Automated Test Case Generation Considering Object States in Object-Oriented Programming

Hiroki Takamatsu, Haruhiko Sato, Satoshi Oyama, Masahito Kurihara

***Abstract*—Testing for object-oriented programs is complicated and burdensome. One reason is the difficulty associated with generating method sequences that create instances and change object states to achieve high branch coverage. Automated test generation using mixed static and dynamic analysis is not only an effective approach to save time and reduce the burden of testing, but also an efficient way to find bugs.**

Seeker is an implementation for automated test generation that includes method sequences using static and dynamic analysis. However, when we need to change several values of variables to cover branches, the technique cannot generate method sequences that achieve all of the desired object states. In this paper, we extend the technique for automated test generation when multiple object states are required for the coverage of more complicated branches. Our approach identifies all variables that are involved in uncovered branches and evaluates method sequences according to a fitness function. Then we apply a retrieval strategy to suppress combinatorial explosion. Our results show that the proposed approach achieves higher branch coverage than that used in a previous study and also suggest that the effectiveness of the proposed approach tends to vary according to the specific characteristics of different projects.

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

I. Introduction

SOFTWARE testing is an important process in software development projects for building high reliability systems. In testing for object-oriented programs, we not only need method arguments but also some method calls to change object states to desired values before verifying assertions. However, in many cases, we cannot spare sufficient time for testing due to the cost, and thus, research of new methods for automated test generation is an active field. Seeker [1] is an implementation of automated test generation for object-oriented programs. When we need to modify multiple variables in order to cover a condition in a certain execution path, Seeker cannot generate test cases that include proper method sequences. However, branches that relate multiple variables tend to be complicated, and thus, we need to sufficiently test such branches. Hence testing of branches related to multiple variables is a considerable problem.

In this paper, we present an approach for automated test generation that requires modifying multiple variables in conditions to cover branches, and we implement the technique as an extension of Seeker. We expected a combinatorial explosion in the number of candidate method sequences, so we suppressed it by assigning evaluation values to candidate method sequences.

Manuscript received January 8, 2014; revised January 30, 2014.

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

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 state our conclusions and discuss future work.

II. Background and Related Works

A. Dynamic Symbolic Execution

Dynamic Symbolic Execution (DSE) [2], [3] is a state-of-the-art automated test generation technique. DSE combines tests with concrete values and symbolic execution [4], so that the technique also is called Concolic testing [5], [6]. Figure 1 shows an overview of DSE. In concrete execution, we explore a method under testing and collect constraints from the predicates in branch statements in the method under testing. Then, we negate a part of the collected constraints and attempt to solve it and assign proper values to variables that relate the condition by SMT solvers [7]. Thus, we can execute different paths using these values as test input data.

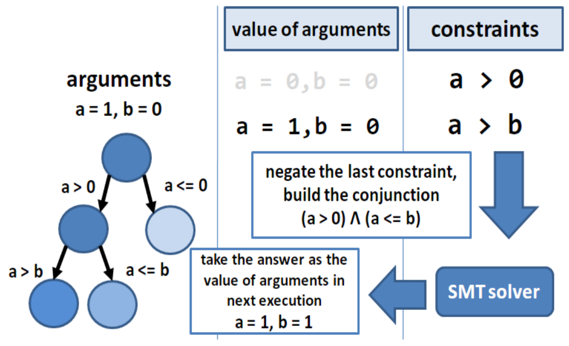
value of arguments
constraints
arguments
a = 1, b = 0
a = 0,b = 0
a > 0
a = 1,b = 0
a > b
a > 0
a <= 0
negate the last constraint, build the conjunction (a > 0) Λ (a <= b)
a > b
a <= b
take the answer as the value of arguments in next execution
a = 1, b = 1
SMT solver

Fig. 1. An Overview of Dynamic Symbolic Execution

B. Method Sequence

Software testing for modern programs, in particular object-oriented languages like C# or Java, often requires a sequence of method calls (in short, a method sequence) to obtain desired object states [8], [9], [10]. To achieve high coverage, receiver objects of methods under testing or arguments must turn into desired states to cover branches. Thus, we need to call some methods to create and transform objects before the method under testing.

For example, we consider a method `AddEdge`, which adds an edge to a graph in Source code 1. The goal is to achieve full coverage of `AddEdge`. 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 Source code 2. In this manner, we often need method calls to test methods especially for object-oriented programs.

Source code 1. Graph Class

```
class Graph {
  private ArrayList<Edge> edges;
  private ArrayList<Node> nodes;

  public void AddNode(Node n) {
      if (n == null) throw new Exception();
      nodes.Add(n);
  }
  public void AddEdge(Node source, Node target
      ){
      if (nodes.Contains(source) &&
          nodes.Contains(target)) {
          edges.Add(source, target);
      } else {
          throw new Exception();
      }
  }
  ...
}
```

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

```
Graph graph = new Graph();
Node s1 = new Node();
Node s2 = new Node();
graph.AddNode(s1);
graph.AddNode(s2);
graph.AddEdge(s1, s2);
// some assertion
```

III. Seeker

Seeker is one of 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, and thus, we illustrate the algorithm in this section.

Seeker takes a target method to generate test cases as 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. We 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 variables that must change in order to cover the branch by analyzing execution traces. The variable is referred to as the *target field*.
4) It analyzes the relation (e.g. inheritance, comprehension) of the class that includes the target field.
5) From the relation extracted in 4), it finds methods that can change the value of the target field.
6) If the branch remains uncovered, Seeker adds candidates for 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 of DSE, execution traces are collected to analyze in each static analysis step. Algorithm 1 shows the pseudocode of Dynamic Analysis.

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 $dominantBranch \in tbList$ do
16:         $inputSeq$ = DynamicAnalysis($dominantBranch, inputSeq$)
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
```

B. Static Analysis

In the static analysis phase, Seeker uses uncovered branches and executed method sequences in the previous dynamic analysis phase as inputs. Then, the program detects the variable that must change in value to satisfy the constraints of uncovered branches (target field). Therefore, it finds the dependency of the class including the target field. From the identified dependency, it 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 for addition to existing method sequences. Algorithm 2 shows the pseudocode of static analysis. Next we describe important components in static analysis.

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
```

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

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 seems trivial, but there is difficulty in many cases. For example, it is simple 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: Next, Seeker builds field hierarchy that indicates the dependency between classes related to the target field from execution traces. We can trace the relation to the target field from field hierarchy.

Source code 3. An Overview of Stack Class

```
class Stack {
    private ArrayList<int> list;
    public int Count() {
        list.Count();
    }
    ...
}
class ArrayList {
    private int _count;
    public int Count() {
        return _count;
    }
    ...
}
```

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

3) Method-Call Graph: A *method-call graph* is a graph that shows the relation of methods that may modify the value of the target field and classes that include these methods. It creates a graph using field hierarchy and code analysis. The terminal nodes in the method-call graph are candidate methods to append existing method sequences. These methods can mutate the value of the target field directly or indirectly. Hence, new method sequences that are candidate methods for appending may convert a target field to a desired state and cover target branches.

IV. APPROACH

We have shown the difficulty in automated test case generation considering object states and the existing approach to overcome this problem by generating method sequences. However, some problems remain. Seeker cannot cover branches that require changing multiple objects to desired states. In fact, branches that relate multiple variables tend to be complicated and thus require sufficient testing. Hence, testing of branches related to multiple variables is a considerable problem. Therefore, we focused on this problem and propose an approach to solve it. We have implemented our approach in C# by extending a previous study of Seeker, because the proposed approach is based on Seeker.

The algorithm of the previous study identifies only one variable as the target field. Here we focus on all variables that influence target branches. We have improved the algorithm to identify all variables related to target branches. However, we can easily imagine that combinatorial explosion will occur among the combinations of all method calls to achieve conversion of all variables to the desired states. For this reason, we introduce a technique to suppress the explosion. We show the pseudocode of the proposed static analysis for multiple target fields in Algorithm 3. Next, we illustrate the details of the proposed algorithm.

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:         calculatePriority($targetseq$)
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
```

A. Static Analysis

The proposed static analysis for multiple variables is based on that of a previous study. The algorithm of the previous study identifies only one variable as target field. In this study, we focus on all variables that influence target branches. Thus, the algorithm identifies these variables as the target field. Field hierarchy and a method-call graph are built for the variables. From this, our algorithm can generate proper method sequences that cover target branches with multiple target fields.

B. Reduction of Candidates

When we consider the combinations of method calls for all target fields, we can easily imagine that the number

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

of combinations of method calls needed to transform all variables to the desired states will lead to combinatorial explosion. Furthermore, permutations of method calls are sometimes important.

For example, two target fields exist and each variable has 10 methods that may modify the value. Then, the program generates 20 method sequences that have a length of one. If the target branch is not covered yet, the program generates 400 method sequences in the next step. Thus, the number of candidate of method sequences increase exponentially. For this reason, we give method sequences priority in order to suppress the explosion. We introduce a fitness function to evaluate branch distance between constraints and method sequences. We give an evaluation value to each method sequence as a priority. In this paper, we define the fitness function only for a 32-bit integer in Table I. The definition is the same as that used by Xie et al [11].

V. EXPERIMENTS AND RESULTS

We applied our tool to four open source projects and evaluated the effectiveness of our approach. We also compared our approach to Seeker. We assessed execution time and the branch coverage of test cases that were generated.

We used four real-world open source projects in our experiments. Table II lists their features including the number of classes, methods, branches and lines. Dsa [12] is a library that provides many data structures and algorithms for the .NET framework. Quickgraph [13] is C# graph library. xUnit [14] and NUnit [15] are widely known unit testing frameworks for .NET languages. In our experiments, we applied its core component, `util` namespace, for NUnit.

We conducted the experiment on a machine that was running 32-bit Windows Vista with a 2.53-GHz Intel Core 2 Duo processor with 4 GB RAM. Our settings were as follows and were the same as used in a previous study.

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

A. Experimental Results and Evaluation

Table III lists our results. First, the proposed approach showed a 1–5% improvememt over the previous study. However, the execution time was approximately doubled. Thus, it requires more time although it achieves a few improvements for some projects such as Quickgraph.

B. Discussion

The proposed approach achieved higher branch coverage than that of the previous study for all projects, but the improvement was only 5% at most. This indicates that branches involving multiple variables, which we focused on, may occur infrequently in many projects. Although the improvement offered by the proposed approach is small, the method of the previous study cannot cover these branches with requiring extra time. Therefore, the proposed approach represents a trade-off between execution time and branch coverage. In practice, especially for the generation of test cases for mission-critical systems, the proposed approach will be useful.

For some projects such as Quickgraph, the proposed approach achieves small improvements by taking more time. Quickgraph is a library that provides data structures and algorithms for graphs, such that many methods involve various objects such as `Node` or `Edge`. If target projects have complex dependency, many operations (or long method sequences) are often required to cover branches. Thus, such projects tend to require extra time and do not allow high branch coverage.

The reasons for the small improvements achieved by the proposed approach are as follows. First, it is thought that branches involving multiple variables occur infrequently. Second, we may prune desirable candidates of method sequences in the candidate reduction step. This can occur in projects that have complex dependency, like Quickgraph, because such projects cause combinatorial explosion of method sequences. If we set a proper number of reserved method sequences, we may avoid this problem. However, execution time is wasted by increasing of the number of reserved method sequences, and thus we must determine the setting considering specific requirements for time and quality of testing. Finally, the fitness function for method sequences may be inappropriate. In our study, we calculated evaluation values only for integers. Otherwise, the fitness function returns maximum value of 32-bit integer (214748367). Therefore, we need to further review the fitness function.

VI. CONCLUSION AND FUTURE WORK

Testing for object-oriented programs requires method sequences, and many previous approaches have been proposed. Here, we focused on generating method sequences that mutate multiple variables and proposed an approach to solve the problem. We also implemented our approach based on Seeker and evaluated our approach compared with that of a previous study by applying them to four real-world projects. Our approach achieved higher branch coverage than that of the previous studies but required additional execution time. Moreover, our results suggest that the effectiveness of the proposed approach varies according to the specific characteristics of different projects.

In our future work, we will improve the fitness function and retrieval strategy for method sequences. We will adopt other fitness functions for evaluating variables other than integers. In addition, we will analyze the dispositions of covered or uncovered branches and propose a strategy for them.

ACKNOWLEDGMENT

This work was supported by JSPS KAKENHI Grant Number 25330074.

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] 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

[3] 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

[4] 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

[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] 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

[8] 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

[9] 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

[10] 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.

[11] 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

[12] D. Structures and A. (DSA), http://dsa.codeplex.com/.

[13] G. D. S. QuickGraph and A. for .NET, http://quickgraph.codeplex.com/.

[14] xUnit.net Unit testing framework for C# and .NET, http://xunit.codeplex.com/.

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