Generating Unit Isolation Environment Using Symbolic Execution

Dávid Honfi, Zoltán Micskei
Budapest University of Technology and Economics
Department of Measurement and Information Systems
Budapest, Hungary
Email: {honfi, micskeiz}@mit.bme.hu

***Abstract*—Research of source code-based test input generation has recently reached a phase, where it can be transferred to industrial practice. Symbolic execution is being one of the state-of-the-art techniques, yet its usage on industrial-sized software is often hindered by several factors, like the interaction with the environment of software under test. The solution of this problem can be supported with isolating the interactions by using so-called test doubles instead of the original objects. This time-consuming process requires deep knowledge of the unit under test. The technique presented in this paper is able to automatically generate isolation environment from the data collected during symbolic execution. We also present our promising preliminary results using a prototype implementation.**

I. Introduction

Generating test inputs and test cases from source code has been one of the main topics in software test research since decades. Numerous techniques and algorithms have been proposed to enhance the test generation processes by analyzing only the source code itself, commonly called as white-box test generation. Symbolic execution [1] is one of the state-of-the-art techniques due to the promising results of its fault detection ability. This technique represents possible paths of the source code with logical formulas over symbolic variables. The execution starts from an arbitrary method in the program and each statement is interpreted in parallel with gathering the expressions over the symbolic variables. The solution of these collected expressions provide input values that drive the execution of a program along different paths. Test cases are formed using these inputs extended with assertions derived from the observed behavior of the program under test.

In our previous research, we used Microsoft Pex [2] (currently known as IntelliTest) for test generation that is a state-of-the-art white-box test generator tool for .NET. Pex uses dynamic symbolic execution, which is an enhanced technique that combines concrete with symbolic execution. The tool generates input values for so-called Parameterized Unit Tests (PUTs) [3] that are simple unit test methods with arbitrary parameters. PUTs can be extended with assumptions and assertions, thus can serve as a test specification. Pex generates test cases from the combination of the generated inputs and the corresponding PUTs.

A current research topic is the industrial adoption of symbolic execution. However, it is hindered by several factors [4]–[7]. One of the main problems is that in the majority of cases, the test cases generated by symbolic execution typically achieve very low source code coverage. Our previous experiences [8] also confirmed these challenges. We applied Pex in testing of a model checker tool and a content management system.

A solution for this problem could be isolating the external dependencies of the unit under test (see Section II). However, in large-scale software that contains numerous components interacting with each other, a plentiful of calls to the environment can be identified. The identification and isolation of these calls is a highly time-consuming task, especially for test engineers who did not participate in the development of the unit.

In this paper, we present an approach and a prototype tool that endeavors to support symbolic execution-based test generation in complex, environment-dependent software by automatically generating code for isolation purposes. Thus, the research question of the paper is the following.

How can the isolation process be supported during symbolic execution-based test generation?

II. Background

Unit-level testing should be done in isolation, thus all the external dependencies of the unit should be removed or replaced. A solution could be to replace the external dependency with a *replacement object*, and call into that instead of the original. This idea and the increasing importance of unit testing led to a whole new area in software test engineering called *test doubles*.

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. Many types of test doubles exist, however the naming conventions can be different across publications. To overcome this, Meszaros wrote a summarizing book [9], that assesses the notions and patterns around unit testing, including test doubles too. We also applied the notions of this book extended with a Microsoft-specific type, called shims. Shims are powerful test doubles where the call is made to the original object, however the call is detoured during runtime to a user-defined method. This allows easy isolation of calls that invoke 3rd party libraries, legacy code or other resources where source is not available.

Consider the following example scenario introducing the isolation problem. Let `UUT` be the unit under test, and let `ED` be the external dependency. The body of method `M1(int):int` contains a branching where the decision depends on the return value of method `M2(int):int` from `ED`. Unit testing class `UUT` shall include the isolation of the mentioned dependency, moreover this shall be applied during symbolic execution-based test generation too. If method `M2` is not accessible during testing due to some reason (e.g., not yet implemented, accesses external resources) symbolic execution would not reach the statements found in the two branches.

```
class UUT
{
    ED ed = new ED();
    int M1(int a)
    {
        if(ED.M2(a)) return 1;
        else return -1;
    }
}
```

If we assume that parameter `a` will not be larger than 10 and only -1 or 1 can be returned, then creating a PUT for method `M1` would look like the following.

```
void M1Test(int a)
{
    Assume.IsTrue(a <= 10);
    UUT uut = new UUT();
    int result = uut.M1(a);
    Assert.IsTrue(result == 1 || result == -1);
}
```

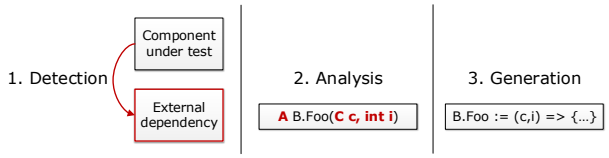
PUTs are used for compact representation of multiple test cases, where different input values trigger different behavior of the unit under test. It can be seen however that the PUT misses the isolation in this example. For the solution, we use the isolation syntax defined by Microsoft Fakes, a powerful isolation framework for .NET that uses shims, thus can isolate wide-range of external invocations. Extending the PUT with the following snippet will isolate the external call of `M2` for every instance of `ED` and return 1 or -1 depending on the value of the passed parameter, thus simulating a custom behavior.

```
ED.AllInstances.M2 = (ED instance, int param) =>
{ return param > 5 ? 1 : -1; };
```

III. Overview of the Approach

In order to support symbolic execution-based test generation, our approach is to generate the source code of the isolation environment automatically. This novel technique uses the collected data from the symbolic execution process itself.

Fig. 1. The approach of automated isolation



The technique builds on top of parameterized unit tests in order to have test doubles, which can give back values that are relevant to the component under test. A quick overview of the approach is presented in Figure 1.

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.

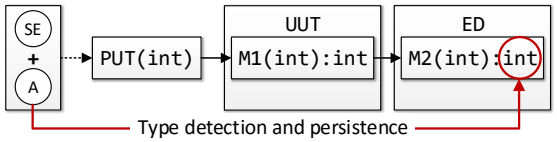
A. Detection

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 by analyzing the reached stack frames. When an external invocation is detected, all the information regarding this call is collected and stored for later usage by the analysis step.

B. Analysis

The analysis step plays the main role, since the decisions are made here that define how double objects are generated. The analysis can be divided into three substeps, which are the *analysis of return value*, the *analysis of parameters* and the *assessment* of the results gathered from the previous steps.

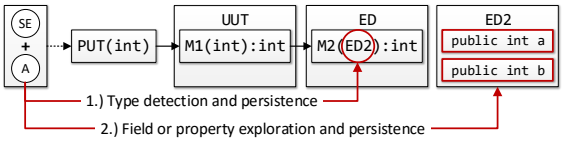
Fig. 2. Overview of return value analysis example



1) Return value analysis: In the first step, the return value of the invoked external method is analyzed, which is a crucial information of the double object. Figure 2 shows the overview of the code example mentioned in Section II, where SE denotes the symbolic execution process and A stands for our analysis technique. The return value can be used in the unit under test in logical conditions, thus execution paths possibly rely on this value. In order to cover these paths, the correct value must be selected. If a path relies on the variable that obtained its value from the external call, 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). 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. In summary, our idea provides a solution with the analysis of the return value, where two actions are done.

- The parameters of the PUT is extended with the return type of the external dependency.
- A test double is created in order to replace the behavior of the original class. In the body of the double, the new PUT parameter is returned, which gives the ability for symbolic execution to handle it as a free variable that can have arbitrary values.

Fig. 3. Overview of parameter analysis example



2) Parameter analysis: The analysis of parameters is the second part of the analysis, 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. 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 in the unit under test. Our idea to overcome 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, the generated results of our approach can simulate the actions made inside the external dependency that can be required to increase the coverage inside the unit under test. Figure 3 presents this process on the extension of the example found in Section II. The scenario contains a new `ED2` external dependency as a complex type parameter.

3) 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, which also includes the doubles of complex type parameters.

C. Generation

The generation is the last step of processing an external dependency, which can be also divided into substeps. Firstly, the newly created parameters of the parameterized unit tests are 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 test doubles of the complex type parameters are generated that are property or field setter methods (if the type of the parameter is located outside the unit under test).

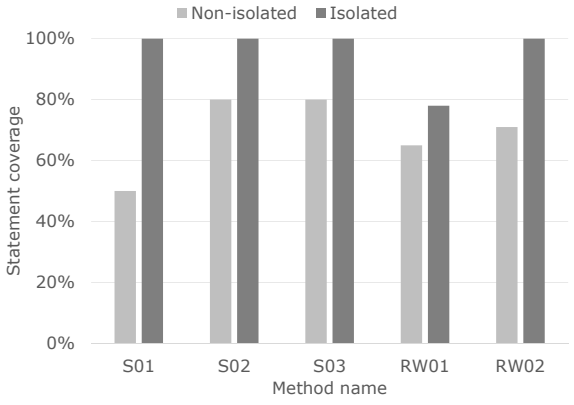
IV. Preliminary Evaluation

We implemented the presented technique as an extension to Microsoft Pex and Fakes. Fakes seamlessly collaborates with Pex and is powerful enough to use for our approach. Fakes is capable of creating stubs and shims for a very wide-range of method calls (regardless their place and type) found in any .NET software.

The implementation had many challenges including the run-time reflection of types and methods. We used this technique to obtain detailed information on each external call that is required for the return value and the parameter analysis.

Although the current prototype implementation does not support every scenario that can be found in arbitrary .NET code, it does work sufficiently for preliminary evaluation. We implemented example scenarios (intentionally similar to real-world source code snippets) to present the potential in our approach. The coverage results with and without our technique is presented in Fig 4. This figure shows the achieved statement coverage with and without the generated isolation environment after symbolic execution on the five different method under test. The names starting with *S* denotes simple scenarios where symbolic execution can benefit from the isolation. The two methods denoted with *RW* represent source code gathered from open-source software ([10] and the CMS system mentioned in Section I. There were no hindering factors during the measurement, however we wanted to make sure that our results are valid: the evaluation was repeated three times on each method. The presented figure shows the average statement coverage of the three executions.

Fig. 4. The results of the preliminary evaluation



The results show that in each case, running Pex with the generated isolation environment achieves higher statement coverage than running the tool without any unit isolation. This may emphasize that this technique has potential in the area of supporting the usage of symbolic execution in large-scale industrial software.

V. Related work

Our idea originally derives from a paper written by Tillmann *et al.* [11], where the idea of mock object generation is described. They also created a case study for file-system dependent software [12], 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 consider reference type parameters that can affect the coverage. Thus, our solution covers a wider area of scenarios and needs rather less user interaction for the automated generation (our approach 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 [13]. These contracts describe preconditions of a method, thus the derived mocks are in respect of them, which makes mocks 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 introduced in parallel with a symbolic execution engine to Java by Islam *et al.* [14]. 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.* [15]. 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 approach relies on symbolic execution.

VI. Conclusion and future work

One of the discovered challenges in real-world scenarios was the unit isolation in testing a software component, because these applications have several external dependencies (e.g., databases, external services). Isolating the dependencies requires large amount of time, which can be reduced by automation.

The described isolation technique in this paper could support the solution of this problem by automatically generating isolation environment. The main idea is to detect the dependencies during the symbolic execution and to generate the isolation environment for the unit under test from the data retrieved from symbolic execution. We also presented our preliminary evaluation of a prototype implementation that showed promising results.

The approach presented in this paper may be continued in the following directions.

- Expanding the implementation to cover all the possible unit isolation cases that could be present in real-world software.
- Experiments and measurements for the presented technique to confirm its usability in different scenarios.
- Combination of automated isolation and compositional symbolic execution [16] that leads to a new level of automated test input generation, where the work of test engineers can be greatly supported.

References

[1] J. C. King, "Symbolic Execution and Program Testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[2] N. Tillmann and J. de Halleux, "Pex–White Box Test Generation for .NET," in *TAP*, ser. LNCS, B. Beckert and R. Hähnle, Eds. Springer, 2008, vol. 4966, pp. 134–153.

[3] J. de Halleux and N. Tillmann, "Parameterized Unit Testing with Pex," in *TAP*, ser. LNCS, B. Beckert and R. Hähnle, Eds. Springer, 2008, vol. 4966, pp. 171–181.

[4] X. Qu and B. Robinson, "A Case Study of Concolic Testing Tools and their Limitations," in *Empirical Software Engineering and Measurement (ESEM), 2011 Int. Symposium on*, Sept 2011, pp. 117–126.

[5] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, "Does automated white-box test generation really help software testers?" in *Proc. of the 2013 Int. Symposium on Software Testing and Analysis*, ser. ISSTA 2013. ACM, 2013, pp. 291–301.

[6] P. Braione, G. Denaro, A. Mattavelli, M. Vivanti, and A. Muhammad, "Software testing with code-based test generators: data and lessons learned from a case study with an industrial software component," *Software Quality Journal*, vol. 22, no. 2, pp. 1–23, 2013.

[7] 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 29th ACM/IEEE Int. Conf. on Automated Software Engineering*, ser. ASE '14. ACM, 2014, pp. 385–396.

[8] D. Honfi, Z. Micskei, and A. Vörös, "Support and Analysis of Symbolic Execution-based Test Generation, TDK thesis, BME," 2014.

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

[10] Telerik, "JustDecompile Decompilation Engine," 2015. [Online]. Available: https://github.com/telerik/justdecompileengine

[11] N. Tillmann and W. Schulte, "Mock-Object Generation with Behavior," *Proceedings - 21st IEEE/ACM Int. Conf. on Automated Software Engineering, ASE 2006*, pp. 365–366, 2006.

[12] 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," *Proc. of the 2009 ICSE Workshop on Automation of Software Test, AST 2009*, pp. 149–153, 2009.

[13] S. J. Galler, A. Maller, and F. Wotawa, "Automatically Extracting Mock Object Behavior from Design by Contract Specification for Test Data Generation," in *Proceedings of the 5th Workshop on Automation of Software Test*. ACM, 2010, pp. 43–50.

[14] M. Islam and C. Csallner, "Dsc+Mock: A Test Case + Mock Class Generator in Support of Coding against Interfaces," in *Proc. of the 8th International Workshop on Dynamic Analysis*. ACM, 2010, pp. 26–31.

[15] B. Pasternak, S. Tyszberowicz, and A. Yehudai, "GenUTest: a Unit Test and Mock Aspect Generation Tool," *International Journal on STTT*, vol. 11, no. 4, pp. 273–290, 2009.

[16] S. Anand, P. Godefroid, and N. Tillmann, "Demand-Driven Compositional Symbolic Execution," in *TACAS*, ser. LNCS. Springer Berlin Heidelberg, 2008, vol. 4963, pp. 367–381.