Augmented Dynamic Symbolic Execution

Konrad Jamrozik
Saarland University – Computer Science
Saarbrücken, Germany
jamrozik@cs.uni-saarland.de

Gordon Fraser
University of Sheffield
Sheffield, UK
gordon.fraser@sheffield.ac.uk

Nikolai Tillmann and Jonathan de Halleux
Microsoft Research
Redmond, USA
{nikolait,jhalleux}@microsoft.com

ABSTRACT

Dynamic symbolic execution (DSE) can efficiently explore all simple paths through a program, reliably determining whether there are any program crashes or violations of assertions or code contracts. However, if such automated oracles do not exist, the traditional approach is to present the developer a small and representative set of tests in order to let him/her determine their correctness. Customer feedback on Microsoft's Pex tool revealed that users expect *different* values and also *more* values than those produced by Pex, which threatens the applicability of DSE in a scenario without automated oracles. Indeed, even though all paths might be covered by DSE, the resulting tests are usually not sensitive enough to make a good regression test suite. In this paper, we present *augmented dynamic symbolic execution*, which aims to produce representative test sets by augmenting path conditions with additional conditions that enforce target criteria such as boundary or mutation adequacy, or logical coverage criteria.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Algorithms, Experimentation

Keywords

Test generation, dynamic symbolic execution, boundary values, mutation testing

1. INTRODUCTION

Dynamic symbolic execution (DSE) is a successful test generation technique, recently made popular with the advent of powerful constraint solving tools. There are many successful applications ranging from parametrized unit testing [12] to white-box fuzzing [7]. In principle, DSE approaches are only limited by the scalability of the underlying constraint solvers. However, all these successful applications make the common assumption that there is an automated way to determine for any given input to a program whether an error has occurred – in other words, there needs to be an automated *test oracle*.

```
1 void test(int x) {
2     if(x <= 100)
3       // target
4 }
5 }
```

```
1 void test(int x) {
2     if(x < 100)
3       // target
4 }
5 }
```

Figure 1: Pex trivially covers the target branch, e.g., with input 0. But what if there is no specification against which to check the result automatically? What if a fault is not present in the current but future versions of the program? There is no guarantee a test suite generated with DSE would detect the regression error on the right.

Automated oracles can be easily defined for some essential properties; for example, in general programs should not crash and there should be no buffer overflows. However, oracles for functional correctness need to be specified in terms of assertions, contracts, or other specification means. In practice, such oracles often do not exist, or are of insufficient quality. In this case, a traditional assumption in automated test generation is that the test set without oracles is handed to a developer, who will then check correctness or add oracles to the tests. Indeed, a common application of Microsoft's Pex [12] tool is to select a target function to "Pex explore" and to check what the function does. However, feedback has revealed that the values Pex returns are not necessarily those that developers would expect. For example, Pex always first tries to use 0 or `null`, and then uses values obtained from an SMT solver. In practice, developers would prefer to see values that they can relate to the code – for example, values that are at the boundaries of comparisons, and often they also want to see more than one value for a particular branching condition.

In fact, this is not simply an issue of convenience: The result of a DSE run is a test suite that can be used to perform regression testing. After changing a program a new DSE exploration would just exercise the current behavior with respect to automated oracles, yet to find regression faults we need to execute the tests produced from an earlier version. However, the nature of DSE is that for each program

ASE '12, September 3-7, 2012, Essen, Germany

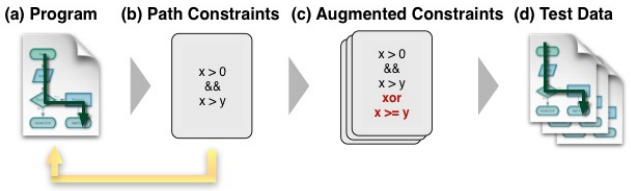

Figure 2: Augmented Dynamic Symbolic Execution is based on exploration of a program (a) through path conditions (b). Yet, before test data is generated, the conditions are augmented with additional conditions (c), deriving multiple test data for the same path condition (d).

path there is at most one input value that exercises it. Consider the simple example in Figure 1: Even though Pex can trivially cover the branch in both program versions, only the input value 100 would be able to detect the regression that was introduced from the left version to the right version.

To overcome these issues, in this paper we present *augmented dynamic symbolic execution* (ADSE, Figure 2): This approach takes the path conditions generated from a program (a) during regular DSE (b) augments them (c) with additional conditions to make sure that the constraint solver returns interesting values, resulting in a test suite (d) satisfying a criterion underlying the augmentation. The augmentation can be based on many different criteria commonly found in software testing. For example, it is straight forward to define augmentation rules for boundary value testing, mutation testing, logical coverage criteria, or error conditions.

Our preliminary experiments using boundary values analysis and mutation testing with our Apex prototype confirm that ADSE can lead to a considerable increase in the mutation score for both types of augmentations.

2. BACKGROUND

Symbolic execution is a technique that maps a program path to a set of conditions on the program inputs. Branching conditions (e.g., `if`, `while`) represent the individual conditions in these sets, and the conditions are based on expressions on the input variables. Any input satisfying the conditions will follow this path through the control flow graph.

Symbolic execution has several limitations, such as inability to reason about interactions with the environment because of its static nature and scalability issues due to exponential number of possible execution paths. *Dynamic symbolic execution* (DSE) overcomes these limitations by interleaving program execution with symbolic execution, systematically exploring uncovered code branches. There are various tools that implement this approach [3]; we built our Apex prototype on top of the Pex [12] tool, a DSE engine for the .NET platform. Pex utilizes parameterized unit tests for the exploration with DSE, and produces unit test suites achieving high branch coverage, as well as tests demonstrating exceptions thrown by the tested code or specification violations.

Many systematic test generation approaches are focused on branch conditions. DSE explicitly considers branch conditions, and search-based testing [8] is usually guided by heuristics that estimate how close individual branches are to evaluating to true or to false. To apply the existing techniques to different target criteria, a common approach is to transform these other criteria to branch coverage problems. This has for example been done for division by zero errors [2], null pointer exceptions [10], mutation testing [14], or boundary value analysis and logical coverage criteria [9]. Here, additional test objectives are explicitly included in the program code in terms of new branch instructions, such that existing tools that are already good at achieving branch coverage can be reused. There are several drawbacks to such an approach:

- It requires a suitable infrastructure to handle the targeted source code or byte code representation.
- Applying transformations to code is cumbersome, as it adds the technical overhead of adhering to the underlying language grammar.
- Source code transformations have to be tailored to DSE engine exploration strategies. For example, Pex by default emits a new test case whenever it increases branch coverage, so transformations that do not add any branches will be ignored by Pex's DSE exploration altogether.
- Source code transformation based techniques might be counteracted by summarization approaches in DSE tools, which collapse artificially introduced trivial multiple execution paths into disjunctive constraints. This would reduce the number of values produced by the DSE tool, such that the criterion underlying the transformation is not satisfied.

These are severe drawbacks threatening the practical applicability of source-transformation based approaches in the context of DSE. In contrast, ADSE performs all transformations directly on the path conditions, avoiding these issues.

3. AUGMENTED DYNAMIC SYMBOLIC EXECUTION

3.1 Generating Test Suites with DSE

DSE executes a program using concrete values, and during this execution it keeps track of the symbolic state determined by expressions and conditions (i.e., branches) on the input values. This results in a path condition, which represents the conditions seen while executing the program. DSE then systematically modifies the path conditions according to the code exploration strategy and solves them using a constraint solver, thus generating test inputs that cover new branches and reveal errors.

For example, to reach `block_A` in code given in Figure 3, DSE will build and solve the following path condition: $(x \geq 3 \wedge x \leq 7) \wedge (y - x \leq 0 \vee y \geq 4)$, which may for example result in concrete values for (x, y) of $(4, 5)$ returned by the constraint solver.

Assuming that there are no assertions violated and no exceptions triggered, the result of DSE in this case is a set of test cases – one for each path condition that increased branch coverage.

3.2 Augmenting Path Conditions

In ADSE, each time a path condition is selected for test generation, instead of solving the path condition itself we

```
1 void methodUnderTest(int x, int y)
2 {
3   if (x >= 3 && x <= 7) {
4     if (y - x <= 0 || y >= 4) {
5       // block_A
6     }
7     else {
8       // block_B
9     };
10  };
11 }
```

Figure 3: Code example to illustrate DSE.

augment it using given augmentation criterion, obtaining a set of *augmented path conditions*; tests are generated by solving these conditions, and each solution to such a condition is added to the resulting test suite. As a simple example, using the identity function as augmentation criterion would produce the original branch coverage test suite. Inputs provided by the constraint solver as a solution to one augmented path condition will sometimes also solve other augmented path conditions (i.e., collateral coverage). We can exploit this to minimize the number of generated tests while fully adhering to the used augmentation criterion.

3.3 Instantiations of ADSE

We now instantiate the transformation criterion for different common testing objectives.

3.3.1 Boundary Value Testing

To satisfy the branch coverage criterion, each branch needs to be covered by at least one test case. However, in practice it is often desired to cover boundary cases of branch conditions of the tested code [9, 13]. For example, to test the boundaries of the condition in Line 3 in Figure 3, the target function should be tested with input x equal to $2, 3, 4, 6, 7$ and 8. To augment path conditions for boundary cases a simple strategy is to transform each comparison according to a set of rules. For example, $A \leq B$ would be transformed to $A = B$ (boundary value) and $A < B$ (representative value). The boundary case where the condition does not hold is automatically handled as by construction ADSE will also consider the negation of the condition, i.e., $A > B$, for which a boundary case might be $A = B + 1$.

3.3.2 Mutation Testing

Mutation testing [5] is a technique where simple syntactic changes (mutations) are applied to the code in order to simulate faults, which was proven to be effective in evaluating testing techniques [1].

Different types of mutations can be defined in terms of mutation operators, where each mutation operator typically can be applied to several different locations in a program, each time resulting in a new mutant; usually, only mutants that differ by a single change from the original program are considered. For example, in Line 3 in Figure 3 the condition `x <= 7` could be mutated to the following conditions: `x < 7`, `x == 7`, `x != 7`, `x >= 7`, `x > 7`.

In general, given a set of mutation operators $M(C)$ for a condition C the augmentation should ensure that there is a value for each of the mutants $C' \in M(C)$ that distinguishes between C and C', i.e., C is true and C' is false (the inverse case is covered when $\neg C$ is explored). Therefore, the augmentation function for condition C simply is:

$$\mathcal{T}(C) = \{C \wedge \neg C' \mid \forall C' \in M(C)\} \tag{1}$$

3.3.3 Logical Coverage

In presence of complex predicates in the source code a branch coverage test suite might simply be too weak. For example, safety standards such as DO-178b [11] require that test suites satisfy the MCDC coverage criterion [4].

As an example, masking MCDC requires that for each clause in a logical condition there exists a state such that the clause determines the value of the condition, and the clause has to evaluate to true and to false. For example, the branch in Line 4 in Figure 3 consists of two clauses, $y - x \leq 0$ and $y \geq 4$. Each of the two clauses determine either the true or false outcome of the condition only if the other one evaluates to false. In general, a clause p determines a condition C if the following xor-expression is true, where $C_{p,x}$ denotes C with p replaced with x: $C_{p,True} \oplus C_{p,False}$. Consequently, the transformation of a condition C requires that for every clause $p \in C$ we add a condition such that C is true and p determines the outcome of C:

$$\mathcal{T}(C) = \{C \wedge (C_{p,True} \oplus C_{p,False}) \mid p \in C\} \tag{2}$$

As the DSE exploration will lead to application of the transformation to both C and $\neg C$, this means that the above transformation will ensure that p evaluates both to true and to false.

3.3.4 Error Conditions

The fourth instance of ADSE we consider in this paper is that of error conditions: the augmented path conditions are to cause the tested code to throw exceptions like arithmetic overflow or array index out of bounds. There have been attempts to make such implicit error conditions explicit to allow test generation tools to cover these cases (e.g., [2, 10]), Pex also makes these branches explicit [12], and SAGE includes such conditions in the properties it checks for [6]. If the test generation tool does not already have treatment of error conditions hard coded, these can easily be represented as path condition augmentation rules. For example,

$$\mathcal{T}(C) = \begin{cases} \{C\} & \textit{if there are no divisions in } C \\ \{C \wedge x = 0 \mid \forall x : \text{divisors in } C\} \cup & \\ \{C \wedge x \neq 0 \mid \forall x : \text{divisors in } C\} & \textit{otherwise.} \end{cases} \tag{3}$$

4. INITIAL RESULTS

Our Apex prototype implements the described approach as an extension to the Pex tool, and we applied it to a set of example functions to evaluate the effects of the condition augmentation on the resulting test suite size as well as fault detection ability. Pex operates on .NET byte-code (CIL), and all complex predicates in the source code are translated to atomic conditions in the byte-code. Furthermore, the symbolic execution engine in Pex already makes error conditions explicit as branches. Consequently, our evaluation focuses on boundary value analysis and mutation testing.

Table 1 shows the initial results achieved with our Apex prototype on a set of nine different examples taken from the literature. The column labelled DSE denotes the standard Pex behaviour, whereas ADSE/B and ADSE/M denote the

Table 1: Averaged values for initial APEX experiments, using nine example programs

Result	DSE	ADSE/B	ADSE/M
Conditions	14.22	595.11	2,225.33
Tests	14.22	109.33	123.56
Mutation Score	48.44%	80.66%	86.16%

results achieved with APEX and augmentation for boundary values and mutation testing, respectively. These results indicate the following:

- ADSE leads to an increased fault detection ability. The mutation scores achieved by both the test sets optimized for boundary value testing and for mutation testing are significantly higher than those for branch coverage test sets.
- ADSE leads to significantly more conditions that need to be solved.
- Because one test case usually covers more than one test target, the number of test cases that results from the large set of conditions is significantly smaller than the number of conditions.

These results are in line with expectations: Choosing a rigorous coverage criterion will lead to larger test suites and larger effort in generating them. The higher mutation score reflects that this higher effort is worthwhile when the fault detection ability needs to be maximized, for example in a regression testing scenario. The higher number of conditions and test cases also suggests that we can accommodate for the desire for more test cases per branching statement expressed by Pex users.

5. CONCLUSIONS

Dynamic symbolic execution can efficiently generate inputs to cover all (simple) paths in a program. Yet, the use of DSE to produce test suites with high coverage of established test criteria has not been explored in depth. In this paper we describe a technique that transforms the path conditions that DSE handles, such that DSE produces test suites satisfying any chosen coverage criterion. We have identified four possible augmentations of DSE and conducted preliminary experiments for two of them: mutation testing and boundary cases coverage, with promising early results.

There are several immediate applications: ADSE can provide the programmer with more interesting tests such as boundary cases, which were requested by Pex users. This is also particularly useful in a scenario where no automated oracle is available, where we can augment DSE to produce tests satisfying a given criterion, e.g., a logical coverage criterion. A further important application is regression testing: When software evolves, one needs a regression test suite to check for regression faults. The stronger the test suite, the more sensitive it is against regression faults, and thus ADSE can lead to improved regression test suites.

Finally, ADSE also has the potential to extend DSE in many ways not immediately targeted by the augmentations we described in this paper, e.g., to enforce values out of known sets to increase tests readability or to trigger non-functional bugs, for example in performance testing.

6. REFERENCES

[1] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 402–411, New York, NY, USA, 2005. ACM.

[2] N. Bhattacharya, A. Sakti, G. Antoniol, Y.-G. Guéhéneuc, and G. Pesant. Divide-by-zero exception raising via branch coverage. In *Proceedings of the Third international conference on Search based software engineering*, SSBSE'11, pages 204–218, Berlin, Heidelberg, 2011. Springer-Verlag.

[3] 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 *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1066–1071, New York, NY, USA, 2011. ACM.

[4] J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, pages 193–200, September 1994.

[5] R. A. DeMillo, R. J. Lipton, and F. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[6] P. Godefroid, M. Y. Levin, and D. A. Molnar. Active property checking. In *Proceedings of the 8th ACM international conference on Embedded software*, EMSOFT '08, pages 207–216. ACM, 2008.

[7] P. Godefroid, M. Y. Levin, and D. A. Molnar. Sage: Whitebox fuzzing for security testing. *ACM Queue*, 10(1):20, 2012.

[8] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

[9] R. Pandita, T. Xie, N. Tillmann, and J. de Halleux. Guided test generation for coverage criteria. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[10] D. Romano, M. Di Penta, and G. Antoniol. An approach for search based testing of null pointer exceptions. In *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, ICST '11, pages 160–169, Washington, DC, USA, 2011. IEEE Computer Society.

[11] RTCA Inc. DO-178b: Software Considerations in Airborne Systems and Equipment Certification. Requirements and Technical Concepts for Aviation, Washington, DC, December 1992.

[12] N. Tillmann and N. J. de Halleux. Pex — white box test generation for .NET. In *International Conference on Tests And Proofs (TAP)*, pages 134–253, 2008.

[13] L. J. White and E. I. Cohen. A domain strategy for computer program testing. *IEEE Trans. Softw. Eng.*, 6:247–257, May 1980.

[14] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. de Halleux, and H. Mei. Test generation via dynamic symbolic execution for mutation testing. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.