Generating Test Suites with Augmented Dynamic Symbolic Execution*

Konrad Jamrozik[1], Gordon Fraser[2],
Nikolai Tillman[3], and Jonathan de Halleux[3]

[1] Saarland University, Saarbruecken-66123, Germany
jamrozik@cs.uni-saarland.de
[2] University of Sheffield, UK
gordon.fraser@sheffield.ac.uk
[3] Microsoft Research, WA, USA
{nikolait,jhalleux}@microsoft.com

Abstract. Unit test generation tools typically aim at one of two objectives: to explore the program behavior in order to exercise automated oracles, or to produce a representative test set that can be used to manually add oracles or to use as a regression test set. Dynamic symbolic execution (DSE) can efficiently explore all simple paths through a program, exercising automated oracles such as assertions or code contracts. However, its original intention was not to produce representative test sets. Although DSE tools like Pex can retain subsets of the tests seen during the exploration, customer feedback revealed that users expect *different* values than those produced by Pex, and sometimes also *more* than one value for a given condition or program path. This 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 with DSE by augmenting path conditions with additional conditions that enforce target criteria such as boundary or mutation adequacy, or logical coverage criteria. Experiments with our Apex prototype demonstrate that the resulting test cases can detect up to 30% more seeded defects than those produced with Pex.

1 Introduction

Automated tools for structural testing are typically applied in one of two possible scenarios: 1) to exercise automated oracles, for example provided in terms of partial specifications, code contracts, assertions, or generic properties such as program crashes; or 2) to produce a representative test suite that needs to be enhanced with test oracles by a developer. Dynamic symbolic execution (DSE) [5] is one of the most successful approaches to exercising automated oracles,

* A preliminary version of this paper was published as a short paper discussing the idea for mutation and boundary analysis without evaluation in [15].

and there are many successful applications ranging from parametrized unit testing [23] to white-box fuzzing [12].

However, the application of DSE in the alternative scenario where representative test suites are desired is less explored. Tools like Microsoft's Pex [23] use simple heuristics to filter the test cases explored during DSE to retain subsets achieving branch coverage. However, there is no systematic way to produce *several* values for an individual branch or path, so it becomes difficult to implement other criteria such as boundary value analysis. This is problematic for regression testing: After changing a program a new DSE exploration would again 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, as the DSE exploration might miss important values, this may lead to inferior regression test suites, potentially missing regression faults. Developer feedback has shown that these additional values are also missed during "Pex exploration", where Pex summarizes what a function does. For example, Pex always first tries to use 0 or `null`, and then uses values obtained from an SMT solver, but developers would prefer to see values that they can relate to the code, such as boundary values, and not `null`.

To overcome these issues, in this paper we present *augmented dynamic symbolic execution* (ADSE): This approach takes the path conditions generated from a program during regular DSE and augments them with additional conditions to make sure that the constraint solver returns interesting values, resulting in a test suite satisfying a criterion underlying the augmentation. To determine what constitutes an interesting value we consider *boundary values*, *mutation testing*, *logical coverage criteria*, and *error conditions*. In detail, the contributions of this paper are as follows:

Augmented DSE: We describe a generic approach to influence the test data produced with DSE by transforming path conditions.

Instantiations of ADSE: We describe several instantiations of augmented DSE; note that most of the underlying transformations have already been used in the past, yet always in a problem-specific context. In particular, we formulate the following transformations in terms of ADSE:

- **Boundary Value DSE**: We instantiate augmented DSE such that boundary value inputs are generated with dynamic symbolic execution.
- **Mutation Testing DSE**: We instantiate augmented DSE such that test cases that are good at killing mutants are derived with DSE.
- **Logical Coverage DSE**: We instantiate augmented DSE such that test cases that satisfy logical coverage criteria are derived with DSE.
- **Error Condition DSE**: We instantiate augmented DSE such that error conditions such as division by zero or overflows are triggered together with regular values.

Evaluation: We have implemented the approach in our Apex prototype as an extension to the popular Pex test generation tool, and present the results of an evaluation on a set of path conditions extracted with Pex.

2 Background

The availability of efficient constraint solvers has made it feasible to apply them to the task of generating test data. This is usually done by solving path conditions generated with symbolic execution. Symbolic execution maps a program path to a set of conditions on the inputs of the program. 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.

Unfortunately, constraint solvers cannot reason about a program's environment and run into scalability issues as programs become nontrivial in size. *Dynamic symbolic execution* (DSE) addresses these problems by using concrete executions to derive path conditions for feasible paths, which are systematically explored by negating individual conditions and deriving new inputs. Various tools implement DSE (e.g., DART [10], CUTE [21], Symbolic JPF [2], Pex [23], and others).

Apex is built on top of the Pex [23] tool, which performs DSE on the .NET platform. The main intended application of Pex and other DSE-based tools is to explore program paths with respect to automatically verifiable specifications such as assertions. Whenever Pex finds a new branch during DSE that was not covered before, a test case for this branch is added to the final test suite.

Many systematic test generation approaches are focused on branch conditions and branch coverage. To apply existing tools to different target criteria, a common approach is to transform these criteria to branch coverage problems. This for example has been done for division by zero errors [4], null pointer exceptions [19], mutation testing [26], or boundary value analysis and logical coverage criteria [18]. Here, additional test objectives are explicitly included in the program code as new branch instructions, allowing for reuse of tools maximizing branch coverage. There are several drawbacks to such an approach:

- A substantial, platform-dependent code manipulation architecture needs to be implemented, considerably increasing the complexity.
- Applied transformations have to conform to underlying language syntax, making them less flexible, harder to combine and cumbersome for dynamical adaptation of exploration targets.
- The code with transformations incorporated is still a subject to optimizations done by the underlying DSE engine, which can reverse or alter the applied modifications in an unexpected way, possibly causing path explosion.

For sure there would be ways to work around these drawbacks. However, in sum these are severe shortcomings threatening the practical applicability of source-transformation based approaches in the context of DSE. In contrast, ADSE performs all its transformations directly on the path conditions, avoiding these issues.

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

Fig. 1. Code example to illustrate DSE.

3 Augmented Dynamic Symbolic Execution

3.1 Generating Test Suites with DSE

Dynamic symbolic execution first executes a program using concrete values, which are usually randomly chosen, or default values. Along the execution path chosen, conditions are collected, such that the resulting path condition represents the control flow path – any solution to the path condition will follow the same path. For example, assume that the example in Figure 1 is first executed with the concrete values $(0, 0)$. The first `if` statement evaluates to false, which means the condition $\neg(x \geq 3 \wedge x \leq 7)$ is added to the conditions for this path, and block C is reached. Now this condition is negated to derive a new path condition, $x \geq 3 \wedge x \leq 7$, for which a constraint solver produces a solution such as $(5, 0)$. This new input is executed, covering block B, and the conditions along the new path are collected: $(x \geq 3 \wedge x \leq 7) \wedge \neg(y - x \geq 0 \vee y \geq 4)$. Here, again a condition not yet explored is chosen, negated, and the resulting path condition is solved, producing for example $(5, 5)$, which finally also reaches block A.

During the DSE exploration done by Pex a separate test suite T is generated. Whenever an input S generated by DSE executes a branch that is not yet covered by T, then a test with this input is added to T; furthermore in Pex all tests that trigger exceptions are added to T.

3.2 Augmenting Path Conditions

Augmentation may only increase the quality of the test suite, but needs to preserve the branch coverage of the test suite produced without augmentation. A basic prerequisite of our approach is therefore that the augmentation of a path condition does not change the execution path the path condition represents, or if it does, the inputs fulfilling the original path conditions are retained in the resulting augmented test suite.

Definition 1 (Control Flow Graph). *The **control flow graph** of a program is a directed graph $CFG = (B, E)$, where B is the set of basic blocks and E is the set of transitions. There is one dedicated entry node $s \in B$, and one dedicated exit node $e \in B$.*

The function $R(B)$ maps each $b \in B$ to its immediate control dependent branch condition c, where $R(s) = true$, and function $N(c)$ maps each condition c to its node $b \in B$. A path from the start node of the control flow graph is a control flow path:

Definition 2 (Control Flow Path). *A **control flow path** is a sequence of transitions $\langle t_1, \ldots t_n \rangle$, where $t_1 = s$ and $(t_i, t_{i+1}) \in E$. If $t_n = e$ the path is complete.*

Each control flow path can be represented as a path condition:

Definition 3 (Path Condition). *A **path condition** is a conjunction of branch conditions $P = \bigwedge_{i \in \{1..n\}} c_i$, such that every input m satisfying P (denoted $m \models P$) allows to execute the control flow path $\langle s, N(c_1), \ldots, N(c_n) \rangle$, assuming there are no jumps. If $N(c_n) = e$ then the path condition is complete.*

We use $C \in P$ as a shorthand when referring to the individual branch conditions in a path condition.

When applying ADSE in a scenario where we require additional values for one particular program path it is important to make sure that the original control flow path is not changed. We call this type of augmentation *strict*:

Definition 4 (Strictly Augmented Path Condition). *A **strictly augmented path condition** P' for path condition P is a transformation of P such that $\forall (I \in inputs)(I \models P' \Rightarrow I \models P)$.*

If we would replace test cases obtained from original path conditions with those obtained from strictly augmented path conditions, we run into the risk of losing branch coverage due to possibility of augmented path conditions infeasibility. In the worst case, no strictly augmented path condition is feasible, thus resulting in no tests generated. Furthermore, each condition will be augmented at least twice, as it will usually be explored in both true and false valuation. This may lead to an unnecessary overhead for verbose augmentations (e.g., mutation testing). Therefore, we define *weak* augmentation such that resulting augmented conditions only need to share a prefix path:

Definition 5 (Weakly Augmented Path Condition). *A **weakly augmented path condition** P' for path condition P at branch condition c_i is a transformation of P such that $\forall j < i : P_j = P'_j$.*

When applying weak augmentation it is important to include the original path conditions for test generation in order to guarantee the branch coverage does not decrease.

A transformation criterion is a function that takes as input a path condition, and produces a set of augmented path conditions as a result.

Definition 6 (Transformation Criterion). *A transformation criterion is a function $\mathcal{T}(P)$ that creates a set of augmented path conditions $\mathcal{P}'$ based on P.*

Algorithm 1 Test suite generation using weak augmentation.

Require: Program M
Require: Transformation Criterion $\mathcal{T}$
Ensure: Test Suite T

```
1: procedure GenerateSuiteAugmented(M)
2:     T ← {}
3:     while none of the exploration bounds were reached do
4:         P ← get next path condition
5:         S ← solve P
6:         if S is satisfiable and covers new branch then
7:             t ← test with S as input
8:             T ← T ∪ {t}
9:             for P' ∈ AugmentCondition(T,P) do
10:                S' ← solve P'
11:                if S' is satisfiable then
12:                    t' ← test with S' as input
13:                    T ← T ∪ {t'}
14:    end while
15:    return T;
16: end procedure
```

Given a transformation criterion $\mathcal{T}(P)$, we can now use DSE to produce a test suite using Algorithm 1. Each time a path condition is selected for test generation (Line 4), and it covers a new branch, we add a test based on it (to retain the code coverage) and apply the transformation function, to try to solve all augmented path conditions. The solution (concrete input values) to each such augmented path condition is added to the resulting test suite after discarding redundant solutions (see Section 3.3). The process ends if any of the exploration bounds is reached, for example in terms of a time limit or limit in the number of path conditions to explore.

The path condition is a conjunction of branch and loop conditions encountered during program execution. The AugmentCondition($\mathcal{T}$,P) call from line 9 weakly augments each of these conditions, one at a time, and discarding all conditions following the augmented condition. For example, assume that in path condition $C_1 \wedge C_2 \wedge C_3 \wedge C_4$ the condition C_1 has already been augmented. When augmenting C_2 the resulting condition would be $C_1 \wedge C_2'$.

3.3 Handling Redundancy

In normal DSE, the constraint solver is only queried for path conditions which lead to execution of a new control flow path. In contrast, the augmentation may lead to a set of path conditions that all follow the same control flow path. Thus, there is the possibility that the solution to one augmented path condition satisfies some of other augmented conditions for the same path condition. This is a typical problem in coverage-oriented test generation, and is often countered by applying minimization as a post-processing step (e.g., [20]).

Algorithm 1 first solves all augmented path conditions. In principle, a further optimization can be added to check if previous test cases already satisfy a new augmented condition. In practice, however, we achieved a higher speedup by simply parallelizing Algorithm 1, because many augmented path conditions turn out to be unsatisfiable. The satisfiable solutions are then minimized using a simple heuristic: Solutions are sorted descending by the length of the path conditions from which they were obtained. Next, sequentially, each of these solutions is evaluated against the other augmented path conditions. If it turns out that a given augmented path condition is satisfiable with a previous solution, the solution obtained directly from this path condition is discarded. As a result, we obtain a minimized set of solutions satisfying the same augmented path conditions; this set is represents the test suite returned to the user.

When dealing with path conditions, each condition in the source code may occur several times in one path condition. In particular, loop guards may be repeated many times. For example, consider the following loop:

```
while ( x  !=  0 ) {
  list.add( x % 10 );
  x = x / 10;
}
```

Assuming the loop unrolled exactly 3 times the pc will be a conjunction of conditions as follows:

$$\begin{aligned} x \neq 0,\ list[0] &= x \bmod 10, \\ x/10 \neq 0,\ list[1] &= x/10 \bmod 10, \\ x/10/10 \neq 0,\ list[2] &= x/10/10 \bmod 10, \\ x/10/10/10 &= 0 \end{aligned}$$

Here, one division operator occurred in conditions 9 times (3^2) after 3 unrollings and all these occurrences may be targets for augmentation, which could lead to scalability problems. In principle, one can avoid augmenting duplicate occurrences of the same code location by incorporating ADSE directly into the underlying DSE engine, such that it has the required mapping of path conditions to source code statements. Based on this information, one can avoid to augment more than one condition corresponding to the same source code statement, if desired.

4 Augmentation Criteria

We now instantiate the transformation criterion for different common testing objectives, such as boundary value analysis, mutation testing, or general logical coverage criteria.

4.1 Boundary Value Testing

For a given path condition, any input values satisfying this condition will follow the represented path and will thus exhibit the same behavior. DSE therefore

assumes that it is sufficient to test each path only once, and it does not matter which value out of the domain for this path is chosen. However, this only holds under the assumption that there is a complete specification that can decide correctness for every path. From the test suite point of view, a single representative per path is sufficient to test for *functional* faults, but not to test for *domain* faults [25].

For example, if the condition in Line 2 of Figure 1 is wrongly implemented and x should range from 2 to 7 instead of 3 to 7, then a test suite derived with DSE might not be able to detect this fault. Similarly, if such a test suite is used for regression testing, then changes in the value domains would not be detected.

An idea that was proposed early on [25] and has been focus of research over the years (e.g., [17,18]) is that testing should focus on values around the boundaries of domains. Using boundary values instead of whatever the underlying constraint solver suggests has the potential to increase the regression fault sensitivity of a test suite produced with DSE, and it also has the potential to make test cases easier to understand, as in many cases values obtained would have an obvious relation to conditions in source code.

There are several different ways to derive boundary values using ADSE. A simple method to derive values at the boundaries of relational comparisons is to augment relational conditions as follows (e.g., [18]):

$$\begin{array}{lll} A = B \rightarrow A = B & A \leq B \rightarrow A = B & A < B \rightarrow A = B - 1 \\ A \neq B \rightarrow A = B - 1 & A \geq B \rightarrow A = B & A > B \rightarrow A = B + 1 \end{array} \tag{1}$$

Boundary value analysis typically requires not only a value at the boundary, but also representative values from somewhere within the domain. This can be achieved by either selecting different values instead of 1 in the above transformation, or by additional conditions as follows:

$$\begin{array}{ll} A \leq B \rightarrow A < B & A < B \rightarrow A < B - 1 \\ A \geq B \rightarrow A > B & A > B \rightarrow A > B + 1 \end{array} \tag{2}$$

Boundary analysis usually also requires to use values on the other side of a boundary, i.e., values outside the target domain. Boundary value analysis thus leads to weakly augmented path conditions, such that we include the original path conditions to ensure there are representative values for all branch conditions.

4.2 Mutation Testing

Mutation testing [7] is a technique where simple syntactic changes (mutations) are applied to the code in order to simulate faults. The main application of this is to quantify the sensitivity of a test suite with respect to changes in the code, and thus to estimate its effectiveness at detecting real faults. However, mutants can also be used to drive test generation. In particular, by mutating constraints we can create inputs that weakly kill mutants [14]; i.e., the state is changed locally after the mutated statement. Mutants are based on actual

fault models and thus should be representative of real faults, and experiments have confirmed that generated mutants are indeed similar to real faults for the purpose of evaluating testing techniques [3]. The estimation of effectiveness is quantified in the mutation score, which is the ratio of detected (killed) mutants to mutants in total. Mutants that survive the test cases offer guidance in where the test suite needs improvement.

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.

In Apex, we have implemented the following mutation operators:

ROR, LOR, AOR: *Relational/Logical/Arithmetic operator replacement* (3 different mutation operators) replace respectively a relational, logical and arithmetic operator with all its other variants.

UOI: *Unary operator insertion* inserts increment, decrement, negation, and bitwise complement operators to variables and constants.

CRO: *Constant replacement operator* replaces variables and arbitrary constants with constants 0, 1 and -1.

For example, Line 2 consists of a conjunction of two comparisons; the LOR operator would replace the `&&` with `||`, `xor` and ROR would replace `x <= 7` with `x < 7`, `x == 7`, `x >= 7`, etc. UOI would result in `(x+1) <= 7` and all other variations. Given a mutation operator M, the augmentation should ensure that for a given condition C there is a value for each of the mutants $C' \in M(C)$ that distinguishes between C and C'. Therefore, a weak augmentation function for condition C simply is to join mutant and original condition with an xor, such that a resulting test input evaluates different for C and C':

$$\mathcal{T}(C) = \{C \oplus \neg C' \mid \forall C' \in M(C)\} \tag{3}$$

4.3 Logical Coverage

The third instantiation of ADSE we consider in this paper is on coverage criteria for logical predicates, as common in source code. Note that DSE tools operating on the byte-code such as Symbolic JPF [2] or Pex do not need to treat complex logical predicates, as usually complex predicates in source code are compiled to nested atomic predicates in byte-code. However, if the instrumentation for DSE is done on the source code (e.g., [21]) or it is done on a model level (e.g., [16]) then complex predicates can exist and need to be covered.

The transformation of logical predicates to conditions that enforce test generation for different coverage criteria is well studied in other domains, e.g., when representing coverage criteria as temporal logic predicates for model checking [8]. To apply this to our context, we adopt the notation used in [1]: A logical *predicate* (branch condition) consists of *clauses* conjoined with logical operators. Without augmentation, the result of DSE is thus a test suite satisfying predicate coverage.

We consider general active clause coverage (GACC), which is a version of MCDC [6]. It requires that for each clause in a logical predicate there exists a state such that the clause determines the value of the predicate, and the clause has to evaluate to true and to false. For example, the branch in Line 3 in Figure 1 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 predicate only if the other one evaluates to false. In general, a clause C determines a predicate P if the following xor-expression is true, where $P_{C,x}$ denotes P with C replaced with x: $P_{C,True} \oplus P_{C,False}$

Consequently, the transformation of a predicate P requires that for every clause $C \in P$ we add a condition such that P is true and C determines the outcome of P:

$$\mathcal{T}(P) = \{P \wedge (P_{C,True} \oplus P_{C,False}) \mid C \in P\} \tag{4}$$

Note that GACC tests always come in pairs, as the MCDC definition requires that the clause C has to be shown to make P evaluate to true and to false. We assume that the DSE exploration will lead to application of the transformation to both P and $\neg P$. In practice, this means that the above transformation will usually achieve that C evaluates to true and to false. Theoretically, however, simply using determination to augment the condition might also lead to a pair of tests where a clause evaluates to the same value in both cases, yet the determined outcome of the condition differs. This can be overcome by considering the values of individual clauses across transformations as will be described below.

In contrast to GACC, General Inactive Clause Coverage (GICC) requires a clause to *not* determine the outcome of the predicate, i.e., changing the value of the clause does not change the value of the predicate. For example, GICC requires that each of the two clauses in the predicate in Line 3 in Figure 1 has to evaluate to true and to false without changing the outcome of the predicate. If the outcome of the predicate is true, then that means the other clause has to be true; if the predicate is false, then in this example there is no way a clause can not determine this outcome. GICC is also known as Reinforced Condition/Decision Coverage [24]. The following transformation creates a pair of augmented conditions for every clause in a predicate:

$$\begin{aligned}\mathcal{T}(P) = \{P \wedge C \wedge \neg(P_{C,True} \oplus P_{C,False}) \mid C \in P\} \cup \\ \{P \wedge \neg C \wedge \neg(P_{C,True} \oplus P_{C,False}) \mid C \in P\}\end{aligned}$$

This transformation again uses the determination function to require the predicate P to evaluate to true with clause C evaluating to true, and also with C evaluating to false, such that C does not determine P. Again, GICC also requires that the same is also shown for $\neg P$, and we assume that $\neg P$ is also augmented as part of the DSE exploration.

There are stronger versions of GACC and GICC, which require that the other clauses in the predicates do not change their values for any given pair of tests for a clause C. This can be achieved by keeping track of values across

transformations. For example, if we know that clause C evaluated to true in the first of a pair of GACC tests, then in the second we can add the condition $C = false$. In fact, when adding this requirement the resulting test set will satisfy the stricter CACC (Correlated Active Clause Coverage) criterion, which requires that the considered clause determines the outcome of a predicate, and both the predicate and the clause evaluate to true and false.

4.4 Error Conditions

The fourth instance of ADSE we consider in this paper is that of error conditions: While many expressions can be efficiently tested in terms of the explicitly listed conditions on the value ranges, there are often implicit conditions that can lead to erroneous behavior. For example, if the expression x/y is part of a condition, then if $y = 0$ the outcome of the expression is division by zero fault, while for any other value the expression is valid. There have been attempts to make such implicit error conditions explicit to allow test generation tools to cover these cases (e.g., [4, 19]), Pex makes these branches explicit [23], and SAGE includes such conditions in the properties it checks for [11].

If the underlying symbolic execution engine does not make such branches explicit (e.g., Pex or SAGE make them explicit, but other tools might not), then error conditions can be explicitly enforced using ADSE. For example,

$$\mathcal{T}(P) = \begin{cases} \{P\} & \textit{if there are no divisions in } P \\ \{P \wedge x = 0 \mid \forall x : \text{divisors in } P\} \cup & \\ \{P \wedge x \neq 0 \mid \forall x : \text{divisors in } P\} & \textit{otherwise.} \end{cases} \tag{5}$$

In a similar way, other error conditions can be used to augment path conditions. For example, every arithmetic condition can be augmented to a version where there is an overflow and one where there is none; every array access with a non-constant value can be augmented to one version where the index is within the range, and one where it is out of range; every pointer access can be augmented to a version where the pointer is `null` and one where it is not.

5 Evaluation

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 and mutation analysis. In our evaluation we aim to determine how augmentation with boundary value and mutation analysis affects the fault detection ability, test suite size, the number of conditions that need to be solved, and time required for computation.

Table 1. Mutants killed using DSE and augmented with boundary value and mutation analysis

Averages: DSE: 59.22%, Boundary Augmentation: 67.90%, Mutation Augmentation: 75.29%

Function	DSE	Bounds	Mutation	Function	DSE	Bounds	Mutation
Factorial	87.10%	92.86%	93.88%	FindMiddle	58.65%	64.69%	74.60%
Power	61.35%	85.66%	90.91%	WrapInc	76.32%	87.50%	95.00%
MaxValue	68.33%	88.28%	89.37%	Remainder	58.87%	74.97%	82.92%
Fibonacci	84.76%	87.80%	89.02%	ToOctal	28.71%	33.16%	40.77%
GCD	41.74%	41.04%	58.07%	ToHex	51.97%	61.03%	70.11%
WBS	27.76%	26.28%	28.27%	Roops avg.	65.08%	71.52%	90.56%

5.1 Experimental Setup

Apex is based on of Pex, but needs to access path conditions and change the test generation strategy that is applied, which is not possible through the public Pex extension interface. Without modifying Pex itself, we implemented Apex in terms of a Pex extension that collects the path conditions Pex uses to generate tests, and external tool processing them in order to derive tests. The resulting Apex prototype can automatically generate unit test suites using ADSE, requiring nothing but the byte-code of the unit under test. Apex leverages PexWizard, custom Pex-extensions and external tools written in C# to fulfill its tasks.

We ran experiments on 11 standalone methods and one class with multiple methods; Factorial, Power, MaxValue, Fibonacci, GCD (GreatestCommonDenominator), Remainder, ToOctal and ToHex are taken from DSA[4], WBS is taken from [22], and FindMiddle and WrapRoundCounter are examples used in [9]; Roops avg.[5] denotes results averaged over a set of 44 methods of integer examples contained in the Roops test generation benchmark. In our experiments, we consider the boundary value and mutation transformations in detail. We do not consider logical coverage as there are only few complex branch conditions in the byte-code-based symbolic execution performed by Pex, and we also do not consider error conditions as Pex already makes these explicit.

To determine the effectiveness of ADSE we applied the boundary value and mutation testing transformations to all our examples, and measured the resulting mutation scores in both cases, as well as the resulting number of test cases, path conditions solved and time elapsed. For boundary value analysis, we generated conditions to derive values directly at the boundaries and using a representative value.

5.2 Results

Table 1 lists the mutation scores achieved with ADSE against mutants of path conditions, as described in Section 4.2; in almost all cases there is a clear improvement over the mutation scores of the branch-coverage test set produced by

[4] http://dsa.codeplex.com/
[5] http://code.google.com/p/roops/

Table 2. # of tests of augmented test suite / # of path conditions solved using boundary value and mutation analysis, and time

Function	Tests/Conditions			Time		
	DSE	Boundary	Mutation	DSE	Boundary	Mutation
Factorial	4/4	6/11	8/98	<1s	<1s	<1s
Power	4/4	10/31	15/286	<1s	<1s	1s
MaxValue	5/5	11/38	15/367	<1s	<1s	1s
Fibonacci	6/6	7/15	9/164	<1s	<1s	<1s
GCD	4/4	4/17	27/1,004	<1s	<1s	2m54s
WBS	18/18	101/174	365/2,508	<1s	2s	1m48s
FindMiddle	11/11	25/180	71/2,240	<1s	<1s	19s
WrapInc	2/2	4/6	6/40	<1s	<1s	<1s
Remainder	11/11	33/139	64/1,838	<1s	1s	58s
ToOctal	6/6	18/301	71/14,524	<1s	16s	35m59s
ToHex	14/14	24/382	101/11,395	<1s	9s	12m27s
Roops avg.	2.05/2.05	2.36/4.68	5.50/94.39	<1s	<1s	<1s
Average	7.25/7.25	20.45/108.22	63.13/2,879.87	<1s	2s	4m32s

Pex. Boundary value testing leads to lower mutation scores for GCD and WBS; for these examples there seem to be more mutants that are not related to boundary values, such that the boundary values represent an unlucky choice. There is of course a potential bias in these results as the mutants used for evaluation are the same as used to produce test cases. However, the aim of this experiment is not to evaluate different coverage criteria, but to demonstrate the general feasibility of ADSE, and that Apex can satisfy the augmented path conditions to a high degree.

In our experiments, ADSE resulted in up to 30% higher mutation scores.

Table 2 reveals where the increase in mutation score comes from: The number of tests produced is considerably higher than in Pex's branch-coverage test sets. The table shows the size of test suites produced by augmentation and number of path conditions that were solved.

In our experiments, ADSE increased the average number of tests by up to 9 times over branch coverage.

Constraint solving can be very expensive, and as the augmentation increases the number of conditions that need to be solved, this can be problematic with respect to the scalability of the approach. Table 2 summarizes the number of conditions that were passed to the constraint solver with and without augmentation (this includes the conditions that are solved as part of the regular DSE exploration), and the right hand side of Table 2 summarizes the time it took to emit the augmented test suites, running on 7 out of 8 logical cores of Intel(R) Core(TM) i7-2675QM CPU @ 2.20GHz processor and 8 GB RAM. The increase is significant, and depends on the actual transformation used. Little surprising, mutation analysis leads to the largest number of augmented path conditions, and thus also significantly increases the effort that goes into test generation.

The fact that mutation testing has scalability issues is well known, and there are ways to improve the performance, primarily by more fine-grained control over path conditions. Other techniques include summarizing loops [13], which directly addresses the scalability problem with loop unrollings (see 3.3), or sampling mutants instead of exhaustively considering all of them. Such techniques can theoretically be applied to ADSE as well. Furthermore, there are techniques to reduce the number of calls to the constraint solver (e.g., [11]).

The ToOctal example has the largest increase in the number of path conditions, and this and ToHex are instances of the loop unrolling problem described in Section 3.3. As our current prototype is not directly integrated into Pex it does not have access to a mapping between path conditions and code, making it impossible to avoid this problem.

On average, ADSE lead to a modest increase in computational time, but the worst case was over 2160 times slower than normal DSE.

5.3 Threats to Validity

The focus of this evaluation is to demonstrate that ADSE can result in better test suites, where the meaning of "better" can be defined by any transformation criterion. There are several threats to the validity of these experiments:

Threats to *construct validity* are on how the performance of a testing technique is defined. We measured the quality of the resulting test sets in terms of their mutation score on path conditions, not the source code. However, in practice developers might have different preferences such as whether the chosen values are similar to values a human tester might have chosen. Furthermore, an increase in mutation score might not be desirable if it comes with a significant increase of the test suite size. The mutation score in our experiments was computed by the same tool used to generate mutation augmented test suites, so the results may be biased. As we used a standard set of mutation operators it is likely that other mutation analysis tools would result in similar scores; however, in general quantifying the relation between different coverage criteria is not the objective of this paper.

Threats to *internal validity* might come from how the empirical study was carried out. To reduce the probability of having faults in our Apex prototype, it has sizable unit test suite and makes heavy use of Code Contracts. The size of the chosen example functions means that there is a threat to *external validity* regarding the generalization to other types of software. The main limitation enforcing our choice of case study subjects was that our approach is not fully implementable through the public Pex extension API.

6 Conclusions

DSE 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 instantiated this augmented dynamic symbolic execution for boundary value analysis, mutation analysis, logical coverage criteria, and error conditions, demonstrating that the resulting test suites are superior at detecting faults to the simple branch coverage test suites produced by Pex.

There are several immediate applications of ADSE: First, a common application reported by Pex users is to simply explore the behavior of a function or program. In this case, the programmer simply looks at the test data produced by Pex, and can thus profit from a wider range of interesting cases. For example, if there is a predicate on $5 \times x == y \times z$ then a valid assignment (and the one chosen by Pex) would be to assign 0 to x, y, and z. Mutation analysis or boundary values would lead to different (more interesting) values.

The second application is a traditional scenario in automated testing when there is no automated oracle available. Here, the aim is often to produce test sets that satisfy a given criterion, for example a logical coverage criterion.

A third application scenario 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 offer to improve regression test suites.

ADSE also has the potential to extend DSE in many ways not immediately targeted by the transformations we described in this paper. Augmenting comprehensibility by transforming to human-readable values (e.g., for strings) or performance testing are just two possible augmentations.

Finally, in our evaluation we demonstrated that augmentation can lead to test suites satisfying different criteria. However, we have not yet investigated which criteria are most useful in practice. For example, the mutation testing experiments indicate scalability problems, which might justify the use of sampling techniques to generate stronger test sets without expensive constraint solving on too many individual mutants. Furthermore, our current prototype is implemented as an extension to Pex, whereas a tighter tool integration would offer further opportunities for optimization. This and other improvements will be the focus of our future work.

Acknowledgments. Thanks to Florian Gross for comments on earlier versions of this paper. This project has been funded by DFG grant Ze509/5-1

References

1. P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 1 edition, 2008.
2. S. Anand, C. S. Păsăreanu, and W. Visser. JPF-SE: a symbolic execution extension to Java PathFinder. In *Proc. TACAS'07*, pages 134–138. Springer-Verlag, 2007.
3. J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proc. ICSE '05*, pages 402–411. ACM, 2005.
4. N. Bhattacharya, A. Sakti, G. Antoniol, Y.-G. Guéhéneuc, and G. Pesant. Divide-by-zero exception raising via branch coverage. In *Proc. SSBSE*, pages 204–218. Springer, 2011.

5. C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Commun. ACM*, 56(2):82–90, Feb. 2013.
6. J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, pages 193–200, 1994.
7. 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.
8. G. Fraser, F. Wotawa, and P. E. Ammann. Testing with model checkers: a survey. *Softw. Test. Verif. Reliab.*, 19:215–261, September 2009.
9. K. Ghani and J. A. Clark. Strengthening inferred specifications using search based testing. In *Proc. SBST*, pages 187–194. IEEE Computer Society, 2008.
10. P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *Proc. PLDI*, pages 213–223, 2005.
11. P. Godefroid, M. Y. Levin, and D. A. Molnar. Active property checking. In *Proc. EMSOFT '08*, pages 207–216. ACM, 2008.
12. P. Godefroid, M. Y. Levin, and D. A. Molnar. Sage: Whitebox fuzzing for security testing. *ACM Queue*, 10(1):20, 2012.
13. P. Godefroid and D. Luchaup. Automatic partial loop summarization in dynamic test generation. In *Proc. ISSTA '11*, pages 23–33. ACM, 2011.
14. W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE TSE*, 8(4):371–379, 1982.
15. K. Jamrozik, G. Fraser, N. Tillmann, and J. De Halleux. Augmented dynamic symbolic execution. In *Proc. ASE 2012*, pages 254–257. ACM, 2012.
16. E. Joebstl, M. Weiglhofer, B. K. Aichernig, and F. Wotawa. When bdds fail: Conformance testing with symbolic execution and smt solving. In *Proc. ICST '10*, pages 479–488. IEEE Computer Society, 2010.
17. N. Kosmatov, B. Legeard, F. Peureux, and M. Utting. Boundary coverage criteria for test generation from formal models. In *Proc. ISSRE*, pages 139–150. IEEE Computer Society, 2004.
18. R. Pandita, T. Xie, N. Tillmann, and J. de Halleux. Guided test generation for coverage criteria. In *Proc. ICSM '10*, pages 1–10. IEEE Computer Society, 2010.
19. D. Romano, M. Di Penta, and G. Antoniol. An approach for search based testing of null pointer exceptions. In *Proc. ICST '11*, pages 160–169. IEEE, 2011.
20. G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proc. ICSM '98*, pages 34–. IEEE Computer Society, 1998.
21. K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Proc. ESEC/FSE-13*, pages 263–272. ACM, 2005.
22. M. Staats and C. Păsăreanu. Parallel symbolic execution for structural test generation. In *Proc. ISSTA'10*, pages 183–194. ACM, 2010.
23. N. Tillmann and N. J. de Halleux. Pex — white box test generation for .NET. In *Proc. TAP*, pages 134–153, 2008.
24. S. A. Vilkomir and J. P. Bowen. Reinforced condition/decision coverage (rc/dc): A new criterion for software testing. In *Proc. ZB '02*, pages 291–308. Springer, 2002.
25. L. J. White and E. I. Cohen. A domain strategy for computer program testing. *IEEE Trans. Softw. Eng.*, 6:247–257, May 1980.
26. L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. de Halleux, and H. Mei. Test generation via dynamic symbolic execution for mutation testing. In *Proc. ICSM '10*, pages 1–10. IEEE Computer Society, 2010.