Automated Test Generation for Object-Oriented Programs with Multiple Targets

Hiroki Takamatsu, Haruhiko Sato, Satoshi Oyama, Masahito Kurihara

***Abstract*—Software testing is costly. In particular, testing object-oriented programs is complicated and burdensome because of the difficulty in generating method sequences for creating objects and changing their states appropriately to achieve high branch coverage. Automated test generation based on static and dynamic analysis is not only an effective approach to saving time and reducing the burden of testing, but also an efficient way to find bugs.**

Seeker is an implementation for automated test generation that involves the generation of method sequences using static and dynamic analysis. However, when we want to change some values of variables to increase the branch coverage, the system fails to generate method sequences to achieve the desired states of objects with multiple targets (meaning fields or variables).

In this paper, we extend the functionalities of the system for automated test generation so that it can handle multiple targets to cover. Our approach identifies all targets involved in uncovered branches and evaluates method sequences according to a fitness function, while applying a search strategy to suppress combinatorial explosion. The experimental results with several open source projects show that our extension achieves higher branch coverage than the original system and the effectiveness of the extension tends to vary according to the specific features of the projects.

***Index Terms*—Automated test generation, Dynamic Symbolic Execution, Method sequence, Branch coverage**

I. Introduction

SOFTWARE testing, the process of executing a program with the intent of finding errors, is an important process in software development for building high reliability systems. However, software testing is often too costly for practitioners to spare sufficient time for. This is the case particularly when it comes to testing object-oriented programs, because when testing object-oriented programs, we have to not only supply appropriate arguments to the method calls but also insert into the code a sequence of method calls to change the states of the objects appropriately before verifying testing conditions. This motivates the reduction of the burden of software testing, and in order to achieve such a goal, automated test generation has become an active research field in the software engineering community. An example is the Seeker [1], an implementation of automated test generation for object-oriented programs, developed by Thummalapenta, et al. However, the current version of Seeker has a limitation. When we want to modify the values of multiple targets (meaning fields or variables) in order to cover a condition in a certain execution path, it fails to generate test cases that have to involve an appropriate sequence of method calls before testing conditions. In general, branches that involve multiple targets tend to be complicated, and thus, we need to sufficiently test such branches [2]. Hence, testing branches associated with multiple targets is a critical program.

Manuscript received June 30, 2014; This work was supported by JSPS KAKENHI Grant Number 25330074.

H. Takamatsu, H. Sato, S. Oyama and M. Kurihara are with the Graduate School of Information Science and Technology, Hokkaido University, Sapporo, Japan, 060-0814. E-mail: {takamatsu, haru}@complex.ist.hokudai.ac.jp, {oyama, kurihara}@ist.hokudai.ac.jp

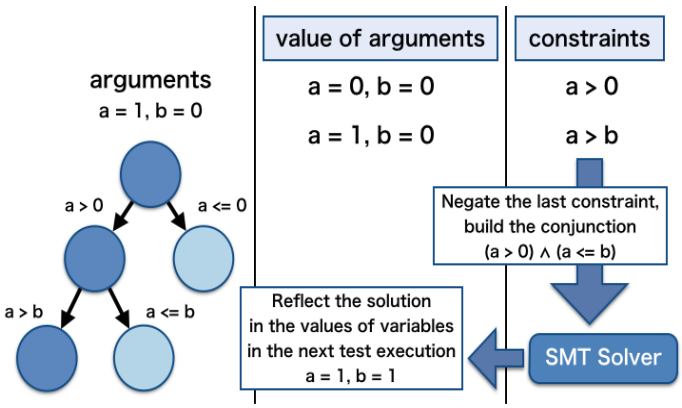

Fig. 1. An Overview of Dynamic Symbolic Execution

In this paper, we present an extension to the functionalities of the Seeker so that it can handle multiple targets in the conditions to cover more branches. The expected combinatorial explosion in the number of candidate sequences of method calls is suppressed by a search strategy based on heuristic evaluation of the candidates.

The rest of the paper is structured as follows. Section II describes the background and related works of automated test generation for object-oriented programs. In Section III, we describe the existing approach and its problems. In Section IV, we explain the idea of the proposed method. In Section V, we present our experiments and the results and in Section VI, we describe our conclusions and discuss future work.

II. Background and Related Works

A. Dynamic Symbolic Execution

Dynamic Symbolic Execution (DSE) [3], [4] is a state-of-the-art automated test generation technique. DSE, also called *Concolic testing* [5], [6], combines tests with *conc*rete values and symb*olic* execution [7]. Figure 1 shows an overview of DSE. DSE applies concrete and symbolic execution alternately. Given a program and a method to be tested, DSE first insert into the program the necessary fragments of code to log the changes of values of variables and the events to be raised in the execution, and then by supplying random values for the arguments, it executes the program and the method in order to collect execution traces. From those traces, DSE collects the constraints (conditions) which, in conjunction, uniquely identify a certain path that was executed. Then DSE modifies the set of those constraints by negating one of them (typically, the one that is most apart from the root of the execution tree) in order to specify yet another path, say p,

Source code 1. Graph Class

```
1  class Graph {
2    private ArrayList<Edge> edges;
3    private ArrayList<Node> nodes;
4
5    public void AddNode(Node n) {
6        if (n == null) throw new Exception();
7        nodes.Add(n);
8    }
9    public void AddEdge(Node source, Node target
         ){
10       if (nodes.Contains(source) &&
11           nodes.Contains(target)) {
12           // Case2 reach here
13           edges.Add(source, target);
14       } else {
15           // Case1 reach here
16           throw new Exception();
17       }
18   }
19   ...
20 }
```

Source code 2. Example Test Cases for Graph#AddEdge()

```
1  // Case1: A simple test case
2  Graph graph = new Graph();
3  graph.AddEdge(null, null);
4  // assert something
5
6  // Case2: A test case with method sequence
7  Graph graph = new Graph();
8  Node s1 = new Node();
9  Node s2 = new Node();
10 graph.AddNode(s1);
11 graph.AddNode(s2);
12 graph.AddEdge(s1, s2);
13 // assert something
```

to be tested. The modified set of constraints is then passed to a standard SMT (Satisfiability Modulo Theory) solver [8], which solves a satisfiability problem with a built-in theory (such as the theory for integer arithmetic). The solution, if any, represents a new set of values to the variables to ensure the execution of the intended new path p in the next test run. Repeating this procedure, DSE can efficiently identify the sets of values for the variables to test a collection of different execution paths to increase the branch coverage.

B. Method Sequence

Software testing for modern programs, in particular object-oriented programs written in, say, C# or Java, often requires a sequence of method calls (in short, *method sequence*) to obtain desired object states [9], [10], [11]. In other words, to achieve high coverage, it is necessary that the states of the objects specified as a receiver or an argument be turned into appropriate states that would meet the testing condition. This means that programmers (or automated test generation systems) have to insert into the code a method sequence to create and transform objects before calling a method to be tested.

For example, suppose we want to test a method `AddEdge`, which adds an edge to a graph in Source code 1. The goal is to achieve full coverage of `AddEdge`. When we execute Case1 in Source code 2, we reach the line 16. The program raises an exception because the graph has no nodes. To test the ability of the operation to add an edge to a graph, we must add method calls to add nodes to a graph in advance, such as Case2 in Source code 2. When we execute Case2 in Source code 2, we reach the line 13. Now we have been able to cover a code block different from Case1 execution. This implies that the block was only reachable by test cases with a proper method sequence.

In this manner, if automated test case generation tools fail to consider the generation of method sequences, the generated test cases will not cover sufficiently many code blocks of object-oriented programs. This implies that to test the behavior of object-oriented programs satisfactorily, it is often necessary to properly generate method sequences to set appropriate conditions before testing.

III. Seeker

Seeker is one of the several novel implementations for automated test generation with method sequences on C#. It uses Pex as the engine for DSE. Seeker is based on the technique of DSE and generates test cases using dynamic and static analyses of programs. Our approach is based on Seeker's algorithm, which we illustrate in this section.

Seeker takes a target method to generate test cases as an input and finally outputs test cases with a method sequence. Seeker repeatedly applies static and dynamic analyses to the target program. In each step, Seeker grows the method sequences and reduces candidates of method sequences that do not contribute to increasing the coverage. In the following, we will provide an overview of Seeker and then briefly describe the main components. Seeker works as follows:

1) Seeker generates a primitive test case that calls only the method under testing.
2) It generates test cases that cover respective paths by changing arguments in DSE.
3) For uncovered branches in the previous step, it detects the variable the values of which should be changed in order to cover the branch by analyzing execution traces. The variable is referred to as the *target field*.
4) It identifies the relation (e.g. inheritance, comprehension) of the class that includes the target field.
5) From the relation identified in 4), it finds methods that can change the value of the target field.
6) If the branch remains uncovered, Seeker adds candidate methods to the existing method sequences, then returns to 2) and repeats the process. Otherwise the process ends.

A. Dynamic Analysis

When we generate test cases for a certain method under testing, we first apply DSE to the target program. DSE generates many test cases that cover paths individually and also returns covered and uncovered branches. During exploration by DSE, execution traces are collected to be analyzed in each static analysis step. Algorithm 1 shows the pseudocode of Dynamic Analysis.

B. Static Analysis

In the static analysis phase, Seeker uses, as inputs, the uncovered branches and the executed method sequences in the previous dynamic analysis phase. Then, the program detects the variable the values of which should be changed to satisfy the constraints of the uncovered branches (target field). Therefore, it finds the dependency of the class including the target field. From the identified dependency, it

Algorithm 1 DynamicAnalysis

Require: tb of TargetBranch (TB)
Require: $inputSeq$ of MethodSequence (MSC)
Ensure: $targetSeq$ of MethodSequence (MSC) or null
1: Method m = GetMethod(tb)
2: MSC $tmpSeq$ = AppendMethod($inputSeq, m$)
3: DSE($tmpSeq$, tb, out $tSeq$, out $covBranch$, out $uncovBranch$)
4:
5: **if** $tb \in covBranch$ **then**
6: **return** targetSeq
7: **end if**
8:
9: **if** $tb \in uncovBranch$ **then**
10: **return** StaticAnalysis($tb, inputSeq$)
11: **end if**
12:
13: **if** $tb \notin uncovBranch$ **then**
14: List<TB>$tbList$ = ComputeDominants(tb)
15: **for all** TB $domiBranch \in tbList$ **do**
16: $inputSeq$ = DynamicAnalysis($domiBranch, inputSeq$)
17: **if** $inputSeq == null$ **then**
18: break
19: **end if**
20: **end for**
21: **if** $inputSeq \neq null$ **then**
22: **return** DynamicAnalysis($tb, inputSeq$)
23: **end if**
24: **end if**
25: **return** null

Algorithm 2 StaticAnalysis

Require: tb of TargetBranch (TB)
Require: $inputSeq$ of MethodSequence (MSC)
Ensure: $targetSeq$
1: Field $targetField$ = DetectField(tb)
2: List<TB>$tbList$ = SuggestTargets($targetField$)
3: **for all** TB $prevTb \in tbList$ **do**
4: MSC $targetSeq$ = DynamicAnalysis($prevTb, inputSeq$)
5: **if** $targetSeq \neq null$ **then**
6: $targetSeq$ = DynamicAnalysis(tb, $targetSeq$)
7: **if** $targetSeq \neq null$ **then**
8: **return** $targetSeq$
9: **end if**
10: **end if**
11: **end for**

builds a hierarchy of fields (referred to as the *field hierarchy*). Finally, it extracts candidates of methods that may mutate the value of the target field and add them to the existing method sequences. Algorithm 2 shows the pseudocode of static analysis. Next we describe important components in static analysis.

1) Detection of Target Field: In the target field detection step, Seeker detects the variable that must be changed in order to cover the uncovered branches in the last DSE. Detecting target field might seem trivial, but there is difficulty in many cases. For example, it is straightforward to identify the target field for branches such as `if (list.size > 0)`, because the variable `size` is a public member of the instance `list` of some container class. When the target field is a public member, we can modify the value directly. However, we often find branches that involve method calls such as `if (graph.ComputeDistance() > 10)`. Then we must analyze what the statement returns and these methods may include further method calls, causing the step of detecting the actual target field to be a complicated task.

2) Field Hierarchy: In the next step, Seeker builds from the execution traces a field hierarchy that represents the hierarchical inheritance relation between classes related to the target field. We can trace the relation to the target field in the field hierarchy. The field hierarchy helps to find the methods that can modify the value of the target field and the classes that are required to invoke these methods. This is used to create a method-call graph mentioned later.

For example, when the target field is a member `_count` in `Arraylist` class, the field hierarchy is `Stack list` → `ArrayList _count` in Source code 3.

3) Method-Call Graph: A *method-call graph* is a graph that represents the caller and callee relation of the methods that may modify the value of the target field and the classes

Source code 3. An Overview of Stack Class

```
1  class Stack {
2      private ArrayList<int> list;
3      public int Push(int element) {
4          list.Add(element);
5      }
6      ...
7  }
8  class ArrayList {
9      private int _count;
10     private int[] _elements;
11     public int Add(int element) {
12         // something to add element
13         ...
14
15         _count++;
16     }
17     ...
18 }
```

that have these methods. Seeker creates the graph using the field hierarchy, the code analysis and the execution trace analysis. The root node is the target field and the level one nodes are the methods that can change the value of target field directly. In the remaining levels of nodes, the methods of parent nodes are called by the methods of child nodes. The terminal nodes are the public methods that can change the target field. Seeker appends these methods to the existing method sequences. In this way, the generated new method sequences may convert a target field to the desired state and cover the target branches.

For example, when the target field is a member `_count` in `Arraylist` class, Then, the root node is the target field `_count` and one of the level one nodes is `Add()` in `ArrayList`. One of the terminal nodes is `Push()` in `Stack` that has `Add()` in `ArrayList` as its parent.

IV. Approach

We have shown the difficulty in automated test generation considering object states and the existing approach to overcome this problem by generating method sequences. However, some problems remain. One of these is difficulty in converting multiple object states into desired states in object-oriented programs. In fact, Seeker cannot cover branches that require the change of multiple objects to desired states. Furthermore, the branches associated with multiple variables tend to be complicated and thus require a significant amount of comprehensive testing. This implies that the testing of branches associated with multiple variables is a critical problem for increasing the branch coverage. In this section

Algorithm 3 StaticAnalysisForMultiTargetFields

Require: tb of TargetBranch (TB)
Require: $inputSeq$ of MethodSequence (MSC)
Ensure: $targetSeq$
1: List<Field>$targetFields$ = DetectAllFields(tb)
2: List<TB>$tbList$ = new List<TB>
3: **for all** Field $targetField \in targetFields$ **do**
4: $tbList$.Append(SuggestTargets($targetField$))
5: **end for**
6: **for all** TB $prevTb \in tbList$ **do**
7: MSC $targetSeq$ = DynamicAnalysis($prevTb, inputSeq$)
8: **if** $targetSeq \neq null$ **then**
9: calculateEvaluationValue($targetseq$)
10: **if** isCandidate($targetseq$) **then**
11: $targetSeq$ = DynamicAnalysis($tb, targetSeq$)
12: **if** $targetSeq \neq null$ **then**
13: **return** $targetSeq$
14: **end if**
15: **end if**
16: **end if**
17: **end for**

we will focus on this problem and propose an approach to solve it.

The algorithm of the previous study [1] identifies only one variable as the target field. In our study, we focus on all variables that influence target branches. Actually, we have improved the algorithm so that it can identify all variables associated with target branches. Since this improvement is conceptually simple and implemented in a straightforward way, we will not present its details. However, you can easily imagine that combinatorial explosion will occur because of the combinations of all method calls to achieve conversion of all variables to the desired states. To suppress the explosion, we have introduced a technique of static analysis for multiple targets as shown in Algorithm 3. We will briefly describe it in the rest of this section.

A. Static Analysis

The proposed static analysis for multiple targets is based on the algorithm of the previous study [1], which identifies only one variable detected indeterminately as a target. In our study, however, we focus on all variables that influence target branches. Thus, the algorithm identifies these variables as the targets. The field hierarchy and the method-call graph are built for all these variables. Using these structures, our algorithm generates appropriate method sequences that cover target branches with multiple targets.

B. Reduction of Candidates

Considering the combinations of method calls and targets as well as the number of permutations of method calls, you can easily imagine that the number of potential method sequences necessary to transform all variables to the desired states will lead to combinatorial explosion. For example, if there are two targets and if each variable has 10 methods that may modify its value, then the program will generate 20 method sequences of length one. If the target branch was not covered yet, the program would generate 400 method sequences in the next step. Thus the number of candidate method sequences increases exponentially.

In order to suppress the explosion, we evaluate the "unfitness" of the method sequences with an evaluation function called the fitness function. Using some heuristics, it evaluates the "distance" between a constraint and a method sequence. More precisely, given a method sequence, the function determines its evaluation value such that it represents a heuristic cost for reaching the desired object states from the states reached by the method sequence in the current branch at hand. (Note that, in this sense, this function may be called the unfitness function.) In this paper, we define the fitness function only for a 32-bit integer as in Table I. The definition is the same as that used by Xie et al [12]. For other types, we give each method sequence the maximum value of 32-bit interger (214748367) as the unfitness.

TABLE I
DEFINITION OF FITNESS FUNCTION

Expression	True	False
$a == b$	0	$\|\|a - b\|\|$
$a > b$	0	$(b - a) + 1$
$a \geq b$	0	$(b - a)$
$a < b$	0	$(a - b) + 1$
$a \leq b$	0	$(a - b)$

TABLE II
AN OVERVIEW OF TARGET PROJECTS

Project	Version	Classes	Methods	Branches	KLOC
Dsa	0.6	27	308	665	3.3
QuickGraph	1.0	88	634	1119	5.1
xUnit	1.6.1	151	1267	2379	11.9
NUnit	2.5.7	225	2344	1810	8.1

In order to suppress the explosion, the algorithm reduces the number of candidate method sequences based on the unfitness. More precisely, if the number of the generated method sequences is more than the predefined limit, those with higher unfitness (i.e., longer distance to the desired states) are deleted. Users can set the limit according to the available resources and the required quality. In this paper, we set the limit to 100.

Table I defines no evaluation value for expressions with the "not equal" operation. This is because we could not think of a suitable measure to make two values different when they are the same. However, our experience shows that new test cases can often cover such "not equal" branches fairly easily in the next DSE step by modifying the target state, say, randomly. Therefore, we assign 0 to such sequences regardless of their concrete values.

It remains to been seen how the composite conditions will be evaluated. We can identify composite conditions when a branch involves two or more conditions connected by the logical operators such as `||` and `&&`. The DSE engine Pex, which we use in our implementation, processes CIL (Common Intermediate Language), in which composite conditions are split into a standard set of atomic conditions. Thus it is not necessary to extend the definition of the fitness function for composite conditions.

V. EXPERIMENTS AND RESULTS

We have implemented our approach in C# by extending the Seeker on a machine running 32-bit Windows Vista with the 2.53-GHz Intel Core 2 Duo processor with 4 GB RAM. Our settings for the experiments are the same as used in [1] as follows.

TABLE III
BRANCH COVERAGE ACHIEVED BY SEEKER AND PROPOSED METHOD TESTS

Project	Seeker			Proposed Method		
	# of tests	Branch coverage	Time (hours)	# of tests	Branch coverage	Time (hours)
Dsa	961	88.1%	5	1387	91.5%	8.5
QuickGraph	1923	68.2%	8	2694	69.5%	17.3
xUnit	1360	41.1%	6.3	2391	46.8%	8.5
NUnit (Util Namespace)	1804	44.3%	12.8	5125	45.5%	23.3

- timeout: 500 sec (default: 120 sec)
- MaxConstraintSolverTime: 10 sec (default: 2 sec)
- MaxRunsWithoutNewTests: 214748367 (default: 100)
- MaxRuns: 2147483647 (default: 100)

The experiment was conducted in order to evaluate the effectiveness of our method, based on the comparison with the Seeker in terms of the execution time and the branch coverage of test cases that were generated. We used four real-world open source projects listed in Table II, which shows their features including the number of classes, methods, branches and lines of code. Dsa [13] is a library that provides many data structures and algorithms for the .NET framework. Quickgraph [14] is a C# graph library. xUnit [15] and NUnit [16] are widely known unit testing frameworks for .NET languages. In the experiments, we used its core component, the `util` namespace, for NUnit.

A. Experimental Results and Evaluation

Table III shows the results. We can see that the proposed method achieved 1 to 5% improvement in terms of the coverage over the Seeker, although the execution time was sometimes almost doubled. At first sight, this might seem to be a moderate improvement. However, we should put more weight on the novel test cases generated by our method. Seeker generated a lot of test cases with a single target fairly easily, but failed to generate test cases with multiple targets. Intuitively, a lot of difficult, hard-to-find bugs may be involved in such complicated branches with multiple targets. Our emphasis in this paper was how this difficulty could be overcome. In this respect, we should say that what was impossible by Seeker was made possible by our method, while covering the branches already covered by the Seeker.

B. Discussion

For some projects such as Quickgraph, the proposed approach achieved only moderate improvements while taking a fairly amount of time. Quickgraph is a library that provides data structures and algorithms for graphs, and many methods involve various objects such as `Node` or `Edge`. If the project to be tested has a complex dependency among the objects, it often requires so many operations or so long method sequences to cover branches that it tends to require a lot of extra time to achieve even a small branch coverage gain. Quickgraph was such a project.

Other reasons for the moderate improvement are as follows. First of all, branches involving multiple targets occur, in general, infrequently. A lot of branches involve only a single target. This is why the Seeker restricted itself to the single target. However, as noted in the previous subsection, a lot of difficult, hard-to-find bugs may be involved in such complicated branches with multiple targets. This means that although the improvement was quantitatively moderate, it was significant in terms of quality.

Second, we might have pruned desirable candidate method sequences in the candidate reduction step. This can occur in the projects that have complex dependency, like Quickgraph, because such projects cause the combinatorial explosion of method sequences. If we had set a proper limit to the number of reserved method sequences, we could have avoided this problem, but even then a lot of execution time might have been wasted by the exponentially increased number of reserved method sequences. This means that we must determine the limit properly, considering specific requirements for the time and quality of testing.

Finally, the fitness function might not have been powerful enough, because it was defined only for integer types and its heuristics were simple. This means that further research is necessary for extending its domain types and developing more powerful heuristics.

VI. CONCLUSION AND FUTURE WORK

Testing for object-oriented programs requires method sequences, and many approaches have been proposed for generating them for a single target. In this paper, on the other hand, we have extended an existing method, focusing on generating method sequences that mutate multiple targets. We implemented our method on top of the Seeker and evaluated its effectiveness by using four real-world projects. We saw that our system achieved higher coverage of difficult branches than the Seeker, while requiring additional execution time. However, the results also showed that the effectiveness of the proposed approach varied according to the specific features of different projects.

Our future work includes the improvement of the fitness function so that its domain includes a lot of data types other than integers and reflects a useful heuristics in those domains. Another interesting work is the design of the search strategy for generating method sequences so that it can contribute to a significant suppress of the combinatorial explosion. In any work, extensive analyses of covered and uncovered branches would be critical for developing a good strategy to make the automated test generation more successful.

REFERENCES

[1] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and Z. Su, "Synthesizing method sequences for high-coverage testing," *SIGPLAN Not.*, vol. 46, no. 10, pp. 189–206, Oct. 2011. [Online]. Available: http://doi.acm.org/10.1145/2076021.2048083

[2] H. Takamatsu, H. Sato, S. Oyama, and M. Kurihara, "Autoamted test case generation considering object states in object-oriented programming," in *Lecture Notes in Engineering and Computer Science: Proceedings of The International MultiConference of Engineers and Computer Scientists 2014*. IMECS 2014, 12-14 March 2014, Hong Kong, pp. 569–573.

[3] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," *SIGPLAN Not.*, vol. 40, no. 6, pp. 213–223, Jun. 2005. [Online]. Available: http://doi.acm.org/10.1145/1064978.1065036

[4] N. Tillmann and J. Halleux, "Pexwhite box test generation for .net," in *Tests and Proofs*, ser. Lecture Notes in Computer Science, B. Beckert and R. Hhnle, Eds. Springer Berlin Heidelberg, 2008, vol. 4966, pp. 134–153. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-79124-9_10

[5] K. Sen, "Concolic testing," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07. New York, NY, USA: ACM, 2007, pp. 571–572. [Online]. Available: http://doi.acm.org/10.1145/1321631.1321746

[6] K. Sen and G. Agha, "CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, T. Ball and R. Jones, Eds. Springer Berlin Heidelberg, 2006, vol. 4144, pp. 419–423. [Online]. Available: http://dx.doi.org/10.1007/11817963_38

[7] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976. [Online]. Available: http://doi.acm.org/10.1145/360248.360252

[8] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–340. [Online]. Available: http://dl.acm.org/citation.cfm?id=1792734.1792766

[9] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 59:1–59:11. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393666

[10] P. Tonella, "Evolutionary testing of classes," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 4, pp. 119–128, Jul. 2004. [Online]. Available: http://doi.acm.org/10.1145/1013886.1007528

[11] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst, "Combined static and dynamic automated test generation," in *Proceedings of the 11th International Symposium on Software Testing and Analysis (ISSTA 2011)*, 2011.

[12] T. Xie, N. Tillmann, P. de Halleux, and W. Schulte, "Fitness-guided path exploration in dynamic symbolic execution," in *Proceedings of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2009)*, June–July 2009, pp. 359–368. [Online]. Available: http://www.csc.ncsu.edu/faculty/xie/publications/dsn09-fitnex.pdf

[13] DSA, http://dsa.codeplex.com/.

[14] QuickGraph, http://quickgraph.codeplex.com/.

[15] xUnit, http://xunit.codeplex.com/.

[16] NUnit, http://www.nunit.com/.