

# Method-Sequence Exploration for Automated Unit Testing of Object-Oriented Programs

Tao Xie<sup>1</sup>, Nikolai Tillmann<sup>2</sup>, Jonathan de Halleux<sup>2</sup>, Wolfram Schulte<sup>2</sup>

<sup>1</sup>Department of Computer Science, North Carolina State University, Raleigh, NC 27695

<sup>2</sup>Microsoft Research, One Microsoft Way, Redmond, WA 98052

xie@csc.ncsu.edu, {nikolait,jhalleux,schulte}@microsoft.com

## ABSTRACT

A unit test for an object-oriented program involves a sequence of method calls, which create, mutate, or observe objects. Some method arguments may have primitive types. Recent advances in symbolic execution enable the effective generation of relevant primitive values, given a fixed sequence of method calls. However, there exists a challenge in test generation: determining a sequence of method calls that may bring an object into a desirable state if the right primitive values are chosen. A desirable state may be a state that causes a particular branch in a particular method to be exercised. Thus, a unit-test generation technique that aims at maximal code coverage (i.e., covering all reachable branches) must involve the selection of relevant method-call sequences. (In that sense, maximal code coverage includes the detection of possible assertion violations.) Existing major techniques for addressing this challenge include bounded-exhaustive testing, random testing, and evolutionary testing; sequence generation in these techniques are often not effective, not sufficiently guided by a test objective such as covering a particular branch or path. In this position paper, we present our recent techniques for method-sequence exploration and our ongoing and future work to address remaining challenges.

## 1. INTRODUCTION

Generating object-oriented unit tests involves generating relevant method sequences that create, mutate, and observe objects. Some method arguments may have primitive types. Given a method-sequence skeleton, i.e. a sequence of method calls that do not fix arguments of primitive types, recent advances in symbolic execution such as dynamic symbolic execution [1, 4] enable effective generation of relevant primitive method-argument values in the method-sequence skeleton. Dynamic symbolic execution concretely explores a path using (initially arbitrary) concrete values for primitive arguments in the method-sequence skeleton and collects symbolic constraints at all branching points of the explored path during concrete execution. Negating a part of the collected constraints yields a constraint system whose solution (if any) serves as new test inputs that will exercise a different path. During each execution, test inputs that lead to an unexplored path are saved and used to generate test inputs that achieve maximal code coverage such as branch coverage or causing specified assertions to be violated if possible. However, symbolic execution techniques do not provide effective support for generating relevant method-sequence skeletons but assume that such skeletons are provided as inputs to symbolic execution.

Existing major techniques for exploring method sequences include bounded-exhaustive testing [6–8], random testing [3], and evolutionary testing [5]. Bounded-exhaustive testing techniques generate exhaustive method sequences up to a small bound (with some pruning based on state equivalence [6,7] or subsumption [8]).

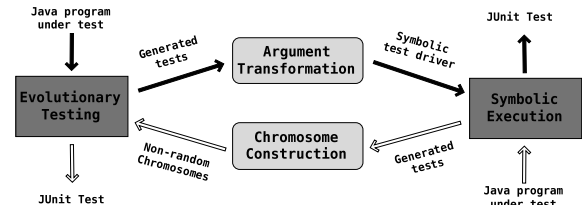


Figure 1: Overview of Evacon

However, sometimes covering some branches or violating some assertions requires long method sequences whose length is beyond the low bound that can be handled by these techniques. Random testing techniques generate random sequences (with some pruning based on execution feedback [3] such as state equivalence [6, 7]). Although random sequences can be long, the random sequences have a low chance of being relevant when relevant method sequences are only a small portion of the method-sequence exploration space. Evolutionary testing techniques [5] represent initial randomly generated method sequences as a population of individuals and evolve this population by mutating its individuals until a desirable set of method sequences is found. Although the fitness of the existing generated sequences is used to indirectly guide the generation of new sequences, the generation is basically guided random search and the generation of primitive method arguments is random by nature; these techniques cannot provide effective support for generating relevant primitive method arguments.

In this position paper, we present our two recent techniques in generating method sequences for object-oriented programs. The first technique called Evacon [2] integrates evolutionary testing and symbolic execution to address the respective weaknesses of these two techniques. The second technique currently incorporated in Pex [4] generates method sequences with a demand-driven mechanism and a heuristic-guided mechanism.

## 2. SEQUENCE EXPLORATION IN EVACON

Evacon [2] integrates evolutionary testing and symbolic execution to address the respective weaknesses of these two techniques in generating relevant method sequences. Figure 1 shows the overview of Evacon, including four components: evolutionary testing, symbolic execution, argument transformation (for bridging from evolutionary testing to symbolic execution), and chromosome construction (for bridging from symbolic execution to evolutionary testing).

The argument-transformation component transforms primitive method arguments of method sequences (produced by evolutionary testing) into symbolic arguments. This transformation allows symbolic execution techniques to do concrete and symbolic execution on the primitive arguments. For each method invocation in sequences requiring a primitive method argument, Evacon replaces instances of concrete method arguments with equivalent symbolic arguments

used to drive symbolic execution. After symbolic execution, Evacon derives the final test suite by aggregating the test inputs generated by symbolic execution and method sequences generated by evolutionary testing.

The chromosome-construction component constructs chromosomes out of method sequences generated using symbolic execution. By using chromosome construction, method sequences from symbolic execution are made available to evolutionary testing through chromosome encoding. Evolutionary testing then tries to find suitable combinations of method sequences, starting from the method sequences generated by symbolic execution.

### 3. SEQUENCE EXPLORATION IN PEX

When Pex [4] needs to create an object, and brings it into relevant states to achieve test objectives, Pex can use a combination of static and dynamic analysis to determine relevant constructors and method sequences. The static analysis determines a constructor and a set of methods that potentially set all fields of the objects to values that are given to the parameters of the constructor and the methods. The dynamic analysis monitors which methods can actually set a field (i.e., in which methods a field-write instruction is reachable). Pex then uses a demand-driven mechanism for combining effective constructors and method sequences to achieve a test objective such as covering a particular branch or path.

Pex monitors all field accesses when it runs a test and the program under test. Pex then determines desirable constraints to be satisfied by new test inputs – including objects and their field values – such that the test and the program under test can behave in other desirable ways such as covering a branch that is not previously covered. Pex next tries to create objects (of certain classes of interest) such that their field values satisfy the desirable constraints. If a class of interest is visible and has a visible constructor, Pex can create an instance of the class by invoking and exploring the constructor. Pex does not explore all code paths of the constructor but suppresses paths that throw an exception in the constructor. If another object is required by the constructor, Pex first tries to create that object, whose constructor might need another object, and so on. When some fields of the class are visible, Pex can set the values of the fields automatically based on the desirable constraints. Furthermore, Pex performs a static analysis to determine a set of visible methods with simple control flow that may set fields that are not visible (e.g., property setter methods). In some cases, these statically determined method sequences are sufficient to achieve the test objective.

However, in some other cases, calling constructors, setting visible object fields directly, and using statically determined methods are not sufficient to create objects and bring them into relevant states to achieve test objectives such as achieving maximal code coverage. For those cases, Pex uses a heuristic-guided mechanism for including and exploring other methods (besides constructors) in method sequences. In particular, Pex gathers the following information dynamically when it tries out new sequences: the fields that a method reads, the fields that a method writes, the amount of code in a method that has not been covered yet, and the target of (direct, indirect, and virtual) calls. Then Pex chooses new sequences with some exploration heuristics, such as placing methods that read a field after methods that write to a field, and targeting those public methods which (indirectly) call methods with large amounts of not-yet-covered code.

### 4. ONGOING AND FUTURE WORK

Although Evacon and Pex guide sequence exploration with test objectives to some extent, the guidance can be further enhanced to improve the effectiveness of generating relevant method sequences.

We plan to address the remaining challenges in sequence exploration with the following ongoing and future work.

The current Evacon technique integrates evolutionary testing and symbolic execution in a loose way: evolution inside the evolutionary testing process being integrated still relies on randomness to mutate some method arguments in existing method sequences. Instead of random testing, dynamic symbolic execution naturally fits in the role of mutating existing method arguments to evolve existing tests to new tests that are more fit towards the test objectives. Some open research issues are to be addressed: which method arguments in which existing method sequences need to be mutated symbolically (given that the population of existing method sequences could be large), how the genetic re-combination of method sequences (in evolutionary testing) can be guided with information gathered via dynamic symbolic execution, and how the fitness function (in evolutionary testing) can be enhanced with information gathered via dynamic symbolic execution.

The current sequence-exploration technique in Pex can be improved by giving more guidance to the demand-driven mechanism and the heuristic-guided mechanism towards test objectives. In particular, when including and exploring methods in method sequences, the exploration strategy needs to consider the test objectives, e.g., which branches to cover. Some open research issues are to be addressed: which existing sequences to expand (given that the existing method sequences could be many), which methods to add to the existing sequences, which positions in the existing sequences to add the methods, which paths in the added methods and the existing method sequences to explore, and whether and how a fitness function can be defined with information gathered via dynamic symbolic execution to guide the search process.

In addition, we plan to explore the integration and comparison of various search techniques currently in Evacon and Pex including the preceding ongoing and future improvements. Furthermore, we plan to integrate other various existing techniques for sequence exploration. For example, to achieve maximal code coverage, one option is to first apply bounded-exhaustive testing [6–8] and random testing [3] to cover the first set of (relatively-easy-to-cover) branches, and then apply the preceding guided search techniques to cover those remaining (relatively-difficult-to-cover) branches.

### 5. REFERENCES

- [1] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. PLDI*, pages 213–223, 2005.
- [2] K. Inkumsah and T. Xie. Evacon: A framework for integrating evolutionary and concolic testing for object-oriented programs. In *Proc. ASE*, pages 425–428, 2007.
- [3] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proc. ICSE*, pages 75–84, 2007.
- [4] N. Tillmann and J. de Halleux. Pex – white box test generation for .NET. In *Proc. TAP*, pages 134–153, 2008.
- [5] P. Tonella. Evolutionary testing of classes. In *Proc. ISSTA*, pages 119–128, 2004.
- [6] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with Java PathFinder. In *Proc. ISSTA*, pages 97–107, 2004.
- [7] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proc. ASE*, pages 196–205, 2004.
- [8] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proc. TACAS*, pages 365–381, 2005.