User-Defined Sandbox Behavior for Dynamic Symbolic Execution

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

***Abstract*—Code-based generation of unit tests is an active topic of research today. One of the techniques is dynamic symbolic execution (DSE) that combines concrete executions with symbolic ones. The mature state of research in DSE allowed the technique to be transferred for industrial use. However, the complexity of software used in industrial practice have shown that DSE has to be improved in numerous areas. This includes handling dependencies to the environment of the unit under test. In this paper, we present a user-oriented approach that enables users to *provide input constraints and effects* for isolated environment dependencies in a parameterized sandbox. This sandbox collaborates with DSE, thus the isolated dependencies do not produce unrealistic or false behaviors.**

I. Introduction

Today, the demand for high quality software is greatly increasing. There are several techniques to improve quality, one of them is unit testing. A unit is a small portion of software, which can be a function, a class or even a set of classes. Most software projects use unit testing throughout the development lifecycle. Due to this fact, significant amount of time and effort is invested into unit testing.

Numerous ideas and techniques have already addressed to reduce this time and effort. Symbolic execution [1] is one of these techniques, as it is able to generate tests based on source code by using symbolic variables instead of concrete ones. During the symbolic execution process, each statement is interpreted and constraints are formed from the symbolic variables. When the execution has finished, these constraints can be solved using special solvers to obtain concrete input values. These inputs are used to execute the path on which the satisfied constraints were collected (path condition – PC). An advanced variant of this technique is dynamic symbolic execution (DSE) that combines concrete executions with symbolic ones in order to improve coverage [2], [3], [4]. The development of DSE has been progressing in the recent years in such a way that its industrial adoption became feasible. However, the technique faces several issues when it is used on large-scale programs with diverse behaviors [5], [6].

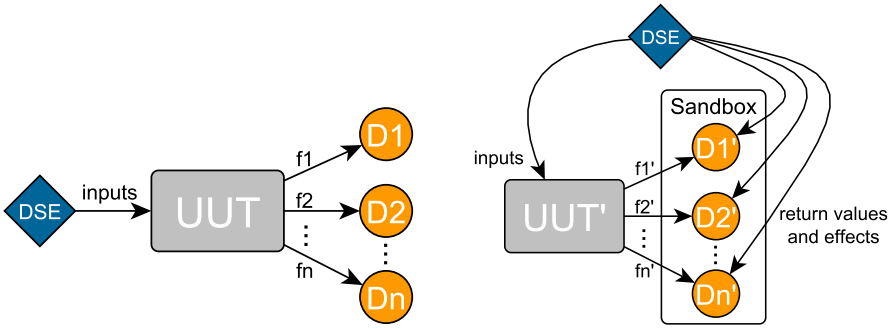
One of the most hindering issues is the isolation of dependencies. Without any isolation, – due to the concrete executions – the DSE test generation process would access the dependencies of the unit under test, which could cause undesirable events during test generation like accessing the file system, databases, or reaching parts of the software that are outside of the testing scope. To alleviate this, isolation (mocking) frameworks are commonly used in unit testing to handle the dependencies. Most of these frameworks however are not designed to be used with DSE-based test generation as the built-in low-level techniques in both may conflict with each other. To overcome this, we introduced a technique (based on a previous idea [7]) that is able to automatically handle the dependencies on the source code level by using transformations [8], [9]. These transformations replace invocations to the dependencies with calls to a parameterized sandbox generated with respect to the signatures of the methods being invoked. Then, the parameterized sandbox is used by DSE to provide return values and effects to objects resulting in the increase of coverage in the unit under test.

As the parameterized sandbox is filled with values by dynamic symbolic execution, the behavior depends on the underlying algorithm only. In most cases, DSE uses values that are relevant only to the code of the unit under test. This may cause important behaviors to be omitted for various reasons like 1) when the solver of the DSE is not able to obtain concrete values, or 2) when a combined behavior of two dependencies yields new behavior in the unit under test, or even 3) when the dependencies have some form of specification, which restricts the possible values. Also, the unrestricted behavior of the sandbox may result in false and unwanted behaviors for the unit under test.

In this paper, we address this issue by presenting an approach that builds on partial behaviors defined by developers to enhance the generated sandbox resulting in covering more relevant scenarios in the unit under test. The rest of the paper is organized as follows. Section II gives a detailed overview of the problem regarding the generated sandbox, along with presenting related concepts. Section III presents the approach with a detailed description of its process. In Section IV, a complex example is presented to provide better understanding of the approach. Section V summarizes our contribution.

II. Overview

Our approach for supporting DSE-based test generation with automated isolation consists of two main steps [9]: 1) transforming the unit under test and 2) generating a parameterized



(a) DSE without isolation.

(b) DSE with parameterized sandbox.

Fig. 1. The essence of the automatically generated parameterized sandbox approach.

sandbox. In step one, the abstract syntax tree of the unit is transformed using various rules, including the replacement invocations to dependencies with calls to the sandbox. Then, in step two, the sandbox is generated for the transformed invocations. The methods defined inside the parameterized sandbox receive the same *input* values as the original dependency would. Also, the generated methods may have *effects* on the passed objects, and may also *return* values given by DSE through the parameters of the sandbox. See Figure 1 and [9] for more details of the approach. The unit under test denoted with UUT uses n dependencies (D_k) through functions f_k. These dependencies also exist in the sandbox in a parameterized form, where the return values and effects are provided by the DSE.

An issue with the parameterized sandbox – that may hinder DSE from generating relevant cases – is that its behavior is uncontrolled, thus completely relies on the values generated by DSE. However, these values, in most of the cases are only relevant for covering more statements: their behaviors are not depending e.g., on 1) the state of the unit under test, 2) the state of other called dependencies, 3) their own state.

In order to overcome this issue, our approach employs inputs defined by the users of DSE, i.e. developers or testers. For the precise definition of input types we employed existing concepts (defined below) from related works on compositional symbolic execution [10].

Partition-effects pair [11]: A partition-effects pair (i, e) has 1) an input constraint i that is an expression over constants and symbolic variables, 2) an effects constraint e, which denotes expressions of symbolic variables bounded to values. A set of partition-effects pairs is used to describe the behavior of a method. A constraint i defines a partition of the input space and an effect e denotes the effects when the given argument values are found in the partition i.

Symbolic summary [12]: A symbolic summary of method m is a set of partition-effects pairs $m_{sum} = \{(i_1, e_1), (i_2, e_2), ..., (i_k, e_k)\}$, where $i_1, i_2, ..., i_k$ are disjoint.

An important aspect of compositional symbolic execution is to execute each unit separately and create method summaries to other units that use the currently analyzed one. The concepts introduced before can be employed for both static and dynamic compositional symbolic executions, although that requires the ability to execute the dependencies.

However in terms of DSE, this may not be possible for various reasons (e.g., no code/binary is available, DSE fails with traversal, dependencies are environments like file systems or databases). The previously presented automatically generated sandbox may be able to handle these cases, yet the behavior of the sandbox must be given somehow as no compositional execution is performed. Our approach is to involve developers and testers to provide those behaviors for the sandbox.

III. Approach

The basic idea of our approach is to obtain behaviors for the generated sandbox incrementally from developers and testers (the *users* of DSE-based test generation). These behaviors are defined in an easily readable format then transformed to symbolic method summaries with a preliminary logical check. When the check passes, source code is generated, which can be used by DSE as summaries for the corresponding execution paths.

Several other related works have already addressed the problematic area of isolation in symbolic execution-based test generation (e.g., [13], [14], [15]). These approaches also rely on user input at some point, however these employ various forms of user inputs differently than ours. Also, our approach uses incremental refinement by allowing the definition of partial behaviors as well.

This section uses an example to provide better understanding of our approach. The method `M(object o, bool b):int` used is a dependency of the unit under test. The real behavior of the method is out of scope for this example. Note that `object o` is passed by reference, thus its state can be affected in `M`.

Incremental refinement. Our approach uses incremental refinement of behavior (see Figure 2). At first, only the automated isolation transformation and sandbox generation are performed (0th iteration). After this step, the behavior of the sandbox depends only on DSE. The generated effects, input and output values for each dependency are presented to the user. Based on these, the user can decide to alter the behavior of each called dependency (1st iteration). The change of behavior is defined in a table structure (presented in Table I

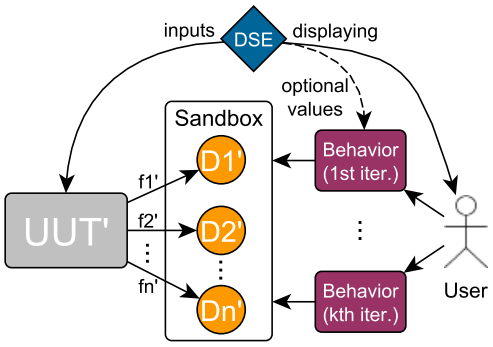


Fig. 2. The approach for user-defined sandbox behavior.

TABLE I
TABLE STRUCTURE FOR PROVIDING BEHAVIOR WITH TWO EXAMPLES.

Input (I)		Effects (E)	Return
o	b	o	
not null	true	new object()	>10
null	true	-	-1

for the example dependency method `M`). This table describes the constraints for each parameter, effect and for the return value as well. Also note that during the incremental usage, the user would see the values provided by DSE in each column.

In Table I, method `M` has two behaviors defined by the user overriding the arbitrary default. The first defines when parameter `o` is not null and parameter `b` is set to `true`, then `o` is assigned to a new object, and the method returns more than 10. The second states that when `o` is null and `b` is `true`, then the method returns constant -1. Note that in this example, the user provided only an under-approximation of the behavior as not all of the input partitions were covered (omitted where parameter `b` is `false`). For unspecified cases, our approach employs an `otherwise` feature, where both the effects and return value are set by the DSE.

Before the next execution, the automated isolation technique transforms the behavior (provided by the user) to source code into the sandbox. However, a validity check is performed before the code generation as the input partitions must be disjoint. In order to check this, our approach transforms the user-defined behaviors to symbolic summaries first, and then generates code from the summaries. The transformation from the table-structured user-defined behavior to the symbolic summaries can be defined as follows.

Transforming behaviors to symbolic summaries. A user-defined behavior b for method `m` consists of an input constraint $i \in I_m$, an effect constraint $e \in E_m$ and a return value r_m. Also, the behavior of method m can be described using multiple $b_k \in B_m$. First, it is checked that if $\bigwedge i_{b_k}$ for every k is unsatisfiable, i.e. checking if the input partitions defined by the user are disjoint, thus there are no conflicts between them. If the check passes, then from all $b_k \in B_m$ the

$$\bigvee_k (i_{b_k}, e_{b_k} \wedge r_{b_k})$$

symbolic summary is created.

Consider the example in Table I. The symbolic summary to be created here – based on the previous description – can be formulated as follows: $(o \neq null \wedge b == true, o == new\,object() \wedge return > 10) \vee (o == null \wedge b == true, return == -1)$.

The code generation for the sandbox is performed along predefined rules. These are designed to produce code that appends the symbolic summary to the path conditions of DSE. The input partition constraints are transformed to `if` statements and conditions, while the effects (as assignment statements) and return constraints are appended to the body of the sandbox method. Consider the example in Table I, the generated code for the sandbox method of method `M(object o, bool b):int` would be as follows.

Listing 1. Sandbox code for method m(object o, bool b):int.

```
public int FakeM(object o, bool b) {
  if(o != null && b == true) {
    o = new object();
    return DSE.ChooseFromRange<int>(11, int.Max);
  }
  if(o == null && b == true) {
    return -1;
  }
  return DSE.Choose<int>();
}
```

After the code generation has finished, the user is ready to re-execute the DSE test generation process to obtain new tests (this time with the user-defined sandbox behavior). In the next step of the refinement, the user gets the values generated by DSE again. The user can refine the behavior again (Figure 2 – nth iteration) by filling, appending or modifying the table previously introduced (Table I). This process can be repeated until the user finds the behaviors acceptable.

IV. COMPLEX EXAMPLE

The following example introduces an advanced scenario, where the user-defined behavior of the parameterized sandbox helps reducing the number of tests that trigger false behavior. The example is based on a backend service of an e-commerce site.

The current example focuses on a module of the service, which is responsible for handling advertisements. More precisely, the method under test (Listing 2) receives a product identifier and gets the advertisements (`Ads`) for that product. In its current implementation, the ads are consisting of related products only. The query of the related products is performed through database using the data access layer of the backend (`DB`). This layer fills in a set of integer identifiers of the related products. If there are related products, then the query returns `true`, while returns `false` if the set remained empty. The method under test adds the set of identifiers to the advertisement object to be returned. The addition causes an error if the set being passed is empty as it cannot occur in real use with respect to the specification.

In this example, the `DB` module is a dependency and has to be isolated during dynamic symbolic execution-based test generation in order to avoid unwanted accesses to the database.

Listing 2. Source code for complex example method.

```
public Ads GetAdsForProduct(int id) {
  if(id == -1) throw new NoProductExists();
  Set<int> ids = new Set<int>();
  bool success = DB.GetRelatedProducts(id, ids);
  if(success) {
    Ads ads = new Ads();
    ads.AddRelatedProducts(ids);
    return ads;
  }
  return null;
}
```

The first step of the incremental refinement is transforming the unit to use the parameterized sandbox. In the 0th iteration, the sandbox is only controlled by dynamic symbolic execution. The generated sandbox code would be as found in Listing 3.

Listing 3. Partial source code of the generated sandbox for advanced example.

```
bool FakeGetRelatedProducts(int id, Set<int> ids) {
  ids = DSE.Choose<Set<int>>();
  return DSE.Choose<bool>();
}
```

The issue with this unrestricted sandbox behavior is that there is no connection between the effects and the return value of the method. This causes a false behavior: the sandbox method can return `true` even when the set of identifiers remains empty causing the method under test to unintentionally crash. To tackle this issue, users can define the partial behavior as found in Table II.

TABLE II
TABLE STRUCTURE FOR PROVIDING BEHAVIOR WITH TWO EXAMPLES.

Input (I)		Effects (E)	Return
id	ids	id	
-	not null	not empty	true
-	null	-	false

Listing 4. Partial source code of the generated sandbox for advanced example

```
bool FakeGetRelatedProducts(int id, Set<int> ids) {
  if(ids != null) {
    for(int i = 0; i <= 10; i++) {
      ids.Add(DSE.Choose<int>(i));
    }
    return true;
  } else if(ids == null) {
    return false;
  }
}
```

The generated code for these restrictions modifies the parameterized sandbox as found in Listing 4. The code contains a loop with 10 iterations (predefined with sandbox loop settings) to fill in the set with identifiers and return `true` if the set was not null. Otherwise, the code returns `false`. There is no need for other branches as the input space is partitioned into two parts by the user-defined behaviors. Using this parameterized sandbox, the DSE will not generate test cases, where the set is empty, and the method returns `true`, thus avoids the problematic false behavior that would cause a known error. Note that this example can be continued with subsequent iterations of the incremental process to refine the behavior of the sandbox.

V. CONCLUSIONS

In this paper, we presented a user-oriented approach that extends our previous work [9]. Our results showed that an automatically generated parameterized isolation sandbox may improve DSE-based test generation. However, we were also able to identify that a fully parameterized sandbox – relying on values from DSE only – may 1) omit important behaviors or 2) cause false behavior to occur. To tackle this issue, we presented the idea of incremental user-defined behaviors in the parameterized sandbox. The users of DSE-based test generation are able to incrementally refine their behaviors defined for each dependency method in the isolated sandbox. In each refinement iteration, a corresponding code snippet is generated from the user-defined behaviors based on existing ideas from compositional symbolic execution. These generated code parts are used by DSE during test generation, thus are able to influence and steer the DSE process. This may reduce the false behaviors in the unit under test triggered by the parameterized sandbox.

REFERENCES

[1] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[2] 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*, vol. 29, no. 7, pp. 1758 – 1773, 2013.

[3] K. Sen, "Concolic testing," in *Proc. of the Int. Conf. on Automated Software Engineering*. ACM, 2007, pp. 571–572.

[4] 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.

[5] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Comm. of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[6] 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)*. IEEE, 2011, pp. 117–126.

[7] N. Tillmann and W. Schulte, "Mock-Object Generation with Behavior," in *Proc. of Int. Conf. on Automated Software Engineering (ASE)*. ACM, 2006, pp. 365–366.

[8] D. Honfi and Z. Micskei, "Generating unit isolation environment using symbolic execution," in *Proc. of the 23rd PhD Mini-Symposium*. BUTE-DMIS, 2016.

[9] D. Honfi and Z. Micskei, "Supporting Unit Test Generation via Automated Isolation," *Periodica Polytechnica, Electrical Engineering and Computer Science*, 2017, Accepted. In press.

[10] 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.

[11] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu, "Differential Symbolic Execution," in *Proc. of the 16th Int. Symp. on Foundations of Software Engineering*, 2008, pp. 226–237.

[12] P. Godefroid, "Compositional Dynamic Test Generation," in *Proc. of the 34th Symp. on Principles of Programming Languages*, 2007, pp. 47–54.

[13] S. J. Galler, A. Maller, and F. Wotawa, "Automatically Extracting Mock Object Behavior from Design by Contract Specification for Test Data Generation," in *Proc. of the Workshop on Automation of Software Test (AST)*. 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 Int. 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," *Int. Journal on STTT*, vol. 11, no. 4, pp. 273–290, 2009.