];
                for each param in method.Parameters do
                    if !IsPrimitive(param) then
                        var foundClass := knowledgeBase.GetClass(param.Class);
                        if foundClass then
                            var paramClassTree := Parse(foundClass.SourceDocument);
                            var propertyData := WalkProperties(paramClassTree);
                            for each property in propertyData do
                                EmitPUTExtension(property);
                                propertyGetters.Add(property);
                            end for
                        end if
                    end if
                end for
                Generate(method,propertyGetters);
            end for
        end if
    end for
end function
```

The discovery of the properties that can be set from outside is far from trivial. Firstly, the algorithm has to check if the property itself is visible from outside. Then, the setter of the property is checked if it exists, and if yes, the algorithm makes sure that it has no accessibility modifiers (e.g. a private setter). A very similar algorithm can be applied to fields, that are not visible from outside only through getter and setter methods. The only complexity is the discovery of the methods that are able to set and get the values for the field. Since our current prototype implementation do not support fields, I do not discuss this here in detail.

The analysis can get more complicated if the parameters or the types are static, generic or have some special visibility attributes. These corner-cases must be handled correctly in order to have a set of efficiently working doubles.

4.3.3 Generation with Microsoft Fakes

Generating doubles for Microsoft Fakes requires sticking to strict guidelines, since the framework has precisely defined name of double objects and methods that are put together from the parameter types, the name of the method and the special attributes (e.g. static, generic). The mocks generated by the tool must use these names to avoid the intervention of the user, which is one of the goals of the tool.

Algorithm 6 Algorithm of walking over the method declarations.

```
1: function VisitMethodDeclaration(MethodDeclaration node)
2:     var name := node.GetFullName();
3:     var type := node.GetTypeName();
4:     var typeInfo := semanticModel.GetTypeInfo(type);
5:     var paramTypeInfo := new[];
6:     for each var param in node.parameters do
7:         paramTypeInfo.Add(semanticModel.GetTypeInfo(param.Type));
8:     end for
9:     knowledgeBase.Add(name,type,typeInfo,paramTypeInfo[]);
10: end function
```

Algorithm 7 Algorithm of walking over the property declarations.

```
1: function VisitPropertyDeclaration(PropertyDeclaration node)
2:     if node.Modifiers.Contains("public") then
3:         var setter := node.Accessors.Get("set");
4:         if setter then
5:             if setter.Modifiers.Count == 0 then visibleProperties.Add(node);
6:             end if
7:         end if
8:     end if
9: end function
```

Currently, the tool supports the generation only with Microsoft Fakes, however I created the architecture of the tool to be capable of using other isolation frameworks. Naturally in these cases, the generator phase must be rewritten.

4.4 Evaluation

As described in the previous sections, the problem to solve is the low coverage due to the large number of dependencies in real-world applications. In the following examples, I introduce some use cases where the automated isolation environment generation can alleviate the problem. Firstly, three basic examples are presented, which is followed by two real-world examples.

4.4.1 Basic examples

In the following three examples, three classes are used to demonstrate the usage of the implemented tool in different basic scenarios. Class `A` is the unit under test, class `B` is the external dependency of class `A` and it is not currently implemented, thus they cannot be used during testing the unit. Class `C` is the common part (used by both class `A` and `B`), and can be thought as also part of the unit under test. Figure 4.6 overviews the scenario. The source code of class `B` and `C` can be found in F.1, while the source of class `A` is introduced in the examples.

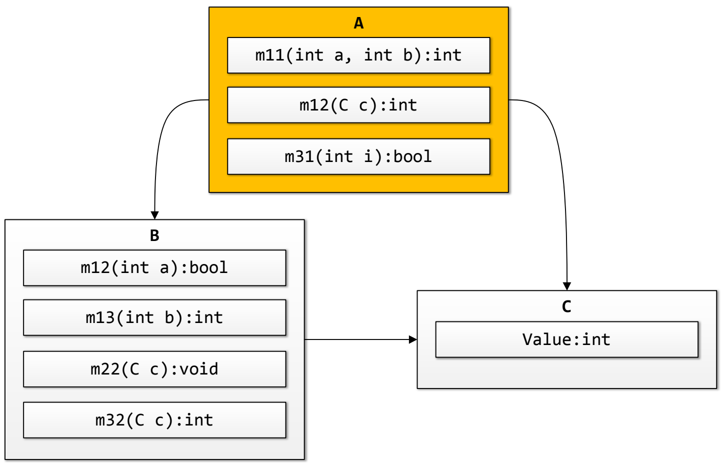


Figure 4.6: The architecture of the simple example.

Multiple dependencies

Preamble In the first basic example, the unit under test U can be defined as $U := \{A\}$. The source code of the method under test (`m11(int,int)`) can be found below. It can be seen that two external calls are made to the dependency, thus the dependencies are: $\mathcal{D}_{A,B} := \{A.m11(int,int) \rightarrow B.m12(int), A.m11(int,int) \rightarrow B.m13(int)\}$. Both of them should be isolated in order to do clear unit testing and have tests that do not depend on the implementation state of class `B`.

```
public int m11( int a, int b)
{
  if (B.m12(a)) return 0;
  if (B.m13(b) < 10) return 1;
  return 2;
}
```

First execution Since method `m12(int)` currently returns always true, thus the only branch that is covered in the first execution is the first one, where the return value is 0. I applied the isolation package for the parameterized unit test and defined the unit as class `A`.

Generated isolation environment The tool was able to generate the mock for method `m12(int)`. However, when I ran Pex again, the coverage was not 100% due to the second external call, where `m13(int)` always returns 100. I attributed the PUT once more to detect the second dependency. Then, I achieved the full coverage with the two mocks. The generated extension parameters for the PUT and the mocks can be found below.

- `bool returnm12`
- `int returnm13`

```
ShimB.m12Int32 = (p0) =>
{
  return returnm12;
};

ShimB.m13Int32 = (p0) =>
{
  return returnm13;
};
```

Results and summary With the help of both of the mocks I achieved 100% code block coverage, which was the goal in the testing this unit. It must be noted however that the scenarios like this may require multiple executions to have all of the dependencies isolated. This is due to the fact that the tool relies on the steps of Pex: if Pex does not traverse a statement, the tool cannot detect the dependency there.

Depending on object state

Preamble The second example is about the state of the external objects that are passed to external dependencies. These are interesting because their state can be modified inside the dependency and then used again by the unit under test. The source code below shows that the `C`-typed object has a `Value` property, which is examined before and after the external call. If modifications are made to the state of the object, the method returns 2.

```
public int m21(C c)
{
  int i = 0;
  if (c.Value < 10) i = 1;
  B.m22(c);
  if (c.Value > 10 && i == 1) i = 2;
  return i;
}
```

First execution As expected, the first execution gave test cases that were not able to cover the second branching, because method `m22` is not implemented yet and cannot alter the state of the object passed as the argument. Thus, mocks are needed to cover the missing branch.

Generated isolation environment I attributed the PUT with the tool, and chose unit U as $U := \{A, C\}$. Thus the dependency that must be mocked is $\mathcal{D}_{A,B} := \{A.m21(C) \rightarrow B.m22(C)\}$. The tool generated the following isolation environment and PUT parameter with only one execution.

- `C cPropValue`

```
ShimB.m22C = (p0) => {
  p0.Value = cPropValue;
};
```

Results and summary By using the generated mock, Pex was able to generate full code block coverage for the method with the test cases. This is achieved by altering the state of the object in the dependency, by modifying the value of its property.

Complex usage of dependency

Preamble In the last simple example, I introduce a method, where the return value and the state of the object is also used as a branching condition in a more complex arithmetic. Namely, the method returns true if the parameter `i` multiplied by the sum of the return value and the value property of the object `c` is larger than 100, otherwise it returns false. The source code of the method is found below. It can be seen that if the unit is defined as $U := \{A, C\}$, then the dependency to be isolated is $\mathcal{D}_{A,B} := \{A.m31(int) \rightarrow B.32(C)\}$.

```
public bool m31(int i)
{
  C c = new C();
  c.Value = 0;
  int ret = B.m32(c);
  if ((ret + c.Value) * i > 100) return true;
  else return false;
}
```

First execution In the first execution, Pex could not cover the branch, where the method return true, since the external dependency always returns true and the value of object `c` is always 0. The mock to be used should return a value and alter the `Value` property of object `c` so that with the multiplication by `i`, it is larger than 100.

Generated isolation environment The tool generated two parameters for the PUT, that must be inserted into to its list of parameters. Furthermore, a mock was generated to method `m2(C)`, which sets a value to the property `Value`. The generated code and the parameters of the isolation environment is the following.

- `int cPropValue`
- `int returnm32`

```
ShimB.m32C = (p0) => {
  p0.Value = cPropValue;
  return returnm32;
};
```

Results and summary By using the newly generated isolation environment, Pex was able to generate the missing test case to cover the branch with the `return true` statement. It generated 999 for `cPropValue` and 976 for `returnm32` parameters.

4.4.2 Real-world examples

In order to have an overview of the usage of the tool in real-world environments I present two examples from industrial software, where the tool was able to help increasing coverage in the first executions without the guidance of the test engineer.

Content management system

The following example, inspired by the system presented in Section 2.2.2, mimics the server-side components of typical business applications.

The architecture of the software is similar to a simple three-layered application. It has a database, which is accessed by the data access layer (DAL) through a stored procedure executor. One layer above the service layer is found, which contains the business logic, and accesses the DAL with the help of a manager object that folds the concrete calls to the other layer. Naturally, the top layer is the graphical user interface, however in this experiment it is out of the scope, thus I do not discuss it here. Figure 4.7 shows the overview of the architecture and the scope of testing marked with orange.

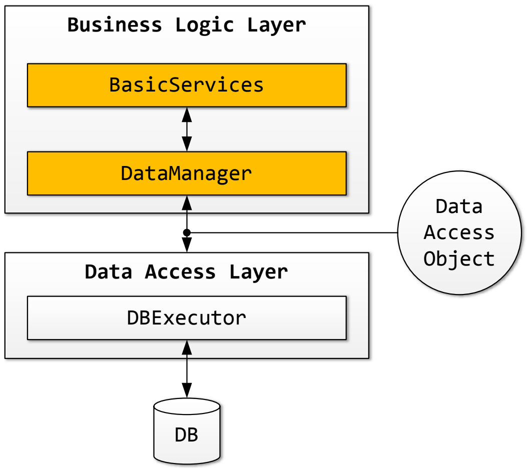


Figure 4.7: The architecture of the CMS software.

Preamble The `BasicServices` class contains a method, which modifies a user of the system with some predefined values that are passed via arguments. This is the entry point for the symbolic execution. In this method, the `DataManager` is invoked with method `ModifyUser(UserDAO)` to execute an action to the database. In this scenario, the database executor has a return value. If it is positive, the action was successful, otherwise it was not. It is important to cover the two cases, because when the action is invalid, the code throws an exception, which must be handled. Furthermore, by specification, the method uses the same object during the operation, thus when it finishes, the reference for the object should contain the modifications. This is used, when the logic checks if the identifier of the user stayed 0 after the action. If yes, it means a problem, since the modified user cannot have the id of 0. In these situations, an exception is thrown with indicating the invalid user. Naturally, during testing, I did not use any database, I supposed instead that the data access layer is yet incomplete. The important code snippets can be found in F.2.

First execution The first execution generated 1 test case covering 15 out of 23 code blocks. This means that the two branchings after the call to the database were not fully covered. In order to increase this coverage, I applied the Pex package for automated isolation environment generation by attributing the parameterized unit test. I defined the unit under test as the scope of testing shown in Figure 4.7.

Generated isolation environment The prototype implementation generated the following. One mock for the `DBExecutor.ModifyUser` method and five settings for the properties of `UserDAO` class, which is the only parameter of the executor: it is a reference type and it is out of the unit under test. Their code to be inserted into the PUT can be found below. Furthermore, to have mocks working, the tool generated that the following parameters should be added to the PUT.

- `long userdaoPropId`
- `string userdaoPropEmailAddress`
- `string userdaoPropFullName`
- `DateTime userdaoPropDateOfBirth`
- `List<string> userdaoPropInterests`
- `int returnModifyUser`

```
ShimDBExecutor.ModifyUserUserDAO = (p0) => {
  p0.Id = userdaoPropId;
  p0.EmailAddress = userdaoPropEmailAddress;
  p0.FullName = userdaoPropFullName;
  p0.DateOfBirth = userdaoPropDateOfBirth;
  p0.Interests = userdaoPropInterests;

  return returnModifyUser;
};
```

Results and summary With the newly generated isolation environment Pex was executed again. The results showed that 2 new test cases were generated being able to cover the branches that were not covered before. Thus, the coverage increased from 15 to 18 code blocks out of 23. In summary, the prototype tool was able to increase the coverage and provided two new test cases with only one discovery execution.

JustDecompile Engine

The second real-world example is from a commercial product, which recently went open-source. It is a .NET assembly decompiler called JustDecompile and developed by Telerik [37]. I selected a method to test and demonstrate the usage and benefits of the tool. The architecture of the tested component (`Utilities`) can be seen on Figure 4.8, where the method under test is marked with orange.

Preamble The method under test is `IsTypeNameInCollisionOnAssemblyLevel(string,AssemblySpecificContext,ModuleSpecificContext)` and its source code can be found in F.3. The task is to decide if the currently analyzed type during the decompilation collides with an other type in the same assembly. The method uses two external dependencies: `AssemblySpecificContext`, `ModuleSpecificContext`. Both of them are used as parameters to decide if the `using` statements in the decompiled assembly cause type ambiguity on assembly level. I chose this method, because it has two dependencies, that can be injected and the logic of the method relies on their state. However, the state of these two object cannot be set from outside, because they have private setters. The tool can help discover these problematic dependencies. It must be stated that I made a small

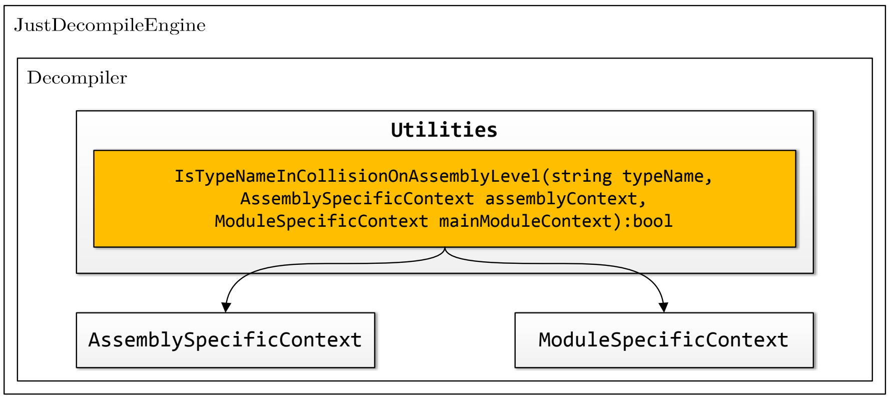

JustDecompileEngine
Decompiler
Utilities
IsTypeNameInCollisionOnAssemblyLevel(string typeName, AssemblySpecificContext assemblyContext, ModuleSpecificContext mainModuleContext):bool
AssemblySpecificContext
ModuleSpecificContext

Figure 4.8: The architecture of the tested part of JustDecompile Engine.

modification to the code due to the prototype state of the tool: I replaced the property getters with getter methods (the logic stayed the same).

First execution With the first execution Pex generated 14 test cases, however the coverage of the method stayed very low, since all of the test cases failed due to exceptions. 11 of them raised `NullReferenceException`, while the remaining caused `ArgumentNullException` to occur. Let us take a look at the source code of the method: the test cases were only able to reach the set intersection call. The statements followed by this line were not covered at all.

Generated isolation environment The tool generated the mocks and parameters found just below. It can be noticed that all of the external calls were caught and a mock was created for them. Running the test in this isolated environment did not lead to increase of coverage, since Pex insisted on `null` inputs. Thus, I had to make assumptions that restrict the range of values, which can be returned by the mocks. I excluded `null` values and also required to have not `null` elements of certain collections in order to avoid `ArgumentNullExceptions`.

- `ICollection<string> returnGetAssemblyNamespaceUsings`
- `ICollection<string> returnGetModuleNamespaceUsings`
- `Dictionary<string, List<string>> returnGetCollisionTypesDatae`

```
ShimAssemblySpecificContext.AllInstances.GetAssemblyNamespaceUsings = (p0) =>
{ return returnGetAssemblyNamespaceUsings; };

ShimModuleSpecificContext.AllInstances.GetModuleNamespaceUsings = (p0) =>
{ return returnGetModuleNamespaceUsings; };

ShimModuleSpecificContext.AllInstances.GetCollisionTypesData = (p0) =>
{ return returnGetCollisionTypesData; };
```

Results and summary After introducing the modifications that were mentioned above, Pex generated 5 test cases, which were able to cover all of the code blocks in the method including the two different return values too. This coverage could be only achieved by mocking due to the private setter problem, however in this example, the tool generated the required mocks automatically, thus alleviated the work during testing.

4.4.3 Limitations

Currently, the approach has two kinds of main limitations: limitations of the algorithms and approach itself. Both of them are discussed below.

Limitations of the approach

Firstly, the idea has *scalability limitations* due to the underlying techniques. When a mock is created, my technique extends the list of parameters by one or more, which increases the state space for symbolic execution. This may largely affect and harden the task for the underlying solver, which generates concrete values. However, this also means that as the underlying techniques evolve, my approach is also alleviated to be applied.

Secondly, there are *several cases in a source code* that must be handled, which hinder the implementation to prepare for all of them. This includes e.g. generic and static types or different visibility questions.

Finally, a hindering factor of the approach could be that in large software systems, the technique does not fulfill its requirements, so that it *may require human intervention* and manual inspection of the generated mocks. This can be due to the large number of generated mocks, which unambiguously leads back to the previous limitation.

Limitations of the algorithms

The following limitations are concerned with the current, proof of concept version of the implementation, but they can be important due to the underlying challenges.

One of the main limitation is that currently the tool only handles external dependencies that have source code attached. The challenge here is to analyze an interface with the Roslyn API without parsing the code into a syntax tree and without runtime compilation which provides type information. Thus, in these cases I will need to get these information from the compiled assembly.

Another limitation is the problem of interfaces and abstract classes that cannot be instantiated, when they are put onto the parameters of the parameterized unit test. This problem also hinders the state modification of complex type parameters in the body of the double object, since the tool do not have access to these. An interesting approach would be to inspect the types during symbolic execution and analyze the superclasses and interfaces of them. Thus, if a suitable class is found then the technique would use that instead of the interface or abstract class.

Currently, this proof of concept implementation does not handle corner cases like static or generic types and also avoids discovering fields instead of properties. These are very common scenarios in real-world software, thus for effective industrial application of this technique, these cases must be covered with solutions.

4.5 Related work

My idea originally derives from a paper written by Tillmann *et al.* [40], where the idea of mock object generation is described. They also created a case study for file-system dependent software [28], which showed promising results. Their technique is able to automatically create mock objects with behavior and ability to return symbolic variables, which is used during the symbolic execution to increase the coverage of the unit under test. However, their solution needs the external interfaces explicitly added to the parameterized unit tests, moreover they did not care about reference type parameters that can affect the coverage. Thus, my solution covers more wider area of scenarios and needs rather less user interaction for the automated generation (my idea only requires the namespace of the unit under test).

The idea of Galler *et al.* is to generate mock objects from predefined design by contract specifications [14]. These contracts describe preconditions of method, thus the derived mocks are also sticking to them, which makes them able to avoid false behavior. However, their approach does not relate to symbolic execution, and it may also introduce work overhead to create contracts. A similar approach is introduces along with a symbolic execution engine to Java by Islam *et al.* [25]. The difference is that they build on interfaces as specifications instead of contracts.

An other approach of mock generation was presented by Pasternak *et al.* [31]. They created a tool called GenUTest, which is able to generate unit tests and so-called mock aspects from previously monitored concrete executions. However, the effectiveness of the approach largely relies on the completeness of previous concrete executions, while my approach relies on the symbolic execution.

A very interesting approach is presented by Godefroid in [16]. He introduced the idea of micro execution, where parts of an arbitrary x86 program can be executed independently, while the memory operations are monitored and caught before they occur. The values to be returned for these operations are generated by other tools or randomly. This can be thought as a form of unit isolation in the lowest level as possible. I mentioned in Section 4.1.1 that an approach like this creates several challenges and raises countless questions, but micro execution may be able to provide solutions to some of them according to the preliminary results.

Chapter 5

Summary

In this chapter, the summary of the results is introduced, and the related future work is also presented.

5.1 Results

In this thesis, the topic of supporting automated test input generation was analyzed. The main research question was the following:

How can the industrial application of symbolic execution-based test generation be supported?

Identification of challenges

I managed to identify several challenges in industrial usage of symbolic execution-based automated test input generation. One of the most significant is its application in the real-world software due to the large complexity of the source codes. I experimented with two case studies from which the results supported the statements of the related research in this topic. Namely, test input generation requires support in order to improve its applicability in real-world scenarios. I analyzed two techniques in detail, which can be able to fulfill the supporting requirements and thus able to alleviate the test input generation in such complex software. I stated the following research questions, which are answered in the thesis.

1. What type of representation can help reduce the lack of perspicuity?
2. What kind of information can be obtained during symbolic execution?
3. What type of information can be represented for the explorations?
4. How the problem of isolation can be alleviated or solved in order to support test generation?

Visualization

The visualization of symbolic execution is an active research topic, however I applied the visualization to alleviate the work with symbolic execution. The first three research question could be answered with the definition of the visualization technique. I chose a graph representation for the visualization for which the appearance is described in detail. Furthermore, I defined and selected the most valuable metadata that can be acquired during symbolic execution and are able to support the work of test engineers. I also described the appearance of these metadata for the graph.

The visualization technique was implemented in a form of a tool, which is open-source and publicly available on GitHub. I experimented with the tool, and the results showed that it may be applied in real-world scenarios, since it does not introduce tremendous overhead for the symbolic execution.

Automated isolation

One of the discovered challenges in real-world scenarios was the unit isolation for unit testing, since these applications have several external dependencies in each of their components (e.g. databases, external services or even lack of design for testability). Isolating the dependencies requires large amount of time, which can be reduced by automation.

The described isolation technique in this thesis could answer the last research question. During my work, I defined the technique in detail including its basic ideas and algorithms. The main idea is to detect the dependencies during the symbolic execution and automatically generate the isolation environment for the unit under test from the collected data. I also presented the capabilities through a prototype implementation by using three basic and two real-world examples.

The results showed that there is great potential in the idea and may be able to alleviate the work with symbolic execution-based test input generation in real-world scenarios and environments.

5.2 Future work

During my research and experiments I managed to reveal new limits and challenges among the analyzed topics. The following future works may be able to increase the usability of symbolic execution for engineering applications.

- Handling loops, recursions and deep execution traces in the symbolic execution tree (e.g. with collapsing branches), which can make the represented graph clearer.
- Optimization of the data collection algorithm may be able to reduce the monitoring overhead of the generation of visualization.
- Detailed case studies of the applicability of visualization in order to support the preliminary results.
- Expanding the implementation of the automated isolation environment generation to cover all the corner cases mentioned in the thesis.
- Real-world experiments and measurements for the automated isolation technique to confirm the usability of the technique.
- Combination of automated isolation and compositional symbolic execution may lead to a new level of automated test input generation, where the work of test engineers can be greatly alleviated.

Köszönetnyilvánítás

Ezúton is szeretném kifejezni hálámat konzulenseimnek Dr. Micskei Zoltánnak és Vörös Andrásnak, akik közös erővel támogattak az utóbbi három év munkája során. Az ő útmutatásuknak köszönhetem az elmúlt időszakban elért sikereimet, amelyek megalapozták szakmai pályafutásomat.

Köszönettel tartozom továbbá Theisz Zoltánnak, aki munkámat szakmai megbeszéléseink során ötletekkel és tanácsokkal segítette.

List of Figures

Bibliography

[1] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil Mcminn. An Orchestrated Survey of Methodologies for Automated Software Test Case Generation. *J. Syst. Softw.*, 86(8):1978–2001, August 2013.

[2] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.

[3] Antonia Bertolino. Software Testing Research: Achievements, Challenges, Dreams. In *Future of Software Engineering, 2007. FOSE '07*, pages 85–103, May 2007.

[4] Pietro Braione, Giovanni Denaro, Andrea Mattavelli, Mattia Vivanti, and Ali Muhammad. Software Testing with Code-based Test Generators: Data and Lessons Learned from a Case Study with an Industrial Software Component. *Software Quality Journal*, 22(2):1–23, 2013.

[5] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High–coverage tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[6] 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.

[7] Cristian Cadar and Koushik Sen. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM*, 56(2):82–90, February 2013.

[8] Ting Chen, Xiao song Zhang, Shi ze Guo, Hong yuan Li, and Yue Wu. State of the Art: Dynamic Symbolic Execution for Automated Test Generation. *Future Generation Computer Systems*, 29(7):1758 – 1773, 2013.

[9] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. DySy: Dynamic Symbolic Execution for Invariant Inference. In *Proc. of the 13th international conference on Software engineering - ICSE '08*, pages 281–290, 2008.

[10] Jonathan de Halleux and Nikolai Tillmann. Parameterized Unit Testing with Pex. In Bernhard Beckert and Reiner Hähnle, editors, *Tests and Proofs*, volume 4966 of *Lecture Notes in Computer Science*, pages 171–181. Springer Berlin Heidelberg, 2008.

[11] Leonardo de Moura and Nikolaj Bjorner. Z3: An Efficient SMT Solver. In C.R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction*

and Analysis of Systems, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin Heidelberg, 2008.

[12] eqqon GmbH. GitSharp. http://www.eqqon.com/index.php/GitSharp, 2013.

[13] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. Does Automated White-box Test Generation Really Help Software Testers? In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 291–301, New York, NY, USA, 2013. ACM.

[14] Stefan J Galler, Andreas Maller, and Franz Wotawa. Automatically Extracting Mock Object Behavior from Design by Contract$^{\text{TM}}$ Specification for Test Data Generation. In *Proceedings of the 5th Workshop on Automation of Software Test*, pages 43–50. ACM, 2010.

[15] Emden R. Gansner and Stephen C. North. An Open Graph Visualization System and its Applications to Software Engineering. *SOFTWARE - PRACTICE AND EXPERIENCE*, 30(11):1203–1233, 2000.

[16] Patrice Godefroid. Micro Execution. In *Proceedings of the 36th International Conference on Software Engineering*, pages 539–549. ACM, 2014.

[17] Patrice Godefroid, Michael Y. Levin, and David Molnar. SAGE: Whitebox Fuzzing for Security Testing. *Queue*, 10(1):20:20–20:27, January 2012.

[18] Gotlieb, Arnaud and Botella, Bernard and Rueher, Michel. Automatic Test Data Generation using Constraint Solving Techniques. In *ACM SIGSOFT Software Engineering Notes*, volume 23, pages 53–62. ACM, 1998.

[19] Reiner Hähnle, Marcus Baum, Richard Bubel, and Marcel Rothe. A Visual Interactive Debugger Based on Symbolic Execution. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 143–146, New York, NY, USA, 2010. ACM.

[20] Martin Hentschel, Reiner Hähnle, and Richard Bubel. Visualizing Unbounded Symbolic Execution. In Martina Seidl and Nikolai Tillmann, editors, *Tests and Proofs*, volume 8570 of *Lecture Notes in Computer Science*, pages 82–98. Springer International Publishing, 2014.

[21] David Honfi, Andras Voros, and Zoltan Micskei. SEViz: A Tool for Visualizing Symbolic Execution. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, pages 1–8, April 2015.

[22] Dávid Honfi, Zoltán Micskei, and András Vörös. Support and Analysis of Symbolic Execution-based Test Generation, TDK thesis, BME. 2014.

[23] IEEE. IEEE Standard Classification for Software Anomalies. *IEEE Std 1044-1993*, pages i–, 1994.

[24] IEEE. IEEE Standard for Software and System Test Documentation. *IEEE Std. 829-2008*, 2008.

[25] Mainul Islam and Christoph Csallner. Dsc+Mock: A Test Case + Mock Class Generator in Support of Coding against Interfaces. In *Proceedings of the Eighth International Workshop on Dynamic Analysis*, pages 26–31. ACM, 2010.

[26] ISTQB. Foundation Level Syllabus, 2011.

[27] James C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7):385–394, 1976.

[28] Madhuri R. Marri, Tao Xie, Nikolai Tillmann, Jonathan De Halleux, and Wolfram Schulte. An Empirical Study of Testing File-System-Dependent Software with Mock Objects. *Proceedings of the 2009 ICSE Workshop on Automation of Software Test, AST 2009*, pages 149–153, 2009.

[29] Gerard Meszaros. *XUnit Test Patterns: Refactoring Test Code*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

[30] Alessandro Orso and Gregg Rothermel. Software Testing: A Research Travelogue (2000&#8211;2014). In *Proceedings of the on Future of Software Engineering*, FOSE 2014, pages 117–132, New York, NY, USA, 2014. ACM.

[31] Benny Pasternak, Shmuel Tyszberowicz, and Amiram Yehudai. GenUTest: a Unit Test and Mock Aspect Generation Tool. *International journal on software tools for technology transfer*, 11(4):273–290, 2009.

[32] CodePlex project. Data Structures and Algorithms. https://dsa.codeplex.com/, 2008.

[33] CodePlex project. .NET Bio. http://bio.codeplex.com/, 2013.

[34] Corina S. Păsăreanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehlitz, and Neha Rungta. Symbolic PathFinder: Integrating Symbolic Sxecution with Model Checking for Java Bytecode Analysis. *Automated Software Engineering*, 20(3):391–425, 2013.

[35] Xiao Qu and B. Robinson. A Case Study of Concolic Testing Tools and their Limitations. In *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on*, pages 117–126, Sept 2011.

[36] Christian Rodemeyer. Dot2WPF - a WPF Control for Viewing Dot Graphs, 2007.

[37] Telerik. JustDecompile Engine. https://github.com/telerik/justdecompileengine, 2015.

[38] Nikolai Tillmann and Jonathan de Halleux. Pex–White Box Test Generation for .NET. In Bernhard Beckert and Reiner Hähnle, editors, *Tests and Proofs*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer Berlin Heidelberg, 2008.

[39] Nikolai Tillmann, Jonathan de Halleux, and Tao Xie. Transferring an Automated Test Generation Tool to Practice: From Pex to Fakes and Code Digger. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 385–396, New York, NY, USA, 2014. ACM.

[40] Nikolai Tillmann and Wolfram Schulte. Mock-Object Generation with Behavior. *Proceedings - 21st IEEE/ACM International Conference on Automated Software Engineering, ASE 2006*, pages 365–366, 2006.

[41] Xusheng Xiao, Sihan Li, Tao Xie, and Nikolai Tillmann. Characteristic Studies of Loop Problems for Structural Test Generation via Symbolic Execution. In *Proc. 28th IEEE/ACM International Conference on Automated Software Engineering (ASE 2013)*, November 2013.

[42] Xusheng Xiao, Tao Xie, Nikolai Tillmann, and Jonathan De Halleux. Covana : Precise Identification of Problems in Pex. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 1004–1006, 2011.

[43] Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. Fitness-Guided Path Exploration in Dynamic Symbolic Execution. In *Dependable Systems Networks, 2009. DSN '09. IEEE/IFIP Int. Conf. on*, pages 359–368, 2009.

Appendix

F.1 Source code of simple isolation examples

```
// To be implemented...
public static class B
{
  public static bool m12(int a)
  {
    return true;
  }

  public static int m13(int b)
  {
    return 100;
  }

  public static void m22(C c) { // To be implemented... }

  public static int m32(C c)
  {
    return 0;
  }
}

public class C
{
  public int Value { get; set; }
}
```

F.2 Source code snippets of the examined CMS

```
public void BasicServices.ModifyUser(string emailAddress, string fullName, DateTime
    dateOfBirth, List<string> interests)
{
  var user = new UserDAO()
  {
    Id = 0,
    EmailAddress = emailAddress,
    FullName = fullName,
    DateOfBirth = dateOfBirth,
    Interests = interests
  };

  dataManager.ModifyUser(user);
}

public void DataManager.ModifyUser(UserDAO user)
{
  var tempId = user.Id;
  if (DBExecutor.ModifyUser(user) <= 0)
  {
    throw new Exception("Operation was not successful.");
  }

  if (user.Id == tempId)
  {
    throw new Exception("The user is invalid.");
  }
}
```

F.3 Source code of the method test in JustDecompile Engine

```
public static bool IsTypeNameInCollisionOnAssemblyLevel(string typeName,
    AssemblySpecificContext assemblyContext, ModuleSpecificContext mainModuleContext)
{
  HashSet<string> usedNamespaces = new HashSet<string>();

  foreach (string usedNamespace in assemblyContext.GetAssemblyNamespaceUsings())
  {
    usedNamespaces.Add(usedNamespace);
  }

  usedNamespaces.UnionWith(mainModuleContext.GetModuleNamespaceUsings());

  List<string> typeCollisionNamespaces;
  if (mainModuleContext.GetCollisionTypesData().TryGetValue(typeName, out
    typeCollisionNamespaces))
  {
    IEnumerable<string> namespacesIntersection = typeCollisionNamespaces.Intersect(
    usedNamespaces);

    if (namespacesIntersection.Count() > 1)
    {
      return true;
    }
  }

  return false;
}
```