Advances on Improving Automation in Developer Testing

XUSHENG XIAO
Department of Computer Science, North Carolina State University, Raleigh, USA

SURESH THUMMALAPENTA
IBM Research, Bangalore, India

TAO XIE
Department of Computer Science, North Carolina State University, Raleigh, USA

Abstract
Developer testing, a common step in software development, involves generating desirable test inputs and checking the behavior of the program unit under test during the execution of the test inputs. Existing developer testing tools include various techniques to address challenges of generating desirable test inputs and checking the behavior of the program unit under test (referred to as test oracles). In this chapter, we present an overview of techniques implemented in these testing tools to address challenges in improving automation in developer testing. In particular, we focus on a recent state-of-the-art technique, called symbolic execution for test inputs. We briefly describe symbolic execution and discuss various challenges (along with the techniques developed to address those challenges) in generating test inputs automatically. For test inputs, the techniques presented in our chapter are summarized from two main aspects: test efficiency (e.g., with a focus on cost) and test effectiveness (e.g., with a focus on benefit). We conclude this chapter by presenting a new frontier, called cooperative developer testing, that is orthogonal to previous techniques.
and involves synergistic cooperation between humans and tools for effectively generating desirable test inputs.

1. Introduction

Software testing is the most widely used approach for improving software quality in practice. Among various types of testing, developer testing (where developers test their code as they write, as opposed to testing done by a separate quality assurance organization) has been widely recognized as a valuable means of improving software quality. Developer testing, often in the form of unit testing, helps developers to (1) gain high confidence of the program unit under test (e.g., a class) while they are writing the code and (2) reduce fault-fixing cost by detecting faults early in the software development life cycle. Recent focus on test-driven development (TDD) [1], where tests are written before the program unit, further signifies the importance of developer testing. The popularity and benefits of developer testing have been well witnessed in industry.
Typically, developer testing includes the following four major tasks: (1) generating test inputs, (2) creating expected outputs (also referred to as test oracles [2]), (3) running test inputs, and (4) verifying actual outputs. Figure 1 shows the overview of developer testing. The figure shows that test inputs are executed on a program under test to generate outputs. These generated outputs are compared with expected outputs to check whether the tests pass or fail. Among these four tasks, there exists many open source testing frameworks such as JUnit [3] (Java), NUnit [4], xUnit [5] (C#), and CppUnit [6] (C++) for automating the last two tasks of running test inputs and verifying actual outputs. Figure 2 shows an example JUnit test case `testPush`
that tests the `push` method of the `Stack` class. In this test case, Statements 17 and 18 represent test inputs, whereas Statement 19 represents the expected output. The JUnit testing framework helps to automatically execute `testPush` and verify that the return value of `s.size()` is the same as the expected value 1 (shown in Statement 19).

Although these frameworks assist in reducing effort for the third and fourth tasks, developers still need to perform the first two tasks manually. Manual developer testing is known to be labor intensive. In addition, manual testing is often insufficient in comprehensively exercising behavior of the program unit under test to expose its hidden faults. For example, because of the program complexity and limited human-brain power, developers may not be capable of coming up with certain test inputs (such as corner or special test inputs) that can expose faults in the program unit.

This chapter presents an overview of state of the art and practice in improving automation in developer testing, with primary focus on generating test inputs and test oracles. In particular, we present existing state of the art and practice in efficiently and effectively producing test inputs. Although these test inputs can help detect robustness-related defects such as Null Dereference, expected outputs for these test inputs are still missing. It is infeasible for developers to create expected outputs for this large number of generated test inputs. Specifications [7] can be used to improve the effectiveness of generating test inputs and check program behaviors when running test inputs without expected outputs. Without requiring specifications (which may be difficult for developers to write), testing tools can use code coverage criteria [8] such as statement coverage and block coverage to select a subset of generated test inputs for developers to manually verify the actual outputs.

We also provide insights to major industrial tools’ key features in improving automation in developer testing. We collect a list of major industrial developer testing tools from various sources. Specifically, we collect the first list of developer testing tools among testing tool finalists of recent annual Jolt Product Excellence and Productivity Awards\(^1\) and prestigious industrial awards in recognizing excellent industrial products. We also include some other industrial tools that first adopted important features later incorporated by some tools in the first list. In the end, this paper discusses (1) three industrial tools among Jolt Award finalists: Parasoft Jtest [9] for Java, Agitar AgitarOne [10] for Java, CodePro AnalytiX [11] for Java and (2) two other industrial tools: Microsoft Pex [12] for C# and SilverMark Test Mentor [13] for Java. Throughout the discussion of features provided by these tools, we also describe selected relevant research work from academia that may help fill the gap left by existing industrial tools. Note that the information for these discussed industrial tools is drawn from the public domain (e.g., from tool materials in respective vendor

\(^1\) [http://www.joltawards.com/](http://www.joltawards.com/)
Web sites). This chapter does not intend to compare these industrial tools side by side or provide ranking among these tools, but highlight valuable features provided by these industrial tools from two main aspects: test efficiency (e.g., with a focus on cost) and test effectiveness (e.g., with a focus on benefit).

Additionally, we present a new frontier in developer testing, called Cooperative Developer Testing, where a human and a computer collaboratively work together in effectively generating test inputs, thereby addressing limitations of existing tools and techniques. Finally, we conclude this chapter by presenting future directions of developer testing: correctness confidence, specifications, (dis)integration testing, and human factors.

2. Test Efficiency

Existing tool support for improving test efficiency includes creating and running test inputs more efficiently. One of the key techniques in improving test efficiency is to capture and replay. Such a technique has been traditionally used in GUI or Web application testing, being supported by various industrial tools such as IBM Rational Robot [14]. In the context of developer testing, the capture phase of the capture-and-replay technique monitors the interactions of the unit under test, e.g., a class, and its environment, e.g., the rest of the system where the class is, during the execution of the system. Such system execution can be induced by manually or automatically running system tests. Based on the monitored interactions, the capture phase automatically creates unit tests for the unit under test. Each unit test includes (1) test inputs as captured method invocations to the unit (in addition to some other necessary method invocations to other units for producing method arguments of the unit) and (2) test oracles as the captured return values of the captured method invocations to the unit). The replay phase of the technique simply reruns the created unit tests, which check the unit behavior with their test oracles.

In contrast to automatically running the system tests, automatically running the created unit tests is faster, since the unit tests focus on only the interactions with the unit under test. In contrast to manually running the system tests, automatically running the created unit tests is much faster because no manual effort is required any more besides focusing on the interactions with the unit. In contrast to manually writing these unit tests for the unit, the technique allows automatic creation of these unit tests.

Note that this technique exercises no new unit behavior beyond the one exercised by the system test execution. Therefore, no new code coverage can be achieved by the created unit tests beyond the system tests. But, it may be possible that the created unit tests can expose new faults not exposed by the system tests because the test oracles for the unit tests can be stronger than the ones for the system tests by checking inside
the system black box [15]. The technique would be primarily useful in regression testing (i.e., checking the behavior of a new version to be the same as the one of the old version). When applying the technique on an initially faulty unit, the capture phase would capture the faulty behavior of the faulty unit and the replay phase would make sure that this faulty behavior would remain in future versions!


One challenge in this technique is to deal with nonprimitive-object-type argument values and return values on the unit interface when creating unit tests from unit interactions. Some tools may just handle only primitive values in the unit interactions. Some tools may cache or serialize an object’s value in the capture phase and deserialize it in the replay phase. However, such a mechanism would produce obsolete or broken unit tests for later versions of the unit where the classes related to these objects are refactored, causing their object fields being changed. One better mechanism is to capture and replay method sequences (not necessarily invoked on the unit) that produce actual object values [16]. Another better mechanism is to use a mock object [17] in place of an argument or return object (as supported by Microsoft Pex [18,19] and related to automatic stub generation provided by Parasoft Jtest [9]). Then, tools can capture and replay arguments and return values of methods of the mock object. Various researchers [17,20,21] have investigated advanced mechanisms of the capture-and-replay technique. In terms of improving efficiency in manually creating unit tests, researchers [22] have also developed IDE support for helping developers write unit test inputs and oracles faster.

Recently, researchers [23,24] investigated new techniques to capture and replay, where during the replay phase, generated tests exercise unit behavior beyond the captured behavior. We next describe these two new techniques, OCAT [23] and DyGen [24], which are yet to be adopted by industrial tools.

### 2.1 OCAT Technique

OCAT captures object states dynamically during program executions and reuses captured object states to assist a random approach. In summary, OCAT includes three major phases: (1) object capturing (CAP): capture object instances from executions; (2) object generation (GEN): generate new object instances via applying method-sequence generation techniques on the captured object instances; and (3) object mutation (MTT): mutate the captured object instances to obtain desired object instances to cover not-yet-covered branches. We next explain each phase in detail.
2.1.1 Object Capturing

In the object-capturing phase, OCAT captures object instances from normal program executions (e.g., ones from system testing or real use). Suppose that we want to test the Eclipse program, it is generally hard to automatically generate desirable object instances; however, if we run and use Eclipse, during execution, there are many object instances being created in the memory heap. Since these object instances reflect real usage, capturing and exploiting them in automated testing could provide potential for being desirable in achieving new branch coverage. Some examples of types of objects are (1) classes under test (e.g., receiver classes), (2) arguments of a method under test, and (3) objects needed to directly or indirectly construct the first two types of objects. OCAT captures objects as serialized instances.

2.1.2 Object Generation

In the object-generation phase, OCAT generates new object instances using a method-sequence generation technique [25] and captured object instances. Particularly, OCAT leverages a method-sequence generation technique by using the captured object instances in two ways. First, the captured object instances can be directly used by de-serializing them. Second, the captured object instances contribute to the creation of other necessary object instances for testing. Let \( C \) be a set of captured object instances by OCAT. Consider two target methods \( m_i \) of class \( i \) and \( m_j \) of class \( j \). Consider two sets of desirable object instances \( D_{m_i} \) and \( D_{m_j} \) that cover code in methods \( m_i \) and \( m_j \), respectively. Let \( R_{m_j} \) be a set of object instances returned by invoking \( m_j \) on \( D_{m_j} \). If \( D_{m_i} \subseteq C \), code in method \( m_i \) can be directly covered by using the captured object instances. If \( D_{m_i} \supset C \), but \( D_{m_i} \subseteq R_{m_j} \) and \( D_{m_j} \subseteq C \), then code in \( m_i \) can be indirectly covered by feeding the object instances returned by invoking \( m_j \) on the captured object instances.

Using captured object instances as initial inputs reduces the huge search space of desirable object instances in the method-sequence generation process, since the captured object instances are likely close to desirable object instances. Therefore, captured object instances make the method-sequence generation approach effective to produce desirable object instances and construct and execute method sequences with the captured object instances to achieve high code coverage.

2.1.3 Object Mutation

Generating object instances by invoking method sequences with captured object instances may not cover all branches. To address this issue, OCAT mutates object instances to satisfy the conditions of not-covered branches. OCAT analyzes the conditions related to not-yet-covered branches and mutates the captured object
instances to satisfy the conditions. This mutation phase helps OCAT to replay the behavior beyond the captured behavior.

The mutation phase includes the following five major steps: (1) OCAT identifies not-yet-covered branches by analyzing source code and branch coverage information. (2) OCAT conducts static analysis to collect constraints starting from the not-yet-covered branches in a backward traversal manner of code analysis. (3) OCAT solves the collected constraints by using a Satisfiability Modulo Theories (SMT) solver. (4) OCAT uses the solved solution from the SMT solver as a concrete input value of the method that has the target not-covered branch. (5) If the solution is related to a member field of an object input, OCAT loads and mutates a captured object instance. When modifying member-field values of an object instance, OCAT does not change a private field value as a default setting because modifying a private field value might break class invariants. To avoid invalid object instances caused by modifying private field values, OCAT provides an option of allowing developers to provide a predicate method (also called repOk() [26]) that checks class invariants of a class. Programming best practices suggest that a programmer provides such a method when writing a class.

2.2 DyGen Technique

In contrast to OCAT, DyGen [24] mines dynamic traces recorded during program executions and generates regression test inputs from mined traces. The key idea behind DyGen is that unit tests captured from dynamic traces tend to exercise only happy paths (such as paths that do not include error-handling code in the code under test), instead of all feasible paths, achieving low coverage of the code under analysis. To address this issue, DyGen transforms concrete values into symbolic values and uses dynamic symbolic execution (described in Section 3.1.1) to generate new tests that achieve high coverage of the code under analysis. In particular, DyGen includes three major phases: capture, minimize, and explore. Figure 3 shows the overview of the DyGen technique. We next explain each phase in detail.

![Fig. 3. Overview of the DyGen Technique.](image-url)
2.2.1 Capture Phase

In the capture phase, DyGen records dynamic traces from program executions. The capture phase uses a profiler that records method calls invoked by the program during execution. The capture phase records both the method calls invoked and the concrete values passed as arguments to those method calls. Figure 4a shows an example dynamic trace recorded by the capture phase. Statement 2 shows the concrete value “<%@Page..\u000a” passed as an argument for the Match method. DyGen transforms these traces into Parameterized Unit Test (PUT) [27] and seed tests. To generate PUTs, DyGen identifies all constant values and promotes those constant values as parameters. Furthermore, DyGen identifies return values of method calls in the PUT and promotes those return values as out parameters for the PUT. In C#, these out parameters represent the return values of a method. DyGen next generates seed tests that include all concrete values from the dynamic traces. Figures 4b and 4c show the PUT and the seed test, respectively, generated from the dynamic trace shown in Fig. 4a.

```
01: TagRegex tagex = new TagRegex();
02: Match mc = ((Regex)tagex).Match("<
   %@Page..
",108);
03: Capture cap = (Capture) mc;
04: int indexval = cap.Index;
```

a. An example trace recorded by the capture phase.

```
01: public static void F1(string val1, int val2, out int out1)
02: {
03:     TagRegex tagex = new TagRegex();
04:     Match mc = ((Regex)tagex).Match(val1, val2);
05:     Capture cap = (Capture) mc;
06:     out1 = cap.Index;
07: }
```

b. A PUT generated from the trace.

```
01: public static void T1() {
02:     int index;
03:     F1("<%@ Page..
", 108, out index);
04: }
```

c. A seed test generated from the trace.

Fig. 4. Illustrative examples for the DyGen Technique.

---

2 Parameterized unit tests, unlike conventional unit tests, accept parameters.
2.2.2 \textit{Minimize Phase}

In the minimize phase, DyGen filters out duplicate PUTs and seed tests. The reason for duplicates is that the same sequence can get invoked multiple times, resulting in duplicate dynamic traces. DyGen considers a PUT, say $P_1$, as a duplicate of another PUT, say $P_2$, if both $P_1$ and $P_2$ have the same sequence of instructions. On the other hand, DyGen considers a seed test, say $S_1$, as a duplicate of another seed test $S_2$, if both $S_1$ and $S_2$ follow the same path in the code under analysis.

2.2.3 \textit{Explore Phase}

In the explore phase, DyGen uses Pex to generate regression tests from PUTs. Although seed tests generated in the capture phase can be considered as regression tests, most seed tests tend to exercise common happy paths such as paths that do not include error-handling code in the code under test. Without any seed tests, Pex starts exploration with an empty execution tree, and all nodes are discovered incrementally. Instead, with seed tests, Pex executes those seed tests and internally builds an execution tree with nodes for all conditional control-flow statements executed along the paths exercised by the seed tests. Pex starts exploration from this prepopulated tree, rather than starting from an empty execution tree. Therefore, using seed tests significantly reduces the amount of time required in generating a variety of tests with potentially deep execution paths from PUTs. This explore phase helps DyGen to exercise behavior beyond the behavior captured by traces.

3. \textit{Test Effectiveness}

In contrast to improving test efficiency (concerning more about reducing cost), improving test effectiveness focuses on improving the quality of the test inputs and test oracles. We next discuss issues and techniques in generating test inputs and test oracles.

3.1 Test Inputs

In testing object-oriented programs, improving test-input generation includes two subtasks: (1) generating desirable primitive-method-argument values and (2) generating desirable method-argument objects and receiver objects. Here, \textit{desirable} test inputs are test inputs for achieving some test objectives not previously achieved such as causing program crashes, violating specified properties, and achieving code coverage (which are all related to test oracles discussed in \textit{Section 3.2}).
3.1.1 Generating Desirable Primitive Values

In the first subtask, industrial tools often use four main techniques:

- Use default values for a primitive type.
- Use constant values (of the same primitive type) that appear in the code under test.
- Use random values.
- Use values derived with symbolic execution [28].

Symbolic execution has been popularly used by tools such as Parasoft Jtest, Agitar Agitator, and Microsoft Pex [19], and it generally performs better than the other three main techniques in terms of achieving high code coverage.

3.1.1.1 Symbolic Execution. Symbolic execution [28], a case of abstract interpretation [29], analyzes programs by tracking symbolic values instead of actual values. Symbolic execution associates symbolic values with program variables (such as program inputs and local variables) and statically simulates program executions to track these symbolic values. Along the simulated execution, symbolic execution collects constraints on symbolic values obtained from predicates in branch statements to form a Path Condition, which is a symbolic expression that can be used to reason about all program inputs that take the same path through the program.

Dynamic symbolic execution (DSE) [30,31] extends the conventional static symbolic execution with additional information collected at runtime, making the analysis more precise [31,32]. DSE first executes a program with simple inputs while performing a symbolic execution to collect path conditions along the execution. By negating some conditions of a collected path condition, DSE obtains a new path condition that leads to a path taking different branches. DSE then uses a constraint solver, such as Z3 [33], to compute new inputs that satisfy the new path condition and executes the program using the obtained inputs. In theory, DSE can exercise all feasible paths of the program by repeating the input or path variations.

Consider the method CoverMe shown in Fig. 5. When DSE is applied to generate test inputs for the method CoverMe, DSE first executes the method CoverMe using the simple initial values 0 for both arguments x and y. In parallel to the actual program execution, DSE associates the program inputs x and y with symbolic values and collects constraints on these symbolic values along the execution to form a path condition. After the first execution, the collected path condition is \( x \neq 12345 \), and a search strategy is used to decide on a constraint in the path condition to negate. In this example, we assume that DSE uses the depth-first search (DFS) strategy that always selects the last constraint of the path condition to negate. Based on the DFS search strategy, DSE negates the constraint \( x \neq 12345 \) to obtain a
Fig. 5. Example method to illustrate symbolic execution.

Fig. 6. Example method to illustrate the Fitnex approach.

new path condition \( x == 12345 \), which takes the true branch at Line 2. This new path condition is then fed into a constraint solver to obtain new test inputs \( x = 12345, y = 0 \) for the next execution. Similarly, after the second execution, DSE collects another path condition \( x == 12345 && y \neq 67890 \) and obtains a new path condition \( x == 12345 && y == 67890 \) by negating the last constraint \( y \neq 67890 \). Solving this new path condition, DSE produces a new test input \( x = 12345, y = 67890 \) to cover the true branch at Line 3. Since these three test inputs already achieve full path coverage of the method \( \text{CoverMe} \), DSE stops the exploration.

Unlike a random approach \([25,34,35]\), the structural test-input-generation approach used by DSE is able to achieve high structural coverage using fewer test inputs.

As shown in this example, symbolic execution generates test inputs by exploring the program under test path by path. In practice, the tremendous number of execution paths limits the size of programs that can be symbolically executed \([36]\), preventing DSE from achieving high structural coverage within a reasonable amount of time. Consider another example method \( \text{loopMethod} \) shown in Fig. 6. By inspecting the method \( \text{loopMethod} \), we can see that to cover the true branch of Line 6 needs exactly 20 times of executing the true branch of Line 4 inside the loop. However, applying DSE to hunt for such a case faces two significant challenges. First, the number of loop
iterations in Lines 3–5 depends on the length of the array `array`, which can range from 0 to $2^{31} - 1$ for a 32-bit computer. A breadth-first or depth-first search strategy would not be able to explore and cover the true branch of Line 6 within a reasonable amount of time. Second, even when we put a bound such as 20 for the loop iterations, the number of paths for the loop iterations is still $2^{20}$, which is too large to explore in practice. This example method illustrates a general exploration problem for DSE: to cover one or more branches (e.g., `v == 121`) whose conditions are computed based on control-flow decisions (e.g., `if (array[i] == 21) v++;`).

To tackle this exploration problem of DSE, Xie et al. propose an approach called Fitnex [37] for guided path exploration in DSE to achieve structural coverage quickly. Fitnex assigns to already explored paths fitness values computed by program-derived fitness functions. (Fitness functions have been traditionally used in search-based test generation [38].) A fitness function measures how close an explored path is in achieving target test coverage. A fitness gain is also measured for each explored branch: a higher fitness gain is given to a branch if negating its constraint in the past helped achieve better fitness values. Then during path exploration, the Fitnex strategy would prefer to negate a constraint that has a higher fitness gain in a previously explored path with a better fitness value. The core Fitnex strategy is effective for only certain exploration problems that are amenable to fitness functions. To address the issue, Xie et al. integrate the Fitnex strategy with other search strategies (e.g., depth-first search, random search), which perform well for other types of exploration problems. The integration of the Fitnex and other strategies achieves the effect of getting the best of both in practice.

Although research approaches such as Fitnex [37] address some challenges faced by test-input generation using symbolic execution, there are still more challenges that require further research, such as complex logics in code involving complicated constraints beyond the capability of constraint solvers underlying symbolic execution and involving a large/infinite number of paths (especially when loops are involved).

### 3.1.2 Generating Desirable Objects

We next present more details on the second task of generating desirable method-argument objects and receiver objects. Achieving high coverage of object-oriented code requires desired object states for the receiver or arguments of a method under test (MUT). These desired object states help cover true or false branches of the conditional statements (such as `if` statements) in the MUT. For example, consider the two classes from the C# QuickGraph [39] library shown in Fig. 7. The figure shows two classes under test `BidirectionalGraph` and `BFSAlgorithm`. `BidirectionalGraph` represents a graph structure including vertices and edges,
**Fig. 7.** Two classes from C# QuickGraph library [39].

which are added using `AddVertex` and `AddEdge`, respectively. `BFSAlgorithm` performs breadth first search on the graph structure. We added an additional field `isComputed` for illustrative purposes.
A desired object state for covering the true branch of Statement 27 (Branch B4) in Fig. 7 is that the graph object should include at least one edge. For example, the following method sequence (S1) produces the preceding desired object state for the graph object, thereby covering B4.

```java
00: BidirectionalGraph ag = new BidirectionalGraph();
01: Vertex v1 = new Vertex(0);
02: ag.AddVertex(v1);
03: ag.AddEdge(v1, v1);
```

In this sequence, AddVertex should precede AddEdge to satisfy the requirement that the vertices passed as arguments should already exist in the graph object (Statements 8 and 11). To achieve this subtask of generating method sequences to produce desirable objects, industrial tools use three main techniques.

- Generate default sequences (e.g., using the null reference or invoking only a constructor).
- Generate random method sequences.
- Generate sequences based on heuristics.

Most industrial tools likely use one or more of these three techniques. For example, Microsoft Pex [12] uses heuristics based on static information of constructors and other methods (of classes under test) that set values to member fields to generate desirable method sequences. Note that, these three techniques may not be effective in practice.

Automatic generation of target sequences that produce a desired object state is challenging due to three major factors. First, target sequences often include methods from multiple classes, resulting in a large search space of candidate sequences. Second, target sequences require specific primitive values that help exercise desired paths in the code under analysis. Third, object-oriented programming features such as encapsulation pose additional challenges, since values cannot be directly set to member fields. Being an interesting and challenging problem, many researchers in academia proposed various techniques to address this problem. These techniques are yet to be adopted by industrial tools.

These academic techniques can be broadly classified into four categories: bounded-exhaustive [16,40], evolutionary [41–43], usage-based [23,24,44], and implementation-based [9,25,34,45] techniques. Given a small bound on the length of sequences, bounded-exhaustive techniques generate sequences exhaustively up to lengths of the given bound. On the other hand, evolutionary techniques evolve an initial set of sequences using a fitness [46] metric computed based on desired object states. Usage-based techniques generate method sequences based on how method
calls are used in practice. In contrary to usage-based techniques, implementation-based techniques rely solely on the implementation information of the method calls. This last category is a broad category and includes both bounded-exhaustive and evolutionary as well. However, we classify the techniques that do not fall into any of the previous categories as implementation-based techniques. We next present more details about one technique from each category.

3.1.2.1 Bounded-Exhaustive Technique: Symstra. We next present more details about Symstra [47] from the bounded-exhaustive category. Given a set of methods from the class under test and a bound on the length of sequences, Symstra systematically explores the object-state space of the class and prunes this exploration based on state comparisons. We next explain how Symstra explores method sequences and generates test inputs using an illustrative example shown in Fig. 8.

Figure 8 shows a binary search tree that implements a set of integers. Each tree object has a pointer to the root node. Each node object has an element and pointers to the left and right children. The BST class also implements the standard set operations: insert adds an element, if not already in the tree, to a leaf; remove deletes an element, if in the tree, replacing it with the smallest larger child if necessary; and contains checks whether the element is in the tree. The class also has a default constructor that creates an empty tree.

Given a class under test such as BST, Symstra explores all sequences, but using symbolic arguments for method calls. Such exploration helps Symstra to automatically infer desired values for arguments of these method calls. Let us consider the symbolic execution of the following sequence:

01: BST t = new BST();
02: t.insert(x1);

```java
1 class BST implements Set {
2     Node root;
3     static class Node {
4         int value;
5         Node left;
6         Node right;
7     }
8     void insert(int value) { ... }
9     void remove(int value) { ... }
10    boolean contains(int value) { ... }
11 }
```

Fig. 8. A set implemented as a binary search tree.
This sequence has four method calls whose arguments are symbolic variables $x_1$, $x_2$, $x_3$, and $x_4$. Although an execution of a sequence with concrete arguments produces one state, symbolic execution of a sequence with symbolic arguments can produce several states, thus resulting in an execution tree. Figure 9 shows a part of the execution tree for this example. Each state has a heap and a constraint that must hold for that heap to be created. The constructor first creates an empty tree. The first `insert` then adds the element $x_1$ to the tree.

The second `insert` produces the three states shown in Fig. 9: if $x_1 = x_2$, the tree does not change, and if $x_2 > x_1$ (or $x_2 < x_1$), $x_2$ is added in the right (or left) subtree. Note that the symbolic states $s_2$ and $s_4$ are syntactically different: $s_2$ has the constraint `true`, while $s_4$ has $x_1 = x_2$. However, these two symbolic states are semantically equivalent: they can be instantiated into the same set of concrete heaps by giving to $x_1$ and $x_2$ concrete values that satisfy the constraints; Because $x_2$ does not appear in the heap in $s_4$, the constraint in $s_4$ is “irrelevant.” Instead of state equivalence, it suffices to check state subsumption: we say that $s_2$ subsumes $s_4$ because the set of concrete heaps of $s_4$ is a subset of the set of concrete heaps of $s_2$. Hence, Symstra does not need to explore $s_4$ after it has already explored $s_2$. Symstra detects this case by checking that the implication of constraints $x_1 = x_2 \Rightarrow \text{true}$ holds.

The third `insert` again produces several symbolic states. Symstra applies `insert` only on $s_3$ and $s_5$ (and does not explore $s_4$). In particular, we focus on $s_6$ and $s_7$, some of the symbolic states that these executions produce. These two states are

Fig. 9. A part of the symbolic execution tree.
syntactically different, but semantically equivalent: we can exchange the variables \( x_2 \) and \( x_3 \) to obtain the same symbolic states. Symstra detects this case by checking that \( s_6 \) and \( s_7 \) are isomorphic. Symstra finally applies remove. Note again that one of the symbolic states produced, \( s_8 \), is subsumed by an existing state, \( s_3 \).

This example has illustrated how Symstra would conduct symbolic execution for one particular sequence. Symstra actually exhaustively explores the symbolic execution tree for all sequences up to a given length, pruning the exploration based on subsumption. After generating the symbolic execution tree, Symstra can generate specific test inputs with concrete arguments. Symstra generates test inputs by traversing the tree and outputting the method calls that it encounters. To generate concrete arguments for these calls, Symstra uses a constraint solver. Symstra generates the following tests for \( s_3 \) and \( s_4 \):

Test 1 (T1):
- BST \( t_1 = \text{new BST}() \);
- \( t_1.insert(-1000000) \);
- \( t_1.insert(-999999) \);

Test 2 (T2):
- BST \( t_2 = \text{new BST}() \);
- \( t_2.insert(-1000000) \);
- \( t_2.insert(-1000000) \);

3.1.2.2 Evolutionary Technique: Evacon. We next present Evacon [42] from the evolutionary category. Indeed, Evacon is not a pure evolutionary technique, but integrates evolutionary testing (used to search for desirable method sequences) and symbolic execution (used to generate desirable method arguments), thereby addressing respective weaknesses of these two techniques.

Figure 10 shows the overview of Evacon, including four components: evolutionary testing, symbolic execution, argument transformation (for bridging from evolutionary

![Fig. 10. Overview of the Evacon technique.](image-url)
public class PersonalAccount {
    private float balance;
    private int frequency;

    public void depositAmount(float money) {
        if (money > 0.0F) {
            balance = balance + money;
        }
    }

    public void transfer(float money) {
        if (money > balance) {
            printError();
            return;
        }

        if (frequency >= 5) {
            printError();
            return;
        }
        balance -= money;
        ++frequency;
    }
}

Fig. 11. A Personal bank account example.

testing to symbolic execution), and chromosome construction (for bridging from symbolic execution to evolutionary testing). We use the personal bank account example (shown in Fig. 11) as an illustrative example.

Evolutionary Testing. Evacon uses an evolutionary technique, called eToc [41], that implements genetic algorithms mimicking natural evolution. In eToc, method sequences represent actions that can be encoded as chromosomes of individuals in a population. A population represents a potential solution to a testing goal, and this solution can be optimized through genetic recombination and mutation. Furthermore, optimizing potential solutions requires the use of a formula of fitness to filter out less suitable individuals with regards to the testing goal while preserving more suitable ones. Recombining and mutating the more suitable individuals then become the basis for generating a new population, which is hoped to be at least as fit as the predecessors. Figure 12 shows an example test generated by the evolutionary testing tool.
Fig. 12. A test generated by an evolutionary testing tool.

Fig. 13. Symbolic test of the test in Fig. 12.

**Symbolic Execution.** Evacon uses a symbolic execution technique, called jCUTE [30], that combines both concrete and symbolic executions. Section 3.1.1 presents more details about symbolic execution. We next focus on how Evacon integrates evolutionary testing and symbolic execution using components of argument transformation and chromosome construction.

**Argument Transformation.** The argument transformation component is used when test generation starts with evolutionary testing followed by symbolic execution. The argument-transformation component transforms primitive method arguments of method sequences (produced by evolutionary testing) into symbolic arguments. Figure 13 is the resulting symbolic test after argument transformation is applied on the test in Fig. 12. A float value is transformed to a symbolic float input represented as cute.Cute.input.Float(), an API method provided by jCUTE.

This transformation allows jCUTE’s symbolic execution technique to do concrete and symbolic execution on the primitive arguments. After symbolic execution, we derive the final test suite by aggregating the tests generated by symbolic execution and method sequences generated by evolutionary testing. In doing so, we preserve the level of coverage achieved by the method sequences obtained from evolutionary testing while augmenting this coverage by generating additional argument values that can achieve new coverage. Figure 14 shows a test generated by jCUTE by using the sequence generated by eToc.

**Chromosome Construction.** The chromosome-construction component constructs chromosomes out of method sequences generated using symbolic


```java
1. public void testGenByEvolAugmentedByConcolic() {
2.     PersonalAccount account = new PersonalAccount();
3.     account.depositAmount(30.00F);
4.     account.transfer(2.00F);
5. }
```

Fig. 14. A test generated by integrating evolutionary testing and symbolic execution.

execution. By using chromosome construction, method sequences from symbolic execution are made available to evolutionary testing through chromosome encoding. Each method call is encoded and all the encoded method calls are joined together. Below is the encoding for the test in Fig. 14.

```
$b0,PersonalAccount,[]: $b0,PersonalAccount,depositAmount,[float]:30.00 $b0,PersonalAccount,transfer,[float]:2.00
```

Each encoding has four parts except for constructor invocations, which have three parts. The first part, which serves as a variable identifier for the receiver object, is a unique alphanumeric value prefixed by the $ symbol. The identifier is assigned by the chromosome constructor. The second part is the name of the class to which the method being invoked belongs (this part is omitted for constructor calls). The third part is the name of the method being invoked. Finally, the fourth part lists the method arguments’ data types and corresponding values.

Below is the chromosome produced for the test in Fig. 14 derived after encoding method calls and joining them together.

```
$b0=PersonalAccount():$b0.depositAmount(float): $b0.transfer(float)@30.00,2.00
```

To produce the above chromosome, the chromosome constructor maintains the association between the chromosome identifier and its associated method calls as well as the associated method argument types and method argument values in their correct order. The final outcome of chromosome construction is a list of nonrandom chromosomes to be used in evolutionary testing. The chromosome-construction component is used when test generation starts with symbolic execution followed by evolutionary testing.

Evolutionary testing tries to find suitable combinations of method sequences, starting from the method sequences and method arguments generated by symbolic execution. For example, given the preceding chromosome, evolutionary testing can help generate desirable method sequences for achieving new branch coverage, such as the test in Fig. 15.
1. public void testGenByConcolicAugmentedByEvol() {
    2.     PersonalAccount account = new PersonalAccount();
    3.     account.depositAmount(30);
    4.     account.transfer(2.00);
    5.     ...//repeated account.transfer(2.00) 4 times
    6.     acc.transfer(2.00);
}

Fig. 15. A test generated by integrating symbolic execution and evolutionary testing.

BidirectionalGraph graph = new BidirectionalGraph();
IVertex a = graph.AddVertex();
IVertex b = graph.AddVertex();
IVertex c = graph.AddVertex();
graph.addEdge(a,b);
graph.addEdge(b,c);

Fig. 16. A sequence for producing a BidirectionalGraph object with three vertices and two edges.

3.1.2.3 Usage-Based Technique: MSeqGen. We next present MSeqGen [44] from the usage-based category. MSeqGen generates target sequences from a novel perspective of how method calls are used together, referred to as usage information, in practice. The key insight of the MSeqGen technique is that the method sequences (in existing code bases) that use object types such as receiver or argument object types of the MUT can help generate target sequences. For example, an existing code base can include the sample sequence shown in Fig. 16 that produces an instance of BidirectionalGraph with three nodes and two edges. We next explain MSeqGen in detail.

MSeqGen includes two major components: code search and analysis and sequence generalization. Figure 17 shows the overview of MSeqGen. MSeqGen accepts an application under analysis and generates test inputs that exercise various paths in the application under test. In the figure, target classes, denoted by \( \{TC_1, TC_2, \ldots, TC_m\} \), represent the classes in the application under test for which sequences need to be collected. MSeqGen also accepts a set of existing code bases, denoted by \( \{CB_1, CB_2, \ldots, CB_n\} \), that already use these target classes. MSeqGen mines various sequences for target classes in given code bases and uses those sequences to assist random and DSE-based approaches in generating test inputs.

Code Searching and Analysis. Initially, MSeqGen searches for relevant method bodies in code bases by using target classes as keywords. The primary
reason for code search is that code bases are often large and analyzing complete code bases can be prohibitively expensive. To avoid analyzing complete code bases, MSeqGen uses a keyword search to identify relevant method bodies including target classes. MSeqGen considers that a method body is relevant to a target class \( TC_j \), if the method body includes the name of the \( TC_j \) target class. For example, MSeqGen uses BidirectionalGraph as a keyword and search for method bodies including that keyword. Figure 18 shows an example method body including the BidirectionalGraph keyword.

![Fig. 17. Overview of the MSeqGen technique.](image)

```java
1   public void SortVertices() {
2       BidirectionalGraph g = new BidirectionalGraph();
3       ArrayList iv = new ArrayList();
4       int i = 0; //adding vertices
5       IVertex a = g.AddVertex();
6           iv.Add(a);
7       IVertex b = g.AddVertex();
8           iv.Add(b);
9       IVertex c = g.AddVertex();
10          iv.Add(c);
11       g.AddEdge(a,b); //adding edges
12       g.AddEdge(b,c);
13       StrongComponentsAlgorithm topo = new
14           StrongComponentsAlgorithm(g);
15           topo.Compute(); ... }
```

![Fig. 18. A relevant method body for classes BidirectionalGraph, ArrayList, and StrongComponentsAlgorithm.](image)
MSeqGen next analyzes each relevant method body statically and constructs a control-flow graph (CFG). The constructed CFG includes four types of statements: method calls, object creations, typecasts, and field accesses. The rationale behind choosing these statements is that these statements result in generating objects of target classes. While constructing a CFG, MSeqGen identifies the nodes (in the constructed CFG) that produce the target classes such as BidirectionalGraph and mark them as nodes of interest. For example, the node corresponding to Statement 2 in Fig. 18 is marked as a node of interest for the target class BidirectionalGraph. MSeqGen also filters out irrelevant method bodies identified during the code searching phase if their related CFGs do not contain any nodes of interest.

MSeqGen next extracts sequences from the CFG using nodes of interest. For each node of interest related to a target class $T_{C_j}$, MSeqGen gathers a path from the node of interest to the end of the CFG. In case of loops, MSeqGen considers the nodes inside a loop as a group of nodes that is executed either once or not. Considering these nodes once can help identify the sequence inside the loop. MSeqGen also annotates these nodes to store the additional information that these nodes (and their associated method calls) exist inside loops. This additional information is used in subsequent phases while generating code, based on extracted sequences.

Often, an extracted sequence can include a few method calls that are unrelated to the target class $T_{C_j}$. MSeqGen uses data-dependency analysis to filter out such unrelated method calls from the extracted sequence. MSeqGen starts with the method call (in short as base method call) associated with a node of interest and filters out method calls that do not share the same receiver object as the base method call. The data-dependency analysis results in a sequence that creates and mutates an object of a target class $T_{C_j}$. For example, Fig. 16 shows a sequence gathered from the code example in Fig. 18. MSeqGen extracts several such sequences for different classes from the same code example. For example, if the set of target classes also includes classes ArrayList and StrongComponentsAlgorithm, MSeqGen automatically extracts one sequence for each of these classes as shown below:

```java
Sequence for ArrayList:
1   IVertex a, b, c; // requires as input
2   ArrayList iv = new ArrayList();
3      iv.Add(a);
4      iv.Add(b);
5      iv.Add(c);

Sequence for StrongComponentsAlgorithm:
9   BidirectionalGraph g; // requires as input
10  StrongComponentsAlgorithm tsObj = new
11      StrongComponentsAlgorithm(g);
12     tsObj.compute();
```
One issue with extracted sequences is that these sequences can include additional nonprimitive types. For example, the sequence for `StrongComponentsAlgorithm` requires nonprimitive type `BidirectionalGraph`. To achieve target states, `MSeqGen` needs new sequences for generating these additional nonprimitive types. In principle, call sites in code bases including sequences for a `TC_j` target class also include the sequences for generating related additional nonprimitive types. However, in practice, often these call sites do not include sequences for these additional nonprimitive types due to two factors. (1) A sequence for an additional nonprimitive type is available in another method body and is not found by our approach as it uses intraprocedural analysis for extracting sequences. (2) A sequence for an additional nonprimitive type does not exist in the current code base $CB_i$ (such as a framework or a library) and expects a reusing application to provide a necessary sequence.

`MSeqGen` addresses this issue by extracting new sequences for additional nonprimitive types by using an iterative strategy. More specifically, `MSeqGen` first extracts sequences for the initial set of target classes and collects all additional classes for which new sequences need to be extracted. `MSeqGen` next extracts sequences for these additional classes and collects more new additional classes. `MSeqGen` repeats this process either till no new additional classes are collected or reaches a fixed number of iterations accepted as a configuration parameter, denoted by $NUM\_ITERATIONS$. A high value for $NUM\_ITERATIONS$ can help collect more sequences; however, a high value can require more time for collecting those sequences.

**Method Sequence Generalization.** `MSeqGen` next generalizes sequences to address an issue that constant values in extracted sequences can be different from values required to achieve target states. This process of converting sequences into skeletons (which are sequences with symbolic values instead of concrete values for primitive types) as *sequence generalization*. For example, consider an example class definition and a method sequence shown in Fig. 19. Although the sequence includes all necessary method calls to achieve the *true* branch of Statement 5 in Fig. 19, it cannot directly achieve, since the value of the field $f_2$ is set to $-10$. To address this issue, `MSeqGen` generalizes extracted sequences. More specifically, `MSeqGen` replaces constant values of primitive types in extracted sequences with symbolic values. Figure 19C also shows the skeleton, where two symbolic variables $x_1$ and $x_2$ are taken as inputs for the sequence. When this skeleton is used along with an approach based on dynamic symbolic execution (DSE) [30,31] approach, the DSE-based approach initially generates a concrete random value for the `symvar` symbolic variable and gathers the constraint ($>25$) in the MUT through dynamic execution. The DSE-based approach next solves the constraint to generate another concrete value for `symvar` such as 30 that satisfies the gathered constraint.
A. Class Definition:
00: Class A {
01:     int f1 { set; get; }
02:     int f2 { set; get; }
03:     void CoverMe() {
04:         if (f1 != 10) return;
05:         if (f2 > 25)
06:             throw new Exception("bug");
07:     }
08: }

B. Method sequence (MCS):
00: A obj = new A();
01: obj.setF1(14);
02: obj.setF2(-10);
03: obj.CoverMe();

C. Skeleton:
00: int x1 = *, x2 = *;
01: A obj = new A();
02: obj.setF1(x1);
03: obj.setF2(x2);
04: obj.CoverMe();

Fig. 19. An illustrative example for method sequence generalization.

MSeqGen can perform well in scenarios where existing code bases that use object types of MUT are available, but is not effective in scenarios where no such code bases are available. For example, if a class $c$ is newly written, it is not possible to find code bases using the class $c$. Furthermore, mined sequences may not include all necessary method calls required for producing desired object states.

3.1.2.4 Implementation-Based Technique: Seeker. The implement-based category is a broad category and we classify techniques that do not belong to any of the previous categories as implementation-based category. We next present more details about a technique, called Seeker [45], that belongs to this category. We use the previous code example shown in Fig. 7 as an illustrative example for explaining Seeker.

Given a desired object state, described in the form of a target branch, Seeker attempts to generate a method sequence that produces the desired object state. For example, consider the code example shown in Fig. 20. Consider that the desired
Client Code:

```java
public static void foo(BFSAlgorithm udfs) {
    ...
    if(udfs.GetIsComputed()) {
        ...   //B6
    }
    //B7
}
```

Fig. 20. A desired object state expressed as Branch B6.

```java
Vertex s1 = new Vertex(0);
BidirectionalGraph ag = new BidirectionalGraph();
ag.AddVertex(s1);
ag.AddEdge((IVertex)s1, (IVertex)s1);
BFSAlgorithm ud = new BFSAlgorithm(ag);
ud.Compute((IVertex)null);
```

Fig. 21. An example method sequence.

object state is that the `isComputed` field is `true` (expressed as Branch B6). Seeker automatically generates a sequence (shown in Fig. 21) that covers Branch B6, thereby producing a desired object state.

The key idea of Seeker is its ability to intelligently navigate the large search space of candidate sequences by synergistically combining static and dynamic analyses. Seeker includes the following three main steps: (1) dynamic analysis generates method sequences to cover branches; (2) static analysis uses dynamic analysis information for not-covered branches to generate candidate sequences; and (3) dynamic analysis explores and eliminates statically generated sequences. We next present more details about dynamic and static analyses that form a feedback loop for systematically exploring large search spaces.

**Dynamic Analysis.** Given a target branch `tb`, Seeker first applies dynamic analysis such as DSE to generate a sequence that covers `tb`. If DSE happens to generate a target sequence, a sequence for covering `tb`, Seeker terminates. Otherwise, Seeker retrieves and analyzes covered (`CovB`) and not-covered branches (`NotCovB`) by DSE and applies static analysis on these branches. For example, when DSE is applied on the code example shown in Fig. 20, DSE generates a sequence that helps cover Branch B7, but not Branch B6. The reason is that DSE could not generate a
target sequence that can help cover B6. After applying DSE, CovB and \( NotCovB \) include \{B7\} and \{B6\}, respectively.

After exploration using DSE, there can be three possible scenarios for the target branch.

- **Scenario 1:** The target branch \( tb \) is covered. In this scenario, Seeker terminates.
- **Scenario 2:** The target branch \( tb \) is not covered and \( tb \in NotCovB \). This scenario happens when DSE successfully covers the alternative branch of \( tb \) and could not cover \( tb \). In this scenario, Seeker uses static analysis to generate a sequence that can help cover \( tb \).
- **Scenario 3:** The target branch \( tb \) is not covered and \( tb \notin NotCovB \). This scenario happens when DSE could not cover all the dominant branches of \( tb \) in the method \( m \). In this scenario, Seeker identifies the dominant branch of \( tb \), referred to as prime dominant, whose alternative branch is covered by DSE and attempts to cover all branches starting from prime dominant to \( tb \).

**Static Analysis.** Seeker next uses static analysis to suggest candidate sequences that can help cover \( tb \). Given a \( tb \), Seeker first identifies the target member field \( tfield \) that needs to be modified to produce a desired object state for covering \( tb \). It is trivial to identify \( tfield \) for branches such as \( \text{if}(\text{stack.size} == 10) \), where \( tfield \) (such as size) is directly included in the branch. However, in object-oriented code, branches often involve method calls such as \( \text{if}(!\text{verticesList.Contains(v1)}) \) in Statement 8 (Fig. 7) rather than fields. It is challenging to identify target fields in the presence of method calls, since the return statements in these method calls may in turn include further method calls, where the actual member field is returned.

To address this issue, Seeker uses an interprocedural execution trace (hereby referred to as \( \text{trace} \)), gathered during the runtime exploration with DSE. This trace includes the statements executed in each method. Seeker performs backward analysis of the trace starting from the method call involved in \( tb \). We use \( retvar \) to refer to the variable or value associated with the return statement in a method call. Seeker uses the following five steps with respect to \( retvar \) to identify \( tfield \).

1. If \( retvar \) is a member field, Seeker identifies \( retvar \) as \( tfield \). This scenario can happen with methods such as getter methods.
2. If \( retvar \) is data dependent on a member field, Seeker identifies that member field as \( tfield \).
3. If \( retvar \) is data dependent on the return of a nested method call, Seeker repeats these five steps with the nested method call to identify \( tfield \).
4. If \textit{retvar} is control dependent on a member field, Seeker identifies that member field as \textit{tfield}. This scenario can happen when DSE failed to generate other object states for that member field.

5. If \textit{retvar} is control dependent on the return of a nested method call, Seeker repeats these five steps with that nested method call to identify \textit{tfield}. The method \texttt{HasElements} (Statements 7–10 in Fig. 22) shows an example of this scenario, where \textit{retvar} is control dependent on the return of another nested method call \texttt{queue.size()}. In this scenario, Seeker repeats the preceding five scenarios with that method call \texttt{queue.size()).

We next explain these steps using a simpler example shown in Fig. 22. To illustrate these five steps, consider Branch B8 as \textit{tb}. Given this \textit{tb}, Seeker applies the preceding steps and detects \texttt{size} (in \texttt{ArrayList}) as \textit{tfield}. Here, Queue includes a member field \texttt{list} of type \texttt{ArrayList}. Initially, Seeker analyzes the method \texttt{IntQueue.HasElements}. Since the executed return statement (Statement 9) is control dependent on a nested method call \texttt{queue.size()}, Seeker analyzes the

```
1 public class IntQueue {
2     private Queue queue;
3     public IntQueue() {
4         this.queue = new Queue; }
5     public void Enqueue(int item) {
6         queue.Enqueue(item); }
7     public bool HasElements() {
8         if(queue.size() > 0) { return true; }
9         else { return false; }
10     }
11 }
12 public class MyCls {
13     private IntQueue intq;
14     public MyCls(IntQueue intq) {
15         this.intq = intq; }
16     public void MyFoo() {
17         if(intq.HasElements()) {
18             ...//B8
19         }
20     }
21 }
```

Fig. 22. An integer queue class.
Queue.size method. Eventually, Seeker reaches the getter method that returns _size member field of ArrayList and thereby identifies _size as tfield.

Along with identifying tfield, Seeker also captures two other pieces of information. First, Seeker identifies the condition on tfield that is not satisfied. For example, Seeker identifies "_size > 0" (Statement 8) as the condition that should be satisfied to cover tb. Seeker applies a constraint solver on the preceding condition to get a desired value for tfield. Second, Seeker also captures the hierarchy of fields, referred to as field hierarchy, that includes all objects starting from the object enclosing tb to tfield. For Branch B8 as tb, the identified field hierarchy is as follows: “FH: MyCls root ⇒ IntQueue intq ⇒ Queue queue ⇒ ArrayList list ⇒ int_size”. This field hierarchy describes that _size of type int is contained in the object list of type ArrayList, which is in turn contained in the object queue of type Queue and so on. Here, root represents the object of type MyCls. This field hierarchy is used for identifying candidate methods as discussed next.

Given a target field tfield such as _size, its current and desired values, and field hierarchy, Seeker identifies pretarget branches that need to be covered to cover the original target branch tb. Initially, Seeker traverses the field hierarchy and identifies the object tobj (in the field hierarchy) that can be modified to achieve a desired value for tfield. The objective of this traversal is to identify the tobj that is nearest to tfield and can be modified by either assigning a value directly or by invoking its public methods. The reason is that the shorter the distance between tobj and tfield, the smaller the amount of code that needs to be explored to achieve a desired value for tfield. For example, consider the preceding field hierarchy FH. Here, the object ArrayList list is near to _size (tfield) compared with CodeIn Queue queue. However, the list object cannot be tobj, since list, a private member field, cannot be modified directly or by invoking its public methods.

To identify tobj, Seeker traverses the field hierarchy from root and considers the object whose next object cannot be modified either directly or through public methods as tobj. For example, root is not considered as tobj, since intq can be modified as intq is set through the constructor. For this field hierarchy, Seeker identifies intq as tobj, since queue cannot be modified outside the intq object.

After identifying tobj, Seeker identifies methods (and pretarget branches within those methods) that help produce a desired value for tfield. Identifying the methods of tobj that modify tfield is nontrivial, since there can be intermediate objects between tobj and tfield, as identified by the field hierarchy. To address this issue, Seeker uses a novel technique based on method-call graphs.

A method-call graph is a directed graph that includes caller–callee relations among methods. Figure 23 shows a sample method-call graph constructed for the field
hierarchy $FH$. The root node of the graph includes $tfield$. The first level of the graph includes the methods (in the declaring class) that modifies $tfield$. Initially, Seeker statically analyzes all public methods of the declaring class of $tfield$ to identify the target methods that modify $tfield$. In particular, Seeker identifies assignment statements, where $tfield$ is on the left hand side. For example, Seeker identifies the methods such as $Add$, $Insert$, and $Reset$ of $ArrayList$ as target methods for the $tfield \_size$. From the second level, the graph includes the methods from the declaring classes of fields in the field hierarchy. The graph includes an edge from a method $M_b$ in one level to a method $M_c$ in the next level, if $M_b$ is called by $M_c$. For example, $Queue.Enqueue$ invokes the $ArrayList.Add$ method and the corresponding edge is shown from Levels L1 to L2.

Seeker next traverses the constructed graph from the top to bottom to identify methods that can be invoked on $tobject$ to achieve a desired value for $tfield$. Furthermore, Seeker identifies pretarget branches within each method that needs to be covered to invoke the method call of the preceding level. For example, Seeker identifies that the $IntQueue.Enqueue$ method can help achieve a desired value for $tfield$. Furthermore, Seeker identifies the pre-target branch in $IntQueue.Enqueue$ that helps invoke $Queue.Enqueue$ method. This pretarget branch is considered as a new target branch that needs to be covered, so as to cover the original target branch $tb$. Seeker returns several candidate pretarget branches. After identifying these pretarget branches, Seeker applies dynamic analysis to generate method sequences that cover these pretarget branches and use those sequences to generate a method sequence that covers the original target branch $tb$.
3.1.3 Open Issues

In test-input generation at the unit-testing level, one issue is to deal with illegal test inputs (also called invalid test inputs), which are test inputs that the unit under test is not expected to handle. Adopting the design-by-contract methodology [7], Parasoft Jtest, CodePro AnalytiX, and Microsoft Pex (when used in combination of Microsoft Contracts [48]) allow developers to specify method preconditions or class invariants for the unit under test, and their test generation engines would filter out or avoid generating those test inputs that violate method preconditions or class invariants. Microsoft Pex also allows developers to write assumptions for parameterized unit tests (PUTs) [27] (unit test methods with parameters, more details described in Section 3.2), and test inputs for PUTs violating assumptions are filtered out or avoided in test-input generation.

Some tools such as Agitar AgitarOne adopt a defensive programming methodology, where developers are advocated to write explicit checking code in the beginning of a method body of the unit under test to detect illegal test inputs and (once detected) throws appropriate exceptions. These tools’ test-input generation engines would still generate illegal test inputs.

One direction to address challenges faced in test-input generation is to allow developers to guide tools in different ways. For example, developers can specify data factories for a nonprimitive object type. Such data factories are called test-input factories or test helpers in Agitar AgitarOne, object repositories in Parasoft Jtest, and factory classes or methods in CodePro AnalytiX and Microsoft Pex. As another example, CodePro AnalytiX and Parasoft Jtest allow developers to directly edit the generated test inputs to improve them. Researchers [49,50] have also explored techniques for exploiting information from manually written unit tests in guiding the generation of new test inputs. Microsoft Pex can allow developers to write parameterized unit tests [27] (e.g., where developers can write desirable method sequences with primitive values being unspecified and generated by Pex).

3.2 Test Oracles

There are two main levels of test oracles: ones specific only to one individual test input and ones applicable to multiple test inputs.

3.2.1 Test Oracles Specific to Individual Test Inputs

Test oracles can be in the form of assertions in manually written unit tests (such as those in JUnit [3]). Developers can relatively easily write assertions for one test input
but writing them for many unit-test inputs (which can be generated by tools) is time consuming and infeasible.

In the regression testing context, similar to the capture-and-replay technique described in Section 2, tools can use the capture-and-assert technique [51], which captures the return values of methods of the unit under test during the execution of the generated unit-test inputs, and then automatically creates assertions based on the captured return values. Parasoft Jtest, CodePro AnalytiX, Agitar AgitarOne, and Microsoft Pex implement such technique for regression testing. A resulting test case with automatically created assertions is called a characterization test by Feathers [52].

In general testing (beyond regression testing), tools such as Parasoft Jtest, CodePro AnalytiX, and Microsoft Pex allow developers to inspect and verify captured assertions in the generated unit test suite. To reduce inspection effort, these tools allow to select only test inputs that can achieve new code coverage (such as statement coverage and block coverage) not previously achieved.

3.2.2 Test Oracles Applicable to Multiple Test Inputs

Test oracles specific to multiple test inputs can be classified into three types. First, developers can determine whether the execution of a test input fails based on whether the execution throws an uncaught exception. Such type of test oracles is related to robustness testing, being supported by all industrial tools. Note that this type of test oracles is quite limited since the execution may throw no uncaught exceptions but produce wrong outputs (such as wrong method-return values).

Second, developers can write properties in the unit code under test based on the design-by-contract methodology [7], where properties can be in the form of method preconditions, method postconditions, and class invariants. Parasoft Jtest, CodePro AnalytiX, and Microsoft Pex (when used in combination of Microsoft Code Contracts [48]) support such type of test oracles. Agitar AgitarOne allows developers to specify class invariants. Developers may have a more difficult time in writing one single property than writing one single assertion in unit tests (as in the first level of test oracles) but a single property can be used to check the execution of multiple test inputs.

To reduce the difficulty of writing properties and stimulate developers to write properties, Agitar Agitator implements a software agitation technique [53] based on dynamic invariant detection [54]. It infers observations of the code behavior from the execution of automatically generated test inputs. These observations summarize common behavioral patterns reflected by the execution of multiple test inputs. Then developers can inspect and verify these observations: if these observations reflect desirable behaviors, developers promote them to be assertions (such as method
postconditions and class invariants); if these observations reflect faulty behaviors, developers detect faults and then fix these faults.

Third, developers can write properties in unit test code such as parameterized unit tests (PUTs) [27], which are test methods with parameters. Developers can write assumptions (similar to preconditions) and assertions (similar to postconditions) in PUTs; however, assumptions or assertions in PUTs are often not specified for only one specific method but for a scenario where multiple methods are invoked. Both Microsoft Pex and Agitar AgitarOne support parameterized unit tests. To some extent, assertions specified in PUTs can be viewed as a middle ground between assertions specified in traditional unit tests and properties specified in unit code under test, in terms of test oracles’ fault-detection capability, ease of writing, and scope of benefits.

4. Cooperative Developer Testing

Manual developer testing is known to be labor intensive and insufficient. To reduce manual effort in developer testing, testing tools can be employed to automate activities in developer testing (such as test execution and test input generation), enabling economical use of resources. To maximize the value of developer testing, effective and efficient support for cooperation between developers and tools is greatly needed. In particular, developer-testing research and practice is in a great need of (1) effective ways for developers to communicate their testing goals and guidance to tools and (2) techniques and tools with strong enough capabilities to accomplish the given testing goals. To meet this need, recent research starts to explore a new research frontier on synergistic cooperation between developers and tools, which is called cooperative developer testing [55].

The methodology of cooperative developer testing consists of three phases: (1) Setup phase: developers prepare the testing environment and perform testing activities with initial inputs; (2) Feedback phase: tools provide feedback to developers; (3) Action phase: developers provide guidances to tools based on the feedback. The feedback phase and the action phase form a feedback-action loop that enables developers and tools to refine and accomplish testing goals for various testing activities. This cooperation between developer and tools facilitates the developers’ understanding of the program under test, improving quality of the program and reducing cost of fault fixing.

In this section, we present state-of-art industry practices and academic research approaches that illustrate how cooperative developer testing can be applied in various testing activities, including test execution and test-input generation.
4.1 Test Execution

Developer testing activities typically include generating test inputs, specifying expected outputs, executing test inputs, and verifying actual outputs. Developers can use the xUnit family of testing frameworks (JUnit [3] for Java, NUnit [4] for C#, and CppUnit [6] for C++) to manually write test inputs and provide their expected outputs. These frameworks automate the activities of executing test inputs and verifying actual outputs against the expected outputs and report the passing and failing tests as the results.

Figure 24 shows an example test method `testPush` that tests the `push` method of `Stack`. This test method is written using JUnit 4 [3]. A JUnit test method consists of two major parts: manually provided test inputs and expected results. The manually provided test inputs are written at Lines 17 and 18: creating a `Stack` object and pushing a number into the stack. The expected results are written using the assertion methods provided by JUnit to check the expected results against the actual results, as shown at Line 19. Developers can execute these JUnit tests using tools like eclipse [56]. During test execution, JUnit uses the checkable comment (also called

```java
public class Stack{
    private ArrayList data = new ArrayList();
    public void push(Object o) {
        data.add(o);
    }
    public boolean empty(){
        if(data.size() == 0){
            return true;
        }
        return false;
    }
    ...
}
@Test
public void testPush() {
    Stack s = new Stack();
    s.push(1);
    Assert.assertEquals(1, s.size());
}
```

Fig. 24. Example JUnit test case [3] for the `Stack`. 
annotation) @Test shown at Line 15 to identify test methods and reports passing and failing tests.

The passing tests give developers high confidences in the program under test, whereas the failing tests indicate that some parts of the program implementation do not conform to the expected behaviors. Based on the failing tests, developers may locate which parts of the program cause the failures and provide fixes to the program. Then, developers can re-execute the same tests to verify whether the changed program produces expected outputs. This feedback and fixing loop refines the program implementation based on the testing goals, capturing faults in the early development stage and significantly reducing cost for fault fixing in later stages of software development. When these xUnit testing frameworks are applied in a test-first approach [57], the development process, called TDD [1], becomes first creating failing tests and then producing a program implementation to pass these tests. TDD relies on the repetition of a very short development cycle, intended to improve quality of the program and responsiveness to changing requirements. As one of the best practices of agile software development [58], TDD expedites the feedback and fixing loops between developers and automated test execution.

Although testing improves quality of the program, the confidences gained from the passing tests are on the parts of the program that are tested. To identify the insufficiency of the tests (e.g., which parts of the programs are not tested), structural coverage that measures the coverage of statement and branches [59] is introduced as another kind of data collected during test execution. There already exist various industrial tools for measuring structural coverage for the tests written in xUnit frameworks, such as Cobertura [60] for JUnit and NCover [61] for NUnit. For example, in Fig. 24, the test method testPush achieves 100% statement coverage and branch coverage of the method push (there are no branches).

The achieved structural coverage guides developers to create tests to cover the not-yet-covered parts of the programs, resulting in a loop of test execution and test creation until sufficient coverage is achieved. For example, the test method testEmpty in Fig. 25 does not cover the false branch at Line 7 or the statement at Line 10. By looking at this coverage information and using their knowledge of the program, developers know that they need to provide a nonempty stack object to

```java
@Test
public void testEmpty() {
    Stack s = new Stack();
    Assert.assertIsTrue(s.empty());
}
```

Fig. 25. A JUnit test case that tests the empty method of Stack using an empty stack.
@Test
public void testEmpty() {
    Stack s = new Stack();
    s.push(1);
    Assert.assertIsFalse(s.empty());
}

Fig. 26. A JUnit test case that tests the empty method of Stack using a nonempty stack.

achieve the coverage of the false branch at Line 7. Figure 26 shows such a test case that creates a nonempty stack for testing the method empty. By executing the test cases shown in Figs. 25 and 26, developers can see that these test cases achieve full statement and branch coverages of the method empty. This loop of test execution and test creation illustrates how developers and test-execution tools cooperate to achieve testing goals.

4.2 Test-Input Generation

Producing high-covering test inputs is an important goal of software testing, since high structural coverage can help identify the insufficiency of test inputs, e.g., showing which parts of programs are not tested by the test inputs. To reduce the manual burden of manually producing test inputs, developers can use tools built based on automated test-generation approaches to generate test inputs automatically, such as random testing [25,34,35] and dynamic symbolic execution (DSE) [30,31].

Random testing [25,34,35] randomly selects test inputs in a given range. Random testing is scalable and able to find previously identified faults automatically [25]. Random testing usually achieves lower coverage compared to a systemic test generation approach such as DSE, because certain branches in the program under test may have a low probability of being covered by randomly generated test inputs.

DSE [30,31] executes the program under test symbolically with arbitrary or default inputs. Along the execution path, DSE collects the constraints in the branch statements to form a path condition and negates part of the path condition to obtain a new path condition that leads to a new path. The new path condition is then fed into a constraint solver, such as Z3 [33], which computes new test inputs for exploring new paths.

Although these automated test-generation tools can easily achieve high structural coverage on simple programs, they face different kinds of challenges to achieve high structural coverage on complex programs in practice, especially for object-oriented programs. Based on recent studies [23,35,55], the top two major problems
public class FixedSizeQueue {
    private Queue queue;
    public FixedSizeQueue(Queue queue) {
        this.queue = queue;
    }
    public void enqueue(Object item) {
        if (queue.size() == 10) {
            throw new Exception("full");
        }
        queue.enqueue(item);
    }
    ...
    }
    public void testEnqueue(FixedSizeQueue queue, Object item) {
        queue.enqueue(item);
    }
}

Fig. 27. FixedSizeQueue implemented using Queue.

that prevent these tools from achieving high structural coverage are (1) the object-
creation problem (OCP), where tools fail to generate sequences of method calls to
construct desired object states for covering certain branches; (2) the external-method-
call problem (EMCP), where tools cannot deal with method calls to external libraries,
such as native system libraries or precompiled third-party libraries.

Based on the recent studies [23,55] that apply automated test-generation tools
to generate test inputs for achieving high structural coverage, OCPs are the top
major problems that cause automated test-generation tools not to achieve high
structural coverage. The main reason is that certain branches of the program under
test require desired object states that cannot be generated by the tools. Consider
the FixedSizeQueue class shown in Fig. 27. A parameterized unit test [27]
testEnqueue is provided to test the enqueue method. To achieve full structural
coverage (statement and branch coverages) of enqueue, the tools need to generate
an empty queue and a queue whose size is 10 as two test inputs. Since the private
field queue can be modified only by invoking enqueue, the tools may not be able to
generate the sequence of method calls that enqueue 10 objects into queue, causing
the true branch at Line 7 not to be covered.

3 Nearly half of the not-covered branches of the program under test are caused by OCPs [23,55].
EMCPs are the second major problems that cause automated test-generation tools not to achieve high structural coverage. The main reason is that external-method calls cannot be precisely analyzed by the tools or throw exceptions to hinder the test generation. Consider the example external-method calls shown in Fig. 28. At Line 3, the return value of `defaultFile.exists()` is used to decide the boolean value of the conditional expression. If the tools cannot generate a desired value for the argument `fileName` that represents an existing file in the testing environment, the true branch at Line 3 cannot be covered. As another example, the external-method call `file.getAbsolutePath()` at Line 10 throws exceptions for invalid arguments, preventing the tools from exploring the remaining parts of the program under test if no valid test inputs are generated.

Since tools are imperfect in dealing with various challenges in achieving high structural coverage, Xiao et al. [55] present an approach that precisely identifies problems faced by tools during test generation (with the focus on OCPs and EMCPs), enabling developers and tools to generate test inputs cooperatively as follows. Developers first apply tools to automatically generate test inputs until tools cannot achieve higher structural coverage or run out of predefined resources. Then, the tools report the achieved coverage and problems that prevent them from achieving...
higher coverage. By looking into the problems, developers provide guidance to tools, helping tools address these problems. After providing guidance to the tools based on the reported problems, developers can reapply tools to generate test inputs for achieving better coverage. Such iterations of applying tools and providing guidance can continue until satisfied coverage is achieved.

As an example of providing guidance to tools, developers can provide factory methods that encode sequences of method calls to produce desired object states to deal with OCPs [12]. Consider the OCP shown in Fig. 27. To cover the true branch at Line 7 of the method push, developers can provide a factory method to create a Queue object whose size is 10 to the tools. Since the field queue can be assigned via the public constructor of FixedSizeQueue, the tools can easily generate a desired FixedSizeQueue object by using the provided Queue object as the argument for the constructor of FixedSizeQueue. Similarly, to deal with EMCPs, developers can instruct tools to instrument and explore the external libraries or provide mock objects [18] to simulate the dependences. For example, developers can mock the file system library call defaultFile.exists shown in Fig. 28 and make defaultFile.exists to return true for covering the true branch at Line 3.

To achieve this cooperation between developers and test-generation tools, the tools need to precisely report problems for reducing effort from developers. Straightforward approaches such as locating all nonprimitive object types and external method calls produce too many irrelevant problem candidates that do not prevent tools from achieving higher structural coverage. For example, in Fig. 28, the external-method calls String.format at Line 15 only format the input strings and do not affect the achieved coverage.

To address the needs of precisely identifying problems, Xiao et al.’s approach [55] prunes the irrelevant problem candidates using the data dependencies of partially covered branch statements on problem candidates. Their approach is built based on the insight that partially covered branch statements have data dependency on real problem candidates. Consider the method enqueue shown in Fig. 27. Since the conditional expression at Line 7 using the queue.size value of the filed queue, the branches at Line 7 have data dependencies on the field queue. If the true branch at Line 7 is not covered, their approach can correctly report an OCP of Queue, which is the object type of the field queue. Similarly, in Fig. 28, the branches at Line 3 have data dependencies on the external-method call defaultFile.exists.

4 Nonprimitive types refer to composite types recursively constructed from basic data types (e.g., int, double, and boolean).
5 Branch statements contain one or more not-covered branches.
If the true branch at Line 3 is not covered, their approach reports an EMCP of `defaultFile.exists`.

Xiao et al. implemented their approach as an extension to Pex [12] and evaluated their approach on two open source projects: xUnit [5] and QuickGraph [39]. The results show that their approach effectively identified problems and pruned irrelevant problem candidates with low false positives and false negatives. Their work demonstrates that cooperation between developers and automated test-generation tools is possible and should be further pursued in both research and practical directions.

### 4.3 Summary

The applications of cooperative developer testing in the two testing activities, test execution and test-input generation, illustrate how the methodology of cooperative developer testing can be used to effectively and efficiently accomplish testing goals. Each of the these applications can be summarized using the three phases: Setup phase, Feedback phase, and Action phase, as shown in Table I.

Currently, the application of cooperative developer testing in test execution is already well studied and broadly adopted in industry, and the application in test-input generation starts to gain attentions from researchers and expect to have impacts in industry with the development of tools. Besides these promising results, the cooperation between developers and tools should be further explored in both research and practical directions to maximize the value of developer testing.

<table>
<thead>
<tr>
<th>Application</th>
<th>Setup</th>
<th>Feedback</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test execution</td>
<td>Writing initial test suites using automated test-execution frameworks, such as xUnit family</td>
<td>Passing and failing tests with achieved structural coverage</td>
<td>Fixing failing tests and writing new tests for not-covered parts of the program under test</td>
</tr>
<tr>
<td>Test-input generation</td>
<td>Applying automated test-generation tools to generate test inputs for the program under test without any human intervention</td>
<td>Problems that prevent the tools from achieving high structural coverage</td>
<td>Providing guidance to help tools address problems, such as factory methods for OCPs and mock methods for EMCPs</td>
</tr>
</tbody>
</table>
5. Conclusion and Future Directions

In this chapter, we presented an overview of techniques implemented in testing tools to address challenges in improving automation in developer testing. In particular, we focused on a recent promising technique, called symbolic execution, and further discussed various challenges in generating test inputs automatically. For test inputs, we summarized the techniques from two main aspects: test efficiency and test effectiveness. In addition, we presented a new frontier, called cooperative developer testing, that is orthogonal to previous techniques and involves synergistic cooperation between humans and tools for effectively generating desirable test inputs.

Although much progress has been made to improve automation in developer testing, many challenges faced by the automated approaches and tools still need to be further explored in both research and practice directions. Based on the advances of techniques around parameterized unit tests (PUTs) [62] and dynamic symbolic execution (DSE) [12,30,31], we next present some promising future directions of developer testing along four dimensions: correctness confidence, specifications, (dis)integration testing, and human factors [63].

5.1 Correctness Confidence

One open question in software testing is how high confidence testers would have on program correctness after a certain amount of testing is conducted. It is a common belief that testing cannot provide high confidence on program correctness. For example, consider a PUT, which includes assertions for asserting correctness for the program under test. Let us assume that (1) the PUT (together with the invoked program under test) has a finite number of feasible paths, (2) DSE explores all these feasible paths, and (3) the constraints collected in the path condition from each iteration are within the capability of the underlying constraint solver.

Now consider that DSE is applied to generate test inputs for the PUT. If DSE finds no violations of an assertion in the PUT, there is in fact 100% correctness confidence with respect to the assertion. However, when testing real programs in practice, DSE cannot establish 100% correctness confidence due to various challenges such as path explosion and complex logics in path conditions. In this case, how do we measure and report to the developers the level of correctness confidence after DSE is applied (when it is not 100%, which could be often the time in practice)? For a measured and reported level, how do we validate that it reflects the real level? One possible direction is to adapt or customize traditional code coverage [8] to take into account of the assertion under consideration and the difficulties faced by DSE. Another direction is to use static verification to complement testing [64,65].
5.2 Specifications

White-box testing has been known to be ineffective in detecting omission faults, which are related to missing functionalities. To address this problem, a PUT or the program under test with assertions in combination of a white-box test-generation tool (such as one based on DSE) can be classified as an integrated form of both white-box and black-box testing. Assertions in a PUT or the program under test can be seen as a form of specifications. In practice, these specifications may be missing. Then could inferred specifications [66,67] be good enough for serving as test oracles? AgitarOne [10] recommends specification candidates for developers to confirm, which are often limited. However, these inferred specification candidates could serve as stimulus for encouraging developers to write more specifications, complementing the recommended specification candidates.

5.3 (Dis)integration Testing

In real-world code bases, a component could have quite some dependencies on external environments such as file systems and network sockets. EMCPs introduced in Section 4 are typical examples of dependencies on external environments. A common solution is to mock or simulate environment dependencies [68–71], so that the unit tests run quickly and give deterministic results. We name such testing as disintegration testing, since it aims to break the integration, being contrary to integration testing. There exist tool support [72] for isolating the environment. Developers could also spend one-time effort for writing parameterized models [68,69,71] for a specific environment to faithfully simulate the behavior of the environment. An open question related to (dis)integration testing is how to measure the quality of parameterized models for faithfully simulating an environment. That is, is there any methodology for systematically modeling the environment with parameterized models? To tackle these questions, modeling techniques in model-based testing [73] can be borrowed. Another question is how to make smooth transition from disintegration testing to integration testing by exploiting the knowledge gained in unit testing of isolated components? One possible direction [71] is to automatically synthesize a real-environment state for the simulated environment state after it is generated during disintegration testing; then the real-environment state could be used during integration testing.

5.4 Human Factors

Tool automation such as improving automated test generation has been a traditional focus of software testing research. At the same time, human factors also play important roles in software testing. Taking human factors together with tool
automation, we advocate a new methodology of cooperative developer testing, as described in Section 4. This methodology should be further explored in both research and practical directions to maximize the value of developer testing.

Our discussion so far has primarily focused on functional correctness. However, all or most of our discussed aspects could be also applicable for other quality attributes such as security and performance. Indeed, these other quality attributes would call for further new future directions. Additionally, our discussion has been primarily on testing sequential code. Concurrency testing [74,75] is another interesting direction in developer testing and would call for further new research directions.

REFERENCES


**About the Authors**

**Xusheng Xiao** is a Ph.D. student at North Carolina State University. He received an M.S. in Computer Science from North Carolina State University and a B.E. in Software Engineering from Chongqing University. He has worked as a research intern in IBM T.J. Watson Research Center, Microsoft Research Redmond, and Microsoft Research Asia. He has published two ICSE papers on improving software quality and security using dynamic/static program analysis approaches or natural language processing approaches. His research work of access control policies in software systems is selected as the finalist for the 2011 annual National Security Innovation Competition.

**Suresh Thummalapenta** is a member of the Programming Technologies and Software Engineering department at IBM Research—India. Suresh received his Ph.D. in Computer Science from North Carolina State University in 2010. Before that, he received an M.S from North Carolina State University and a B.Tech. from Andhra University, both in computer science. His research interests include Automated
Software Engineering with emphasis on Software Testing and Mining Software Engineering Data. His main goal is to develop techniques and tools that can help deliver high-quality software more productively.

Tao Xie is an Associate Professor in the Department of Computer Science at North Carolina State University. He received his Ph.D. in Computer Science from the University of Washington in 2005. Before that, he received an M.S. in Computer Science from the University of Washington in 2002, an M.S. in Computer Science from Peking University in 2000, and a B.S. in Computer Science from Fudan University in 1997. He has worked as a visiting researcher at Microsoft Research Redmond and Microsoft Research Asia. His research interests are in software engineering, including software testing and analysis and software analytics.