Supporting Unit Test Generation via Automated Isolation

Dávid Honfi[1,2], Zoltán Micskei[1]
[1]Budapest University of Technology and Economics
Department of Measurement and Information Systems,
Budapest, Hungary
[2]MTA-BME Lendület Cyber-Physical Systems Research Group,
Budapest, Hungary
{honfi,micskeiz}@mit.bme.hu

October 5, 2016

Abstract

Testing is a significantly time-consuming, yet commonly employed activity to improve the quality of software. Thus, techniques like dynamic symbolic execution were proposed for generating tests only from source code. However, current approaches usually could not create thorough tests for software units with dependencies (e.g. calls to file system or external services). In this paper, we present a novel approach that synthesizes an isolation sandbox, which interacts with the test generator to increase the covered behaviour in the unit under test. The approach automatically transforms the code of the unit under test, and lets the test generator choose values for parameters in the calls to dependencies. The paper presents a prototype implementation that collaborates with the IntelliTest test generator. The automated isolation is evaluated on source code from open-source projects. The results show that the approach can significantly increase the code coverage achieved by the generated tests.

1 Introduction

Nowadays, the demand for *higher quality software* is significantly increasing. Testing is one of the most commonly used techniques to improve the quality of software. During different phases of a software development process testing can be conducted at multiple levels. This paper focuses on *unit testing*, where the goal is to test a well-defined, isolated module commonly called as a unit.

Software testing is a time and resource consuming task and developers face several questions during unit testing [10]. Numerous techniques have been proposed to reduce the time required for unit testing by *automatically generating*

tests using only the source code [7, 23, 1]. These techniques are able to select relevant inputs for the unit under test. *Symbolic execution* is one of the code-based techniques, while *dynamic symbolic execution* (DSE) is a state-of-the-art variant that combines symbolic with concrete executions.

Several tools exist implementing symbolic execution for different programming languages or even binary code. Among several others, KLEE [5], EXE [6], CUTE [31], and DART [13] are designed to be used for C programs. SAGE [14] is a tool for the x86 instruction set. Java PathFinder [26], jCUTE [31] can be used on Java, while IntelliTest (formerly Pex [33]) is available for C#.

An ongoing research area of code based-based test generation techniques (including symbolic execution) is concerning their industrial adoption [35, 3, 36] as the techniques are hindered by numerous already confirmed factors [8, 28, 4]. As a result of these issues, tests generated by dynamic symbolic execution typically achieve low code coverage on complicated source code. Interaction with *dependencies* of the unit is often mentioned among these factors as it may involve accessing the environment (e.g., file system, network) or reaching external modules that are outside the scope of testing. Environment accesses may cause undesired side effects (e.g., creating files), while calling external modules may lead to incorrect test results for the unit under test.

A commonly used technique in unit testing tackling the interaction problem is the *isolation* of calls to the dependencies. Isolation of the unit under test can be performed using *stubs* (returning only a given value) or *mocks* (both returning different values and verifying the interaction). Currently existing isolator frameworks are using two different approaches in terms of implementation: 1) a runtime proxy that detours calls to another objects, or 2) low-level runtime detouring of calls that invoke external modules. Both of them poses a challenge for symbolic execution-based test generation as runtime code intervention is a hindering factor of the technique. Moreover, both implementation approaches have their own limitations of isolating special cases like static or abstract types, which tightens their usage scenarios on source code that is not prepared or designed for testability (e.g., legacy code or complex communicational modules).

Although challenges exists, several attempts were made to enhance test generation on environment-dependent software [34, 22, 2, 32]. For example, the concept of parameterized mock objects [34] is a technique, which collaborates with mocks during test generation. This special type of mocks is designed to obtain return values from the symbolic executor process by adding them as new variables to the path constraint. Using this technique, the test generator is able to select relevant values for the dependencies. This is crucial to cover parts of the unit under test, which rely on return values from dependencies. Parameterized mock objects may solve the problem of dependency interaction in certain cases, still the common limitations of existing isolation approaches and their general collaboration capability with symbolic execution-based test generators leave numerous issues.

The approach presented in this paper addresses the problem of unit isolation for DSE-based test generation by generating a *sandbox*, which interacts with the test generator to increase the covered behaviour in the unit under test.

The approach employs *code transformations* to replace invocations to external dependencies with fake ones with corresponding signatures. These replacement methods include *configurable generated logic* interacting with the test generation process to obtain values to be returned and to alter states of objects passed to the dependency. Moreover, these generated fake methods can be extended with user-provided assumptions restricting the possible behavior of the given dependency. The fake methods together form a *fully parameterized code sandbox* around the unit under test hence making the dependencies explorable for the test generation process. This technique may be employed in scenarios, where dynamic symbolic execution-based test generators usually fail due to the lack of isolation.

We have already presented a preliminary version of our idea in a conference paper [16]. This paper enhances the approach with 1) analyzing possible solutions to the addressed problem in detail, 2) extending the technique with source code transformations, 3) introducing a vastly enhanced implementation and 4) presenting a more thorough evaluation.

Section 2 presents the importance of the unit isolation problem during DSE-based test generation. Then, the main contributions are arranged as follows.

- We give an overview of possible solutions to the problem of unit isolation during dynamic symbolic execution-based test generation (Section 3).
- We introduce a source code transformation approach that may be able to overcome the issues of existing unit isolation approaches and to seamlessly collaborate with test generators. A prototype tool that implements the approach is also presented (Section 4).
- We evaluate the approach and the implemented prototype tool using artificial code samples and modules from open-source projects (Section 5).

2 Background and Motivation

Symbolic execution represents the possible paths of the source code with quantifier-free first order logic formulas over symbolic variables created from program variables. The solution of a path formula (path constraint) provides values for each variable that drive the program execution along the given path. The solution is obtained using constraint solvers that are able to reason over different types of variables.

Classic symbolic execution [21] interprets each statement in a static way, hence the program does not need to be executed. Dynamic symbolic execution (DSE) [30, 8] is an advanced variant that executes the program, while dynamically gathering symbolic constraints over the variables. Notice that DSE requires initial values to start from, which can be simply predefined for each variable type or can be generated randomly. After each concrete execution, the gathered path constraint or a part of it is transformed (e.g., negated) and then solved to be able to steer the concrete execution to a different path. The process is repeated until no more new execution path can be discovered or a predefined boundary criteria (e.g., time, memory) is met.

Example 1. Consider the following example method (Listing 1), where the process of DSE is demonstrated. The method has three different execution paths ending in `return` statements with different values.

Listing 1: Example method for the demonstration of DSE

```
1  public int Example(int a, int b)
2  {
3    if(a > 5)
4    {
5      if(b > 10) { return 1; }
6      return 0;
7    }
8    return -1;
9  }
```

DSE can be started from an arbitrary method, which is `Example` in this case. The technique selects the most simple inputs at first to start a concrete execution. As the two parameters (`a` and `b`) are both integer types, let their assigned values be `0`. Thus, the first concrete execution will execute the path, which ends in statement `return -1`. Along this path, the symbolic execution engine collects the first constraint on the program variables, which is `a > 5`. The DSE engine discovers that if `a := 0` then this path constraint is not satisfied, hence solving this formula may give a new execution path. The solution is calculated by a constraint solver, and a satisfying value is returned. Let this value be `a := 6`, while `b` remains `0`. This executes the body of the first `if` statement as `a > 5` evaluates to true. However, the next statement reached by the execution is `return 0` and a new constraint is added to the path formula (`b > 10`), which has to be satisfied to obtain new execution paths. In the last step, this constraint is solved that gives the value of `11` for variable `b`. Finally, a concrete execution is run with `a := 6` and `b := 11` reaching the only uncovered statement `return 1`. As no more new constraints were collected, thus no new execution paths can be revealed, the DSE algorithm stops and yields the test cases found in Table 1. The last column (Expected result) denotes that the DSE algorithm observed that specific behavior (return value) for the given inputs.

Table 1: Set of generated test cases by DSE for method in Listing 1

#	Value of `a`	Value of `b`	Observed result
1	0	0	-1
2	6	0	0
3	6	11	1

Code-based test generators (including those based on DSE) may alleviate the work of developers and testers by generating an initial set of test cases that can be extended to a whole test suite manually. However, the *testability issues* of the modules may more likely to hinder the test generation process as test generators reach their limitations.

A frequent testability issue is caused by developing a module without considering testing, which prevents testers to inject every external object and con-

figuration into the unit under test. This issue hinders the environment and dependency isolation during testing as there is no possibility to replace original objects to fake ones. Subsequently test generators also usually face several difficulties in these scenarios, as they cannot execute the entire code under test.

Example 2. Let us consider the following example, where a simple method is the unit under test (with two data objects: `FileData` and `FileContentData` that are also included in the unit) in a problematic testing scenario that hinders the work of test generators. The method `GetPermissions(int,byte):int` implements a logic, which decides on permissions of a file. The decision is based on a header indicator in the file, and on results from a permission analysis using another module.

Listing 2: Example method for isolation case

```
1  public int GetPermissions(int fileLength, byte indicator)
2  {
3    Stream file = File.Open(CONFIGLOCATION, FileMode.Open);
4    byte[] fileContent = new byte[fileLength];
5    file.Read(fileContent, 0, fileLength);
6    if (fileContent[0] < indicator)
7    {
8      return -1;
9    }
10   FileContentAnalyzer fca = new FileContentAnalyzer();
11   FileData fd = new FileData();
12   FileContentData fcd = fca.Analyze(fileContent, fd);
13   if(fd.IsReadable && !fcd.IsSecret)
14   {
15     return 1;
16   }
17   return 0;
18 }
```

In this setting, the first challenge that a dynamic symbolic execution-based test generator may face is found in line 3, where the configuration file is opened. If the file does not exist, test generators would always fail here and would not explore remaining parts of the code (**C1**). The issue can be solved via isolating the call or creating the file. During test generation, accesses to the file system should be isolated as unintended behavior may occur. Note that we assume these test generators can seamlessly collaborate with different isolation approaches and frameworks.

The next difficulty, where a test generator may fail is found in line 5, where the stream of the file is read into an array. Assuming the opening of the file is isolated, this call shall be also handled similarly. Otherwise statement `return -1` could not be reached. However, only runtime detouring of the call can be carried out due to the method (unit) structure and design (**C2**). With the help of runtime detouring, statements `return -1` and `return 0` are considered reachable.

Proceeding further one may notice that line 15 may not be executed due to the fact that an external object (`FileContentAnalyzer`) is called. If the type is not implemented yet or contains behavior that may affect test results of the unit (e.g., throws unexpected exceptions), the call to method `Analyze` shall also be

isolated. Moreover, in this specific call, the state of the reference-type argument shall be changed using the properties to reach line 15 (**C3**).

Based on the previous example, some common identified challenges of code-based test generation in a strongly environment-dependent software are the following.

- **C1:** Access to the environment of programs (e.g., file system, network).
- **C2:** Limitations and collaboration capability of isolation approaches.
- **C3:** Change of object states in external invocations.

Supporting test generation even for this simple method may require tremendous effort. Furthermore, we assumed that test generators can collaborate with arbitrary isolation approaches. On the contrary, it is not the fact: their collaboration introduce more issues [17]. This simple example has introduced the main challenges for DSE-based test generation caused by the lack of isolation.

3 Overview of the Supporting Approaches

As presented in Section 1 and 2, invoking dependencies from the unit under test may raise numerous issues when using DSE-based test generation. Thus, their usage on such source code is burdensome. In general, we distinguish four different ways of supporting this test generation technique on software units that possess several external dependencies.

3.1 Using Default Behavior

When using the default behavior of DSE, the test generation process is fully automated. The motivating example presented in Section 2 demonstrated how test generation can fail on various dependencies: access to the file system and memory streams, or invoking methods that are outside the scope of testing. DSE may fail due to these issues as they are included in the general limitations of the technique [8, 28, 4]. Environment dependencies like the file system or low-level library accesses (e.g., FileStream) are hindering the exploration of the code (e.g., when handling files, in certain cases the file shall exist and in some cases shall not). Hence, DSE is unable to collect constraints through some parts of the execution path or even cannot finish a whole path.

Reconsider the example presented in Listing 2, the statements below line 8 were not possible to reach due to the exception occurred in the invoked method if the file is not found or not accessible. Thus, no new symbolic constraint could be collected during the concrete execution.

For example, running the IntelliTest DSE-based test generation tool on this method without any guidance yields the results found in Table 2. Notice that – as the opened file does not exist – only one test case is generated, which shows the hindered behavior of the tool.

Table 2: Results of simply running IntelliTest on Listing 2

#	`fileLength`	`indicator`	Observed result
0	0	0	`FileNotFoundException`

3.2 Guiding Test Generation

Guiding DSE-based test generation can be achieved by employing preconditions (assumptions). These preconditions are included in every path constraint collected during the DSE process, hence every input that is generated must also fulfill these preconditions. The assumptions are written by the user to steer DSE along different, more relevant paths. In case of Listing 2, one can make an assumption on the file location that points to a valid file. For example, the following constraint can be added to every path to handle file locations: `CONFIGLOCATION == "C:\test.txt"`, where `test.txt` is a file preconfigured for testing purposes. Using this guidance, DSE is not hindered by an invalid file access, thus new constraints can be collected throughout the rest of the code.

In the example method of Listing 2, the guidance of the IntelliTest tool can be achieved using a Parameterized Unit Test (PUT) [11], which serves as the starting point of the test generation process. The list of parameters consists of the following variables: `target:PermissionProvider`, `fileLength:int`, `indicator:byte`. We extend this list with the `CONFIGLOCATION` variable in order to assign new values. Furthermore, we make an assumption in the body of the PUT describing that the value of this variable for all generated test cases shall be equal to `"C:\test.txt"`. This specific PUT method containing the mentioned modifications can be found in Listing 3.

Listing 3: PUT with assumption for method found in Listing 2

```
1  public int GetPermissionsTest(
2    [PexAssumeUnderTest]PermissionProvider target,
3    int fileLength, byte indicator,
4    string configLocation // the extra parameter
5  )
6  {
7    // Adding assumption to the configLocation variable
8    PexAssume.AreEqual("C:\\test.txt",configLocation);
9    // Setting the configuration target variable
10   target.CONFIGLOCATION = configLocation;
11   // Calling the method under test
12   target.GetPermissions(fileLength, indicator);
13 }
```

The modification introduced inside the PUT – in order to guide IntelliTest – produces the outcome found in Table 3. The yielded results show that the tool reached the branches in the code where `0` and `-1` is returned. However, the branch where `1` is returned remains uncovered.

Employing assumptions during DSE could alleviate the issues caused by the lack of isolation in certain cases, however several other corner cases exist, where preconditions are not powerful enough. These occur, when the unit under test

Table 3: Results of running IntelliTest with guidance on Listing 2

#	`fileLength`	`indicator`	`configLocation`	Observed result
0	0	0	`"C:\\test.txt"`	`IndexOutOfRangeException`
1	1	0	`"C:\\test.txt"`	0
2	int.MinValue	0	`"C:\\test.txt"`	`OverflowException`
3	1	58	`"C:\\test.txt"`	-1

uses values from external calls and its behavior depends on them. In these cases different isolation approaches and frameworks may provide solutions.

3.3 Approaches for Using Isolation

The commonly known and employed isolation approaches are stubs and mocks. Involving stubs and mocks into the DSE-based test generation process is not a new idea as mentioned in Section 1 (parameterized mock objects). This special type of mocks is able to return inputs necessary to cover parts in the unit that depend on values from external invocations (e.g., content of the file).

Reconsider the example method in Listing 2. The previously defined PUT can be reused by extending its body with a parameterized mock using the Fakes isolation framework. By using this mock, the remaining uncovered branch can also be covered, however this requires manual analysis of the code with scrutinizing its behavior (e.g., the required values for the variables to cover the remaining branch). The Fakes code including the parameterization of the mock is found in Listing 4. The first statement in the body assigns a new `FileData` object for the `data` parameter, while the second statement returns a new `FileContentData` object. Both assignments obtain the objects from the IntelliTest tool by using its `PexChoose.Value` method. The resulting test cases are found in Table 4. The table – compared to Table 3 – is extended with one test case, which executes the path, where the method returns 1. Hence, all of the possible execution paths are covered with using this approach.

Listing 4: Fakes code in the PUT for the method Listing 2

```
1  ShimFileContentAnalyzer // mock for FCA
2    .AllInstances // valid for all instances of FCA
3    .AnalyzeByteArrayFileData = (fca,content,data) => // Replaceing Analyze
4  {
5    data.IsReadable = PexChoose.Value<bool>("data.IsReadable");
6    return PexChoose.Value<FileContentData>("fcdata");
7  };
```

However, employing parameterized mocks during dynamic symbolic execution-based test generation leaves questions open. First, the concept does not deal with state change of objects in an external invocation, which is possible both on the called object itself and on the object-type parameters of the method being called. Second, the creation of mocks rely on isolator approaches that have limited applicability (runtime proxy or detour) as some cases – like native calls – are difficult to handle. Third, one of the most challenging problems is that the

Table 4: Results of running IntelliTest with Fakes on Listing 2

#	`fileLength`	`indicator`	`fd.IsReadable`	Observed result
0	0	0	-	`IndexOutOfRangeException`
1	1	0	`false`	0
2	int.MinValue	0	-	`OverflowException`
3	1	58	-	-1
4	1	0	`true`	1

core ideas of isolator approaches hinder the DSE-based test generation process in general due to the code and call interventions during runtime (as described in Section 2).

3.4 Transforming the Unit Under Test

The previously mentioned challenges (Section 2) with the collaboration of DSE and isolation approaches demands for a new technique, which could alleviate these problems. To overcome the proposed challenges, treating calls in a novel way could provide support to the dynamic symbolic execution-based test generation process. More specifically, replacement of these calls to fake, static methods that have same signatures and contain value generation behavior so that it 1) may not introduce complexity to test generators, along with 2) maintaining functionality of the unit under test. This special procedure on the source requires identifying all external calls and objects. Considering the motivating example (Section 2), these methods are the following: `File.Open`, `FileStream.Read`, `FileContentAnalyzer.ctor`, `FileContentAnalyzer.Analyze`. Moreover, the code also contains two references of external types: `FileStream` and `FileContentAnalyzer`. The replacement procedure involves the following two steps for this method.

1. Rewriting references of external types to a special type, which acts both as a state container and a placeholder, to maintain the syntactical correctness of the code.

2. Replacing every external call to a static invocation into a fake class with same signature.

After conducting these two steps on the source code found in Listing 2, the resulting code of the method is found in Listing 5. Lines 3, 5, 10 and 12 are changed and transformed to isolate external dependencies (marked with color). In line 3, a `DynamicFake` object replaces the original `FileStream` as a state container (see step 1) and the opening of the file is replaced with a call to a static method `FileOpen` in the class `Fake` (see step 2). The reading of `FileStream` is replaced to method `FileStreamRead` (line 5). The instantiation of `FileContentAnalyzer` is transformed to the instantiation of a state container (`DynamicFake`) (line 10) and a call to method `Analyze` is also changed to a fake one (line 12). Argument lists of `FileStreamRead` and `FileContAnalyzerAnalyze` are extended with a

`DynamicFake` that may be able to store the current state of their original container objects dynamically.

Listing 5: Transformed example method for isolation case

```
1  public int GetPermissionFromFileContent(int fileLength, byte indicator)
2  {
3    DynamicFake file = Fake.FileOpen(CONFIGLOCATION, FileMode.Open);
4    byte[] fileContent = new byte[fileLength];
5    Fake.FileStreamRead(fileContent, 0, fileLength, file);
6    if (fileContent[0] < indicator)
7    {
8      return -1;
9    }
10   DynamicFake fca = new DynamicFake();
11   FileData fd = new FileData();
12   FileContentData fcd = Fake.FileContentAnalyzerAnalyze(fileContent,fd,fca);
13   if (fd.IsReadable && !fcd.IsSecret)
14   {
15     return 1;
16   }
17   return 0;
18 }
```

Although the invocations have been replaced, the replacement methods also have to be implemented in the `Fake` static class. Method `FileOpen` shall be able to return a new `DynamicFake` object, method `FileStreamRead` shall be able to fill the `fileContent` byte array with arbitrary content, and finally `FileContentAnalyzerAnalyze` shall be able to set the properties of `FileData` and `FileContentData` to different values. The source code of class `Fake` is found in Listing 6. Note that in the current example, we used a method `ChooseValue<T>` that represents interaction with the test generator to obtain values of a specific type `T`. For example, when using the IntelliTest tool, this can be replaced to method `PexChoose`, which was already presented in the previous sections. In case of array initializations (line 10 and 21), we did not parameterize the size of arrays as it may require preliminary assumptions to avoid unintended overflows.

Listing 6: Example fake container class for replacement methods

```
1  public static class Fake
2  {
3    public DynamicFake FileOpen(string p0, FileMode p1)
4    {
5      return new DynamicFake(); // Returning a state container
6    }
7
8    public int FileStreamRead(byte[] p0, int p1, int p2, DynamicFake obj)
9    {
10     p0 = new byte[2]; // Assigning a new array to p0
11     for (int i = 0; i < p0.Length; i++)
12     {
13       // Filling p0 with arbitrary values
14       p0[i] = ChooseValue<byte>("fsr-p0-"+i);
15     }
16     return ChooseValue<int>("fsr-ret"); // Choosing arbitrary int to return
17   }
18
19   public FileContentData FileContentAnalyzerAnalyze(byte[] p0, FileData p1,
         DynamicFake obj)
```

```
20    {
21      p0 = new byte[2]; // Assigning a new array to p0
22      for (int i = 0; i < p0.Length; i++)
23      {
24        // Filling p0 with arbitrary values
25        p0[i] = ChooseValue<byte>("fcaa-p0-"+i);
26      }
27      // Setting a property of p1
28      p1.IsReadable = ChooseValue<bool>("fcaa-p1-IsReadable");
29      // Returning a new FileContentData object
30      return ChooseValue<FileContentData>("fcaa-ret");
31    }
32  }
```

Using this fake method container class in combination with the special transformation of the unit under test, a dynamic symbolic execution-based test generation process is alleviated from the issues caused by external dependencies (**C1**, **C2** and **C3** in Section 2). Thus, a white-box test suite could be generated easily to cover the unit under test with additional information about the dependencies. This data describes which behavior (return value, state change of parameters) steers the program executions along different paths.

Table 5: Possible set of generated inputs for the transformed example method

indicator	fcaa-p1-IsReadable	fcaa-ret	**result**
00	false	null	0
00	true	null	-
00	true	`new FileContentData(IsSecret=false)`	1
01	-	-	-1

Implementing this approach for IntelliTest provides the generated set of test inputs found in Table 5, which covers every execution path in the method under test. We use notations of the variables from Listings 5 and 6 (see assigned parameters of method `ChooseValue`). Note that the table only contains variables that needed to have different values for the test generation process. The constant assignments for the other variables are the following: `fileLength = 6`, `fsr-p0-0 = fsr-p0-1 = 00`, `fsr-ret = 0`, `fcaa-p0-0 = fcaa-p0-1 = 00`.

The proposed procedure has three main steps: 1) static code analysis, 2) code transformation and 3) sandbox generation. All of them can be automated using special algorithms and techniques. Static code analysis is viable using code traversal algorithms to identify external types and invocations. The rewriting of the source code can be achieved using specific transformations to replace the parts identified during the analysis step. Finally, the synthesization of the sandbox around the unit under test can be accomplished using code generation techniques. In Section 4, we present this automated isolation approach in detail.

4 Approach for Automated Isolation

The approach presented in this paper tackles the unit isolation problem for DSE-based test generation using syntax transformations and sandbox code synthesis.

Our technique uses Abstract Syntax Trees (ASTs) [20] to gain information and to modify the source code. ASTs are graphs representing parts of source code, which can be obtained via code parsing. The nodes of the tree denote different structures taking place in the code, hence types of the nodes are depending on the grammar of the programming language being used.

4.1 Generic Definition

Our isolation approach involves three main steps: 1) analysis of the code under test, 2) syntax transformation of the unit and 3) synthesis of an isolation sandbox. The overview of the whole approach can be found in Algorithm 1. The process starts from a predefined unit, which is given with the fully qualified names of elements to include. An element can be a method, a class or even a whole module. The concept of a unit can be formalized as found in Definition 1.

Algorithm 1 High-level algorithm of the presented approach

```
1: function AutomatedIsolator(Unit[] units)
2:     for all unit in units do                          ▷ iterating through units
3:         ast := parseAst(unit.getSource());            ▷ getting AST of unit
4:         syntaxData := analyzeSyntax(ast);             ▷ analyzing syntax tree
5:         newAst := transformAst(ast,syntaxData);       ▷ transforming the AST
6:         sandbox := synthesizeSandbox(syntaxData);     ▷ creating the sandbox
7:         outputCode(newAst, sandbox);                  ▷ emitting the results
8:     end for
9: end function
```

Definition 1 (Unit Under Test)**.** *Let the unit under test UUT be a set so that an element $u \in UUT$ is an arbitrary module of the software, which can be identified by its fully qualified name.*

For example, `MySoftware` has three modules (`Module1`, `Module2`, `Module3`), then a possible unit under test is $UUT = \{\texttt{MySoftware.Module1}, \texttt{MySoftware.Module2}\}$. In that context, `Module3` is thought as external during unit testing. Note that every ancestor of classes included in the unit are also automatically added to the unit to avoid issues caused by rewriting external types in signatures.

4.1.1 Syntax Analysis

The tasks during the analysis are 1) to reveal invocations of methods thought as external from the unit and 2) to identify references to types thought as external. The detection is performed using the ASTs and the attached semantic models that are obtained from runtime compilation of the source code under test. The semantic model contains information about the types used in the source. The AST is traversed and every node *Call* is scrutinized in detail that matches the following two constraints at once.

Definition 2 (Call node)**.** *$Call(s, c)$ is an AST node with signature s (e.g., its name, parameters, etc.) and container c so that $Call$ is a method invocation or member access expression, and c is an external module: $c \notin UUT$.*

Deciding if the invoked method or accessed member is external or not is achieved via type analysis using the semantic model, which can be used to obtain fully qualified names for the elements (e.g., variables, methods) in the AST. Note that basic types or specific primitive or system types shall be included in the unit by default to avoid overisolation (e.g, isolating integers).

Furthermore, in order to detect external type usages, every node Typ is collected for further analysis that satisfy the following two constraints.

Definition 3 (Typ node)**.** *$Typ(t)$ is an AST node so that it is a parameter of type t or a return type of t or a variable declaration expression of type t. Furthermore, t is an external type: $t \notin UUT$.*

Before any other step could be taken, external method invocations shall be analyzed more deeply to discover their signature, which can be used in the body of the replacement method. In there, the actual state of different parameter objects can be altered possibly simulating the original behavior. Hence, during the analysis of parameters, variables with types included in the unit are sought ($t, Typ(t) \in UUT$). However, changing the state of these objects requires further and more deeper examination.

Using the semantic model, parameters of external invocations are analyzed that have types included in the unit. The type analysis discovers members (e.g., fields) of the object, which can be modified from any other object (i.e., it is public and writable from outside). This process is performed recursively as several levels of references among types may exist.

All the information collected during the analysis of invocations, variables and types is stored for use in the forthcoming steps of the automated isolation process.

Example 3. Let us consider the example method found in Listing 7 in order to demonstrate the workflow of syntax and type analysis. The method indicates if the weekend is near by returning true if the day after tomorrow is Saturday and false otherwise. Let the under under test be only this method, thus $UUT = \{$`WeekendNotifier.IsWeekendNear():bool`$\}$.

Listing 7: Example method to the demonstration of isolation workflow

```
1  public class WeekendNotifier {
2    public bool IsWeekendNear()
3    {
4      DateTime date = DateTime.GetNow();
5      date.AddDays(2);
6      if(date.GetDay() == "Saturday")
7      {
8        return true;
9      }
10     return false;
11   }
12 }
```

The analysis starts with parsing the source code into an AST. A simplified version of the AST for this example method is found in Figure 1. The first step discovers external invocations, which we marked with light gray on the AST. The signature and type information of all the three invocations are collected and persisted for later use. The next step during the analysis phase is to identify external type usages. In this case, there is only one variable declaration, which uses an external type (marked with dark gray on the AST).

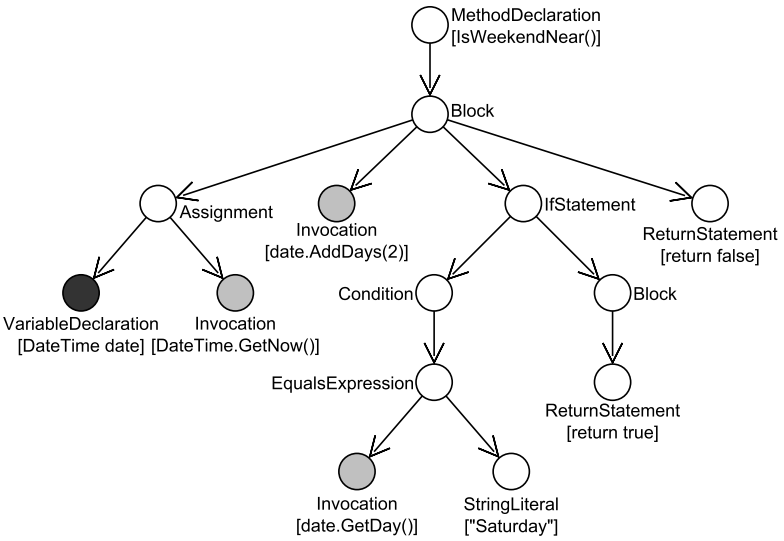

Figure 1: The simplified AST of method IsWeekendNear

4.1.2 Syntax Tree Transformation

In order the replace the invocations and type usages detected during the analysis step, the AST is transformed for each detected node ($Call$ or Typ). The approach rewrites 1) method and constructor invocations, 2) member accesses and 3) type usages. The rewriting algorithm conducts the following transformations on the previously collected nodes ($Call$ or Typ).

- $Call(s, c) \rightarrow Call(s', f)$, where $Call(s', f)$ denotes a method invocation with similar s' signature in container f that stands for `Fake`. s' is a slightly modified (s' has the method name combined from the unique name of the containing type and the original method name), and possibly extended variant of s signature if the method call is not static. In this case, the list of parameters is extended with a `DynamicFake` parameter to maintain the state of the external object.
- $Typ(t) \rightarrow Typ(df)$, where df denotes the type `DynamicFake`.

Let us reconsider the example method used during the demonstration of the analysis step. All the required information in method `IsWeekendNear` was collected regarding the external invocations and type usages. We demonstrate the AST transformation through this simple method to give a better understanding (Figure 2). The modified nodes of the AST have bold labels. First, the invocations are transformed in order to invoke replacement methods instead of the original ones. Note the invocation of method `AddDays`: the containing type and original method name is combined (`DateTimeAddDays`) for unique identification and the list of parameters is extended with a new variable of `DynamicFake` type representing the state of the external object. Note that if there were multiple types named `DateTime`, then the approach would use the fully qualified name of the type (e.g., `SystemDateTime`, `OtherDateTime`). Moreover, if method names are colliding, the list of parameter types are also added to their names to ensure uniqueness. The other two invocations were also transformed, however method `DateTime.GetNow` is static, thus the list of parameters is not extended there. Furthermore, the only one external type usage node (VariableDeclaration) is transformed to use the `DynamicFake` type instead of the original `DateTime`.

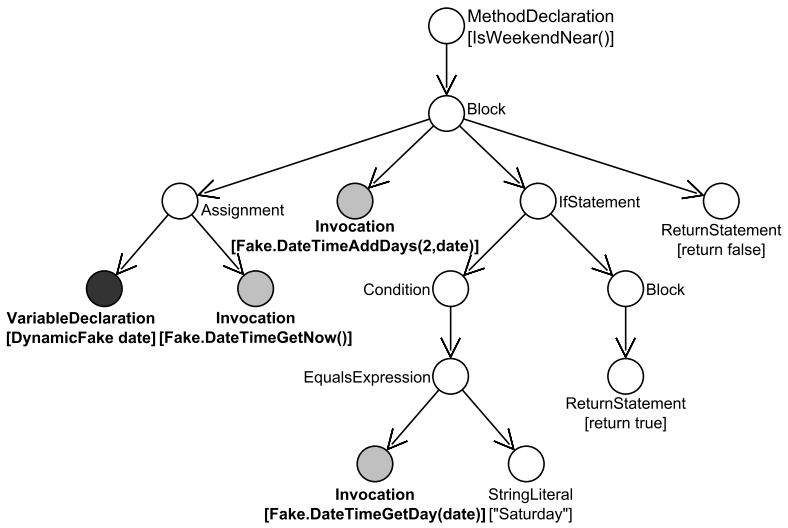

Figure 2: The transformed AST of method IsWeekendNear

A question may be raised about the modified code in the unit under test if it influences or alters the behavior or not. In a wrongly designed source code with numerous external dependencies, isolating these external calls is hindered by the lack of injection possibility. This causes the modification unavoidable for the code under test. Manual testability refactoring [15] techniques mostly employ approaches, which also introduce modifications to the source code, like extracting interfaces [24]. Although these may provide solutions for testability

issues, manual refactoring may require tremendous effort in large and complex software. Our code transformation 1) does not modify the behavior of the unit under test and 2) does not require manual effort as it is performed automatically.

4.1.3 Sandbox Synthesis

As the original source code is rewritten to invoke replacement methods, the container for the definitions needs to be created. The replacement calls are acting as static calls hence the container itself is also a static class called `Fake`.

`Fake` contains all the m methods that were invoked in the unit under test and thought as external ($m \notin UUT$). If the method m has signature s then the method in the container has signature s' corresponding to the invocation in the transformed AST. These methods are not only need to be defined in the container `Fake`, they should also define arbitrary and extensible behavior to simulate the original.

Numerous possibilities exist to simulate the original behavior of external components. Our technique currently defines two different behaviors.

- *Simple behavior:* If the method does return any value, it will act as a *stub* and will not define any logic or behavior. If it does return a value of some type, then it is obtained from the DSE-based test generator by adding the variable to the path constraint.
- *Advanced behavior:* Extends the simple behavior with object state handling. Hence, if a parameter has a type, which is included in the unit, then all of its recursively discovered modifiable members has an assignment in the body of the method. The assignments obtain values for the members from the test generation process by adding them as variables to the path constraint.

Let us reconsider the method `IsWeekendNear` and its transformed AST (Figure 2) used during the previous examples. The synthesized sandbox (`Fake`) from the data collected during the analysis contains the definitions of the three methods that are invoked in the transformed unit under test. Note that there is no parameter, which state can be altered in the body of the replacement method. The synthesized sandbox code for the `IsWeekendNear` example can be found in Listing 8.

Listing 8: Sandbox for the example method IsWeekendNear

```
1  public static class Fake
2  {
3    public DynamicFake DateTimeGetNow()
4    {
5      // Return a state storage object instead of the original
6      return new DynamicFake();
7    }
8
9    public void DateTimeAddDays(int days)
10   {
11     // A simple test stub
12   }
```

```
13
14    public int DateTimeGetDay()
15    {
16      // Getting value from test generator
17      return ChooseValue<int>();
18    }
19  }
```

4.2 Possible use cases

The presented approach is suitable for two different possible use cases as it is currently tied to the dynamic symbolic execution technique. Furthermore, it is designed to alleviate the challenges of a code-based test generator (**C1**, **C2** and **C3** of Section 2) in a strongly environment-dependent software.

The first and main use case of the approach is the support of the DSE-based test generation by alleviating the isolation problem that may hinder the process. By replacing the external invocations and type usages, the unit under test is isolated from everything thought as external and could be run in a parameterized sandbox filled by DSE. The test cases generated for the transformed source code may reveal problems in the original code as it only focuses on the behavior of the unit under test in a simulated environment altered by DSE.

The other use case is for the integration of modules. By using the presented approach, one can analyze the influence of external invocations in the unit under test. This includes checking the possible interactions and their values to decide if the module is ready for integration with others. Also, one can describe and restrict the generated behavior of external invocations in order to have more realistic simulation of the original dependencies. These descriptions can be achieved via using assumptions for the test generator.

4.3 Tool-specific Implementation

The implementation of the approach requires a code-based test generator that uses dynamic symbolic execution to create tests. Moreover, the technique employs a special abstract syntax tree transformation, which demands for a source code parser and transformer that enables the definition of custom transformation rules.

We selected IntelliTest as the DSE-based test generator for the implementation. It is one of the most advanced tools available and its transfer to industrial practice was already investigated indicating its maturity [35]. As IntelliTest currently only works with source code written in C# language, the number of possibilities to choose a source code parser was reduced. One approach could have been to write our own C#-to-AST parser, however the .NET developer team provides a code analysis library, called Roslyn [25]. This library is able to construct ASTs from C# source code and also supports the transformation of the trees, which makes it a suitable tool for the requirements of our approach.

We implemented the whole approach in a tool, which is an extension of the Visual Studio integrated development environment. The user first defines the

elements of the unit under test using their fully qualified names. Then, when invoking the tool the ASTs of the unit elements are transformed, and the code of the sandbox (`Fake`) – containing the implementations of the replacement methods – is also generated. The bodies of the replacement methods contain statements that collaborate with IntelliTest by obtaining concrete values from the tool. Note that the current implementation does not support the state container feature of `DynamicFake` objects.

5 Experimental Evaluation

As the approach has been implemented in a proof-of-concept prototype tool, the preliminary experimental evaluation of the approach became feasible. We employed two types of source code in this experiment: 1) snippets from an evaluation framework and 2) parts of open-source projects from GitHub.

5.1 Objective

This evaluation intends to decide whether the approach and the implemented prototype tool is able to support DSE-based test generation process. Hence, the current experiment aims to answer the following research question.

Is the automated isolation approach able to enhance block coverage for DSE-based test generation?

5.2 Process

Answering the RQ requires collecting software modules that are implementing different behaviors with diverse logic constructs. The subjects of evaluation can be obtained from various places like open-source code repositories, where C# projects can be found (e.g., CodePlex or GitHub). We chose GitHub as the sources of the projects.

First, we used environment-dependent snippets from SETTE [9] (Symbolic Execution-based Testing Tool Evaluator), a framework specially created to compare test generator tools. The SETTE snippets were translated manually from Java to C#.

Next, we randomly chose projects available on GitHub meeting a predefined set of criteria. We defined the following selection criteria.

- The project repository shall have at least 100 stars indicating its popularity, thus the mature state of the code.
- The project repository shall have been updated in the last five days, which indicates the active development.
- The project shall be compiled and built with one-click in order to speed up the evaluation process.

- The selected modules of a project shall not contain code with multi-threading as it cannot be handled by IntelliTest.

The answer requires measurements of the yielded results in terms of coverage. We measured the basic code block coverage of simply running IntelliTest on the analyzed module first. Then, the transformation was applied to the code using the automated isolation approach, and IntelliTest was executed again. Note that we did not provide any support manually (e.g., factory methods, assumptions, etc.) for IntelliTest, or for the automated isolation tool either.

5.3 Setup

The selection process of the projects from GitHub was very simple. We searched for repositories that have more than 100 stars and had been updated in the last five days. Then, we selected an arbitrary project from the list that contained no multi-threaded logic, then we tried to compile it. If the compilation finished successfully, the project was included in the evaluation, otherwise we looked for another repository that matched our criteria. Using this procedure, we managed to select the following projects and modules for this preliminary evaluation. The class selection criteria was to choose classes that act crucial roles in the business logic. From SETTE, we selected three classes implementing various behaviors with environment interaction. The detailed statistics of the selected modules are found in Table 6. We used these classes as the unit under tests (UUT).

- *Abot* is customizable and lightweight web crawler. Class `WebContentExtractor` obtains the required content from the currently crawled site. `RobotsDotTextFinder` is responsible for seeking the robots.txt file, which is a de facto standard for describing the intended behavior of web crawlers for the website. `CrawlDecisionMaker` decides on the behavior, when a possible branch occurs in the crawled website path.

- *Textc* is a natural language processing library. Command Syntax Definition Language is an included notation, which can be used to define syntaxes. Syntaxes form the basis of processing as they are matched against the input tokens. Class `SyntaxParser` implements the default behavior of parsing an arbitrary text with a selected syntax. The result is information about an expression that was parsed from an input text using the specified syntax. `CsdlToken` represents a token in the syntax description language. Class `CsdlParser` can parse texts written in CSDL to define new syntaxes.

- *LiteDB* is a NoSQL document store that uses only a single file for storage. The application is lightweight and is rich of features. Class `LiteFileStorage` is a collection to store files or data streams. `DataService` provides basic CRUD methods to create, read, update and delete arbitrary data in the document store. Finally, `TransactionService` is responsible for managing transactions.

- *Papercut* is an SMTP e-mail receiver. The application is useful for testing applications that are sending e-mails. Class `MessageRepository` is responsible for managing the storage of incoming messages. `NetworkHelper` is a class, which provides methods for frequently used actions in networking (e.g., getting the IP address). `TempDirectoryCleanupService` is a service, which cleans the temporary storage directory, when a specific event occurs.

- *SETTE* is a symbolic execution-based testing tool evaluator framework implemented in Java. We translated three environment-dependent classes to C# for this evaluation. Class `SetteFileIo` performs various file operations like writing and reading. `SetteNetworking` implements a complex networking behavior including a server that processes requests from a client. `SetteStdio` uses the standard input and output for various operations.

Table 6: Details of the selected classes for evaluation

Project	Class	Method count	Lines of code
Abot	`WebContentExtractor`	1	63
	`RobotsDotTextFinder`	1	15
	`CrawlDecisionMaker`	4	68
Textc	`SyntaxParser`	1	35
	`CsdlToken`	6	101
	`CsdlParser`	5	53
LiteDB	`LiteFileStorage`	14	57
	`DataService`	8	80
	`TransactionService`	5	38
Papercut	`MessageRepository`	5	38
	`NetworkHelper`	4	29
	`TempDirectoryCleanupService`	3	17
SETTE	`SetteFileIo`	3	44
	`SetteNetworking`	1	41
	`SetteStdio`	4	36

Two steps were taken to obtain the results: 1) we executed IntelliTest and obtained the coverage results, then 2) the automated isolation tool was executed to transform the code and to create a sandbox, and IntelliTest was run once again to obtain the new coverage metrics. Each execution was repeated 3 times. During the evaluation, we observed no differences among the outcomes of repetitions.

The evaluation was performed on a laptop running Windows 10 and Visual Studio 2015 Enterprise Update 3.

5.4 Results

The results obtained can be found in Table 7. Here, GT denotes the number of generated tests, while BC indicates the block coverage reached.

Table 7: Detailed results of the evaluation.

		IntelliTest default		Automated isolation	
Project	Class	GT	BC	GT	BC
Abot	`WebContentExtractor`	1	7,87%	17	93,59%
	`RobotsDotTextFinder`	1	29,03%	5	92,59%
	`CrawlDecisionMaker`	17	30,23%	61	95,82%
Textc	`SyntaxParser`	24	91,07%	28	100,00%
	`CsdlToken`	61	46,54%	88	58,29%
	`CsdlParser`	41	54,62%	7	35,96%
LiteDB	`LiteFileStorage`	22	55,28%	51	100,00%
	`DataService`	21	16,08%	20	30,61%
	`TransactionService`	5	25,37%	6	33,90%
Papercut	`MessageRepository`	5	16,22%	37	90,28%
	`NetworkHelper`	4	81,67%	10	81,67%
	`TempDirectoryCleanupService`	3	62,5%	11	91,67%
SETTE	`SetteFileIo`	7	69,35%	21	100,00%
	`SetteNetworking`	1	23,08%	3	92,31%
	`SetteStdio`	25	100,00%	12	100,00%

The first project we employed was *Abot*. The results for project Abot show significant increase in the number of generated tests and block coverage in all three cases. The block coverage reaches more than 90%, which can be thought as successful.

Comparing the results of Textc to the previous project, the initial number of generated tests are significantly larger, meaning that IntelliTest could more easily handle these classes. When the automated isolation was applied, the number of generated tests and also the block coverage increased in the first two classes. Note that there were blocks in class `CsdlToken`, which were not covered even in the transformed code. This issue is related to IntelliTest as it could not instantiate a map object with specific elements that is required to execute different branches. Also, class `CsdlParser` shows interesting results: the coverage decreased after applying automated isolation. This anomaly was due to the following two root causes: 1) current implementation of the `Fake` container does not support returning arbitrary sizes of arrays of given types, 2) a type query (`typeof`) statement could not be transformed in the AST. The combination of these issues led to missing a whole method and multiple other blocks to cover. It must be emphasized here that this issue is only related to the current state of the prototype tool and not to the approach.

In terms of project LiteDB, the yielded coverage and test metrics show sim-

ilar results to Abot: a significant increase is seen for all three classes in both metrics. In case of the last two classes, there are code blocks, which were not covered even in the transformed code, which caused the coverage metric to stay very low in both cases. After scrutinizing the generated test cases, we found two different reasons. For class `DataService`, the blocking issue was that IntelliTest could not provide a map for the `Fake` container that is indexable with different values. In case of class `TransactionService`, reaching 100% coverage would require construction of method invocation sequences (e.g., begin, commit, save). Thus, these issues are not related to the approach or the prototype tool.

The results for project Papercut show similar increase of both examined metrics than in the previous three projects. For class `NetworkHelper`, the coverage did not show any growth, however the number of generated tests was raised from 4 to 10. This was caused by a tool-related issue, particularly the lack of transforming a special structure in the code (`using`). This feature is currently not implemented in the tool.

Finally, identical results can be discovered for project SETTE. For classes `SetteFileIo` and `SetteNetworking`, the coverage increased significantly. In the latter class, the cause of omitting full block coverage is that the server-side code contains an infinite loop, which can be only stopped by thread handling. Thus, IntelliTest is not able to reach some statements after the loop. For class `SetteStdio`, the block coverage remained 100% in both cases, however the number of generated tests is reduced as the unit to explore is smaller when using isolation.

In summary, the results for the five projects showed a clear increase in the number of generated tests and also in block coverage, when using IntelliTest supported by the automated isolation approach. Based on these results, the approach could be able to help increasing the coverage for tests generated by DSE.

5.5 Limitations

During the preliminary evaluation, we managed to identify bugs and issues hindering our approach, though they were caused only by the prototype state of the tool, and were not related to the approach itself. One of these issues was the lack of transformations of some special structures. These AST transformations require numerous different scenarios to be prepared for as the C# language is very vast. Among others, we found the following issues with the tool when it was executed on the selected classes. Also note that most of these were *not fixed* – due to the complexity – during the evaluation and the results for the RQ could be influenced by them.

- The `DynamicFake` state container objects are currently acting as a dummy type and are not storing anything.
- Class `DynamicFake` is not disposable and not enumerable meaning that it cannot be used in some special code contexts.

- Enums are thought as external types in some cases, however they are not harmful as they have no behavior. Thus, should not act as subjects of isolation.
- Properties of class `Fake` have no generic type indicators in certain situations.
- Methods in the `Fake` container are not able to throw exceptions in their current implementation state.
- Generic methods and types are not handled properly in the AST transformations in certain cases.
- Casting to an external type is not transformed in the AST.
- Delegates and anonymous methods are not handled during the AST transformation.
- `Using` structures are currently not supported in the AST transformations.
- Classes in the unit that implement external interfaces or external abstract classes are currently not supported due to the large amount of transformations required.

Some of these issues caused the compiling of the transformed unit under test to fail. During the evaluation, we only fixed the blocking issues manually. We decided to not fix the rest of the issues for this paper as 1) the results of the prototype version was already able to show the potential in the underlying approach, and 2) designing and implementing the missing features would require significant effort.

6 Related work

Our idea originally derives from a paper written by Tillmann *et al.* [34], where the idea of mock object generation is described. They also conducted a case study for file-system dependent software [22], that showed promising results for using parameterized mocks. Their presented technique is able to automatically create mock objects with behavior and has the ability to return symbolic variables, which is used during the symbolic execution to increase the coverage of the unit under test. However, their solution requires external interfaces explicitly added to the parameterized unit tests (i.e. needs user intervention), moreover they did not consider any state change of object inside mocks that can affect the coverage in the unit under test. Hence, our solution covers a wider area of scenarios and needs minimal user interaction for the automated generation (our approach only requires the fully qualified names of the units under test).

The idea of Galler *et al.* is to generate mock objects from predefined design by contract specifications [12]. These contracts describe preconditions of a method, thus the derived mocks are created in respect of them. This makes the

mocks able to avoid false behavior. However, their approach does not relate to dynamic symbolic execution and provides no mention of collaboration with any test generation process. The approach may also introduce work overhead when creating contracts as specification.

Samimi *et al.* proposed the approach of declarative mocking [29]. Their technique requires developers to write specifications in a domain-specific language (DSL) to describe the intended behavior of the method to mock. The specification is then executed by a special tool called PBnJ. Hence, this approach needs developers to write their own tests, and it has also no mention of collaboration with code-based test generator approaches and tools.

A similar approach is introduced in parallel with a symbolic execution engine to Java by Islam *et al.* [18]. The difference with previous two techniques is that this one uses interfaces as specifications instead of contracts or a special DSL.

Another approach of mock generation was presented by Pasternak *et al.* [27]. 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 our presented approach uses only the previous compilation with static AST transformations.

A model-based approach of isolation is presented by Jeon *et al.* [19] for Java programs that largely rely on frameworks. Their technique derives a framework model in order to support and collaborate with symbolic execution during the test generation process. Their implemented tool PASKET is able to instantiate a model from code artifacts and tutorial programs, which has a matching behavior with the original framework.

7 Conclusion and future work

In this paper, an approach for automatically isolating external dependencies has been presented to support dynamic symbolic execution-based (DSE) test generation in complex software. This technique is designed to collaborate with DSE-based test generation by obtaining values for dependencies directly form the test generator tool. These values are used as return values for external methods and for assignments, where changing states of object is possible inside the dependency. The presented approach replaces the calls to external dependencies with fake invocations to a sandbox. The extensible sandbox is synthesized from various information collected during code and type analysis. This sandbox is able to collaborate with dynamic symbolic execution.

The paper also presented a prototype tool that implements the approach for C# by using Roslyn and is able to collaborate with IntelliTest, a state-of-the-art dynamic symbolic execution-based test generator. The technique was evaluated in terms of increase in code coverage of the generated tests. The preliminary evaluation employed snippets from a symbolic execution-based tool evaluator framework, and modules from selected open-source projects on GitHub. The results of the evaluation were promising as the prototype tool was able to trans-

form these modules, and was also capable of synthesizing a sandbox without serious blocking issues. Furthermore, the transformed code that used the sandbox had significantly higher code coverage with increased number of generated tests.

In terms of future work, our plan is to elaborate the use of `DynamicFake` objects that are able to store and maintain the state of externally-typed objects. This would improve the whole approach and may provide better matching with software system environments. We are also continuously fixing the issues of the prototype tool. Another way of improvement could be to enhance the behavioral logic inside the sandbox from sample programs or observations of concrete executions (similarly to [19] and [27]). Furthermore, we would like to provide an incremental isolation refinement method for our current approach in order to avoid isolating calls that are not necessary for improving DSE-based test generation.

References

[1] A. Arcuri, M. Z. Iqbal, and L. Briand. Random testing: Theoretical results and practical implications. *Software Engineering, IEEE Transactions on*, 38(2):258–277, 2012. doi: 10.1109/TSE.2011.121.

[2] A. Arcuri, G. Fraser, and J. P. Galeotti. Automated unit test generation for classes with environment dependencies. In *Proc. of the Int. Conf. on Automated Software Engineering*, pages 79–90. ACM, 2014. doi: 10.1145/2642937.2642986.

[3] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel symbolic execution for automated real-world software testing. In *Proc. of the Int. Conf. on Computer systems*, pages 183–198. ACM, 2011. doi: 10.1145/1966445.1966463.

[4] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013. doi: 10.1145/2408776.2408795.

[5] C. Cadar, D. Dunbar, and D. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. of Operating systems design and implementation*, OSDI'08, pages 209–224. USENIX Association, 2008.

[6] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):10, 2008. doi: 10.1145/1180405.1180445.

[7] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice:

preliminary assessment. In *Proc. of the Int. Conf. on Software Engineering (ICSE)*, pages 1066–1071. ACM, 2011. doi: 10.1145/1985793.1985995.

[8] T. Chen, X. song Zhang, S. ze Guo, H. yuan Li, and Y. Wu. State of the art: Dynamic symbolic execution for automated test generation. *Future Generation Computer Systems*, 29(7):1758 – 1773, 2013. ISSN 0167-739X. doi: 10.1016/j.future.2012.02.006.

[9] L. Cseppento and Z. Micskei. Evaluating symbolic execution-based test tools. In *Proc. of Int. Conf. on Software Testing, Verification and Validation (ICST)*, pages 1–10. IEEE, 2015. doi: 10.1109/ICST.2015.7102587.

[10] E. Daka and G. Fraser. A Survey on Unit Testing Practices and Problems. In *Proc. of Int. Symp. on Software Reliability Engineering*, pages 201–211, Nov 2014. doi: 10.1109/ISSRE.2014.11.

[11] J. de Halleux and N. Tillmann. Parameterized Unit Testing with Pex. In B. Beckert and R. Hähnle, editors, *TAP*, volume 4966 of *LNCS*, pages 171–181. Springer, 2008. ISBN 978-3-540-79123-2. doi: 10.1007/978-3-540-79124-9_12.

[12] S. J. Galler, A. Maller, and F. Wotawa. Automatically Extracting Mock Object Behavior from Design by Contract$^{\text{TM}}$ Specification for Test Data Generation. In *Proc. of the Workshop on Automation of Software Test (AST)*, pages 43–50. ACM, 2010. doi: 10.1145/1808266.1808273.

[13] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. *ACM Sigplan Notices*, 40(6):213–223, 2005. doi: 10.1145/1064978.1065036.

[14] P. Godefroid, M. Y. Levin, D. A. Molnar, et al. SAGE: Automated Whitebox Fuzz Testing. In *NDSS*, volume 8, pages 151–166, 2008. doi: 10.1145/2090147.2094081.

[15] M. Harman. Refactoring as testability transformation. In *Proc. of the Int. Conf. on Software Testing, Verification and Validation Workshops*, ICSTW '11, pages 414–421, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4345-1. doi: 10.1109/ICSTW.2011.38.

[16] D. Honfi and Z. Micskei. Generating unit isolation environment using symbolic execution. In *Proceedings of the 23rd PhD Mini-Symposium*. Budapest University of Technology and Economics, Department of Measurement and Information Systems, Budapest University of Technology and Economics, Department of Measurement and Information Systems, 2016. ISBN 978-963-313-220-3.

[17] D. Honfi, Z. Micskei, and A. Vörös. Isolation and Pex: Case Study of Cooperation, 2013. Technical report, Budapest University of Technology and Economics.

[18] M. Islam and C. Csallner. Dsc+Mock: A Test Case + Mock Class Generator in Support of Coding against Interfaces. In *Proc. of the 8th Int. Workshop on Dynamic Analysis*, pages 26–31. ACM, 2010. doi: 10.1145/1868321.1868326.

[19] J. Jeon, X. Qiu, J. Fetter-Degges, J. S. Foster, and A. Solar-Lezama. Synthesizing Framework Models for Symbolic Execution. In *Proc. of the Int. Conf. on Software Engineering*, ICSE '16, pages 156–167, New York, NY, USA, 2016. ACM. doi: 10.1145/2884781.2884856.

[20] J. Jones. Abstract Syntax Tree Implementation Idioms. In *Proc. of the Int. Conf. on Pattern Languages of Programs (PLOP 2013)*, pages 1–10, 2003.

[21] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976. doi: 10.1145/360248.360252.

[22] M. R. Marri, T. Xie, N. Tillmann, J. De Halleux, and W. Schulte. An Empirical Study of Testing File-System-Dependent Software with Mock Objects. In *Proc. of the Workshop on Automation of Software Test (AST)*, pages 149–153, 2009. doi: 10.1109/IWAST.2009.5069054.

[23] P. McMinn. Search-based software testing: Past, present and future. In *Proc. of Int. Conf. on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 153–163. IEEE, 2011. doi: 10.1109/ICSTW.2011.100.

[24] G. Meszaros. *XUnit Test Patterns: Refactoring Test Code*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006. ISBN 0131495054.

[25] Microsoft. .NET Compiler Platform ("Roslyn"), 2016. https://github.com/dotnet/roslyn.

[26] C. S. Păsăreanu and N. Rungta. Symbolic PathFinder: symbolic execution of Java bytecode. In *Proc. of the Int. Conf. on Automated Software Engineering (ASE)*, pages 179–180. ACM, 2010. doi: 10.1145/1858996.1859035.

[27] B. Pasternak, S. Tyszberowicz, and A. Yehudai. GenUTest: a Unit Test and Mock Aspect Generation Tool. *Int. Journal on STTT*, 11(4):273–290, 2009. doi: 10.1007/s10009-009-0115-4.

[28] X. Qu and B. Robinson. A case study of concolic testing tools and their limitations. In *Proc. of Int. Symp. on Empirical Software Engineering and Measurement (ESEM)*, pages 117–126. IEEE, 2011. doi: 10.1109/ESEM.2011.20.

[29] H. Samimi, R. Hicks, A. Fogel, and T. Millstein. Declarative mocking. In *Proc. of the Int. Symposium on Software Testing and Analysis*, ISSTA 2013, pages 246–256, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2159-4. doi: 10.1145/2483760.2483790.

[30] K. Sen. Concolic testing. In *Proc. of the Int. Conf. on Automated Software Engineering*, pages 571–572. ACM, 2007. doi: 10.1145/1321631.1321746.

[31] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *Computer Aided Verification*, volume 4144 of *LNCS*, pages 419–423. Springer, 2006. doi: 10.1007/11817963_38.

[32] K. Taneja, Y. Zhang, and T. Xie. MODA: Automated test generation for database applications via mock objects. In *Proc. of the Int. Conf. on Automated Software Engineering*, pages 289–292. ACM, 2010. doi: 10.1145/1858996.1859053.

[33] N. Tillmann and J. de Halleux. Pex–White Box Test Generation for .NET. In B. Beckert and R. Hähnle, editors, *TAP*, volume 4966 of *LNCS*, pages 134–153. Springer, 2008. ISBN 978-3-540-79123-2. doi: 10.1007/978-3-540-79124-9_10.

[34] N. Tillmann and W. Schulte. Mock-Object Generation with Behavior. In *Proc. of Int. Conf. on Automated Software Engineering (ASE)*, pages 365–366. ACM, 2006. doi: 10.1109/ASE.2006.51.

[35] N. Tillmann, J. de Halleux, and T. Xie. Transferring an Automated Test Generation Tool to Practice: From Pex to Fakes and Code Digger. In *Proc. of the Int. Conf. on Automated Software Engineering (ASE)*, pages 385–396. ACM, 2014. doi: 10.1145/2642937.2642941.

[36] T. Xie, J. De Halleux, N. Tillmann, and W. Schulte. Teaching and training developer-testing techniques and tool support. In *Proc. of the Int. Conf. on Object oriented programming systems languages and applications companion*, pages 175–182. ACM, 2010. doi: 10.1145/1869542.1869570.