A Comparison of Constraint-based and Sequence-based Generation of Complex Input Data Structures

Rohan Sharma, Milos Gligoric, Vilas Jagannath, and Darko Marinov

Department of Computer Science
University of Illinois
Urbana IL, 61801, USA
{*sharma27, gliga, vbangal2, marinov*}*@illinois.edu*

***Abstract*—Generation of complex input data structures is one of the challenging tasks in testing. Manual generation of such structures is tedious and error-prone. Automated generation approaches include those based on constraints, which generate structures at the concrete representation level, and those based on sequences of operations, which generate structures at the abstract representation level by inserting or removing elements to or from the structure. In this paper, we compare these two approaches for five complex data structures used in previous research studies. Our experiments show several interesting results. First, constraint-based generation can generate more structures than sequence-based generation. Second, the extra structures can lead to false alarms in testing. Third, some concrete representations of structures cannot be generated only with sequences of insert operations. Fourth, slightly different implementations of the same data structure can behave differently in testing.**

I. Introduction

Constraints are increasingly used in software testing, verification, and analysis. A lot of recent progress has been made on using constraints for primitive types and some data structures such as lists or trees. However, more complex data structures remain challenging for full code verification that attempts to prove certain code properties for all possible input sizes. A promising alternative is to use *bounded-exhaustive testing* [1]–[4] to test the code for all inputs up to a given bound.

In this paper, we compare two approaches to bounded-exhaustive test generation of complex data structures. The goal of both approaches is to create data structures to be used as test inputs, and more specifically to create all such structures within a given bound. An important question to ask in this domain is which structures are generated. (Another important question is the time required for generation, but we do not consider it in this paper). Various approaches could generate different structures and result in testing that has false negatives (i.e., it misses bugs) or false positives (i.e., it reports false alarms that are not actual bugs).

Consider, for instance, the often used example of testing an operation on a set implemented as a red-black tree. The operation could be adding an element to the set (inserting it into the tree) or removing an element from the set (removing it from the tree). We need to create several test inputs, i.e., sets/trees to invoke the operation on each of them, and to check the resulting output and the modified set/tree. The bound for a tree would be the number of nodes. If we do not generate some valid tree within a given bound, we could potentially miss a bug that can be found within that bound. On the other hand, if we generate an input that is not a valid tree, and the operation fails for that input, we could incorrectly report that there is a bug.

In *constraint-based generation*, with tools such as Korat [1] or TestEra [2], the structures are created at the concrete representation level. The user encodes an invariant for data structures and provides bounds, while a tool automatically generates the structures. In the example with red-black trees, the user encodes which object graphs are valid red-black trees (e.g., the nodes should have no sharing and should be properly colored either red or black), and the tool generates all such structures with a given number of nodes. The advantage of this approach is that it can test each operation in isolation, e.g., we can test the remove operation even if we do not have the add operation. The disadvantage is that it relies on the user to provide an invariant. The invariant might not correctly represent the desired data structures.

In *sequence-based generation*, the structures are generated from sequences of operations. For instance, to generate a set, we would create an empty set and then apply a sequence of add (and potentially remove) operations. To generate all sets, we need to explore various sequences up to some length. Effectively, this generation is done at the abstract representation level and need not consider the concrete representation. However, the concrete representation is typically considered to prune generation: if two different sequences result in the same concrete state, we need not extend both of these sequences into longer ones but can consider only one of the two sequences. The advantage of this approach is that it generates only the states that are actually reachable by sequences of the operations. The disadvantage is that it requires testing several operations at once and does not guarantee that all structures were explored. For example, can sequences of n add operations generate all red-black trees up to size n? (The answer is in Section III).

We compare constraint-based and sequence-based generation for five complex data structures, chosen for two reasons. First, the relationship between constraint-based and sequence-based approaches was not obvious for these structures. Second, all of them were used in previous research studies [5], [6], and constraints/invariants for these structures are readily available in Java code [7] or pseudocode [8]. We did not use some other structures from previous studies, such as linked lists or unbalanced binary search trees or binomial heaps, because they do not have a variety of shapes [9], and it was obvious that constraint-based and sequence-based generation should produce the same structures.

Our experiments show several interesting results. First, constraint-based generation can generate more structures of a given size than sequence-based generation, i.e., the *commonly used* constraints/invariants for these data structures (soundly) over-approximate the set of all possible states reachable through sequences of operations. Therefore, if a bug in some operation cannot be found for any structure generated from constraints, it cannot be found for any structure generated with a sequence of operations (within the same bound). Second, the extra structures from constraint-based generation can lead to false alarms. Third, some concrete representations of structures cannot be generated with sequences of only insert/add operations. Namely, to generate all possible structures with up to n nodes, we have to use some remove operations and thus sequences with more than n operations. Fourth, slightly different implementations of the same data structure can behave differently in testing. In our example, two independent implementations of a red-black tree based on the same textbook algorithms [8] provided different results. In summary, these results show that sequence-based generation does not subsume constraint-based generation, and using constraint-based generation provides benefits even when testing several operations at once.

II. Background

This section describes constraint-based and sequence-based generation in more detail. We use the red-black tree data structure as our driving example. Figure 1 shows a snippet of red-black tree code written in Java. Each RedBlackTree object represents a red-black tree, which has a `root` node and `size` nodes reachable from the root. Each `Node` object stores a `value` and has pointers to the `left` and `right` children, a pointer to the `parent` node, and a `color` that can be either red or black. This class provides several operations/methods to create an empty tree, insert an element into the tree, remove an element from the tree, and query whether the tree contains an element. For brevity, we do not show implementation of these methods, but we will discuss their relevant parts later on in the text.

```
class RedBlackTree {
  Node root; int size;
  static boolean RED = false, BLACK = true;
  static class Node {
    Node left, right, parent;
    boolean color; int value;
  }
  RedBlackTree() { /* empty tree */ }
  void insert(int value) { ... }
  void remove(int value) { ... }
  boolean contains(int value) { ... }
}
```

Figure 1. RedBlackTree implementation

A. Constraint-Based Generation

To generate all red-black trees with a given number of nodes using a tool like Korat [10], the user needs to provide the following inputs:

- Declaration of the data structure to be generated, i.e., all the fields that are used to represent an instance of the data structure. For the RedBlackTree in Figure 1, these are the fields in classes `RedBlackTree` and `Node`.
- Constraints that encode which instantiations of the field values forms a valid data structure. In Korat, these constraints are encoded as a Java method, commonly called `repOk`, which takes as input the instantiation of the fields and returns `true` for a valid instantiation and `false` otherwise. Figure 2 shows the `repOk` method for the RedBlackTree class. This method checks that (1) the underlying object graph forms a tree along the `left` and `right` fields, (2) the node colors satisfy the properties for red-black trees, and (3) the values in the tree are ordered for binary search (values to the left/right of a node are smaller/larger than its value).
- Finitization parameters that bound the values that can be used to instantiate the fields. Details of writing finitizations can be found elsewhere [7], [11]. Intuitively, a finitization specifies the size of the data structures to generate and the values in the nodes.

Given these inputs, Korat generates all the valid instantiations of the data structure fields within the provided finitizaton bounds. More precisely, Korat generates only non-isomorphic structures [2], [11], [12], where two structures are defined to be isomorphic if they differ only in the identity of the nodes in the structures but have these nodes connected in the same (rather, isomorphic) way and have the same primitive values within the nodes. In most of our comparisons, we will consider trees that have *exactly* n nodes, although one can easily generate all trees whose size is *up to* some given n.

For example, running Korat on this `RedBlackTree` and `repOk` for 5 nodes generates 14 red-black trees. This is indeed the number of all red-black trees of size 5 [13].

We do not present in detail how Korat generates these structures. A naive implementation would explore the entire finite state space encoded in the finitization by enumerat-

```
boolean repOk() {
  return treeness() && coloring() && ordering();
}
boolean treeness() {
  if (root == null) return size == 0;
  Set<Node> visited = new HashSet<Node>();
  visited.add(t.root);
  List<Node> workList = new LinkedList<Node>();
  workList.add(t.root);
  if (root.parent != null) return false;
  while (!workList.isEmpty()) {
    Node current = workList.removeFirst();
    Node cl = current.left;
    if (cl != null) {
      if (!visited.add(cl)) return false;
      if (cl.parent != current) return false;
      workList.add(cl);
    }
    Node cr = current.right;
    if (cr != null) {
      if (!visited.add(cr)) return false;
      if (cr.parent != current) return false;
      workList.add(cr);
    }
  }
  return size == visited.size();
}
boolean coloring() {
  // Part 1): red node must have black children
  ...
  // Part 2): number of black nodes on all paths is the same
  int numberOfBlack = −1;
  List<Pair> workList = new LinkedList<Pair>();
  workList.add(new Pair(root, 0));
  while (!workList.isEmpty()) {
    Pair p = workList.removeFirst();
    Node e = p.e; int n = p.n;
    if (e != null && e.color == BLACK) n++;
    if (e == null) {
      if (numberOfBlack == −1) numberOfBlack = n;
      else if (numberOfBlack != n) return false;
    } else {
      workList.add(new Pair(e.left, n));
      workList.add(new Pair(e.right, n));
    }
  }
  return true;
}
boolean ordering() { ... }
```

Figure 2. repOk for RedBlackTree

ing every single candidate structure and checking whether `repOk` returns `true` for that structure. Korat uses optimizations to prune this search by monitoring executions of the `repOk` method and by avoiding isomorphic structures. The interested reader can find more details elsewhere [10], [11].

We point out that Korat was used previously both for generating data structures for testing library classes [6] and for testing real applications [14], [15]. Our focus in this paper is on data structures such as red-black trees. The constraints to be encoded in `repOk` for such structures are readily available, e.g., in data structure and algorithms textbooks [8]. Many such `repOk` methods were written and used in previous studies [6], [7], [11].

The key question that we ask in this study is how the structures generated based on these *naturally written* `repOk` methods compare to the structures that can be generated by sequences of operations. Note that in theory we can always write `repOk` methods that do not properly encode data structures, in our example red-black trees. In the limit

```
class RedBlackTreeDriver {
  static void main(String[] args) {
    // number of nodes
    int numOfNodes = ...;
    // length of sequence (and method parameters)
    int lenOfSeq = ...;

    RedBlackTree rbt = new RedBlackTree();
    for (int i = 0; i < lenOfSeq; i++) {
      switch (Verify.getInt(0, 1)) {
      case 0:
        rbt.insert(Verify.getInt(0, lenOfSeq − 1));
        break;
      case 1:
        rbt.remove(Verify.getInt(0, lenOfSeq − 1));
        break;
      }
    }
    if (rbt.size == numOfNodes) {
      printStructure(rbt);
    }
  }
}
```

Figure 3. JPF driver for generating RedBlackTrees

on one side, we can have a `repOk` method that returns `false` for every input. Such a method does not admit any structure and is not useful for testing as it is vacuously true that for all structures for which such `repOk` returns `true`, the code under test behaves correctly. In the limit on the other side, we can have a `repOk` method that returns `true` for every input. Such a method admits any graph as a "red-black tree" and is also not useful for testing. For example, it could create a graph that has a cycle for which the code under test does not terminate, although it is not a bug in the code under test but a "bug" in the input (rather a violation of the precondition that the input should not have a cycle).

B. Sequence-Based Generation

Another approach to systematically create data structures such as red-black trees is through sequences of method calls [16], [17]. Figure 3 shows sample code that can be used to achieve this. The code is written in Java with the extensions from Java PathFinder (JPF), a popular model-checking tool for Java [18], [19]. Specifically, the JPF method `Verify.getInt(int lo, int hi)` non-deterministically chooses an integer between `lo` and `hi`, inclusive. Note that we use the same method to choose between the methods `insert` and `remove`, as well as to choose the values for these methods. JPF can explore all possible combinations of these non-deterministic choices. In our example, this exploration corresponds to generating all bounded sequences of methods calls (where the length is up to `lenOfSeq`) to `insert` and `remove` with bounded parameters (where the values range between 0 and `lenOfSeq`).

A feature of JPF that we do not explicitly show in Figure 3 is that it is an explicit-state model checker that can compare the state of structures being generated. Therefore, it need not explore literally all possible sequences up to `lenOfSeq`: if it finds two different sequence prefixes that result in the same state for `rbt`, it does not explore various sequence suffixes for both prefixes but only for one. Additionally, it does not print the same structure more than once. We point out that the structure comparison that JPF performs for this code is based on the same isomorphism as used in Korat [11]. Also, note that we use Korat and JPF simply as two tools that implement constraint-based and sequence-based generation, respectively, but we draw no specific conclusions about the tools but rather about the generation approaches.

For example, running JPF on this code for both `lenOfSeq` and `numOfNodes` being 5 generates 6 trees. However, increasing `lenOfSeq` to 7 while keeping `numOfNodes` to 5 generates 8 trees. Recall that Korat produces 14 red-black trees with 5 nodes. There are several interesting questions about these numbers. Why do the numbers of generated structures differ for different `lenOfSeq`, while `numOfNodes` is the same? Why does constraint-based generation produce more trees than sequence-based generation? What happens for the code under test for those extra trees? The next section answers these questions.

III. Comparison

In this section we describe the methodology that we used to compare the constraint-based and sequence-based generation approaches introduced in the previous section. We also present the results of the comparison and our analysis of those results.

A. Methodology

We compared the approaches using five data structures: RedBlackTree, AVLTree, DisjSet, FibonacciHeap, and HeapArray. The RedBlackTree data structure is the same as presented in Section II. The AVLTree data structure is also a balanced search tree but its balancing relies on heights rather than node coloring. DisjSet is an implementation of the disjoint set data structure. The FibonacciHeap data structure is a binary heap represented as a group of trees. The HeapArray data structure is also a binary heap but implemented using an array. The Java code for the RedBlackTree, AVLTree, and DisjSet was written by an undergraduate researcher (the first paper author) using the classical CLRS book as a reference [8]. The Java HeapArray code was obtained from the Korat repository [7], and the Java FibonacciHeap code was obtained from the JGraphT project repository [20].

For each of the data structures we took the `repOk` method from the Korat repository or implemented one based on the CLRS book. Note that these `repOk` methods were written "naturally" and not specifically for this study. We also wrote finitization methods to generate structures using Korat, and we implemented JPF drivers to generate structures using method sequences. We then generated all the data structures for various sizes using both Korat and JPF. While generating the data structures using method sequences, we used different sequence lengths as illustrated in Section II-B.

B. Results and Analysis

Figure 4 shows the results of the comparison. The first column of the table shows the name of the structures. The second column, *size*, shows the sizes (measured by the number of nodes) of the structures that are generated. The third column, *lenOfSeq*, shows the length of method sequences used to generate structures with the JPF driver. The fourth column shows the number of structures generated with the JPF driver, and the fifth column shows the number of structures generated using Korat. The numbers demonstrated several interesting results.

(1) Constraint-based generation can generate more structures than sequence-based generation. This occurs for DisjSet and RedBlackTree. While analyzing this result, we found and fixed a bug in our DisjSet implementation. Even after that, the naturally written `repOk` for DisjSet generates more structures, because it is non-trivial to exactly characterize reachable structures when the code uses so called *path compression* [8]. In our analysis of the `repOk` for RedBlackTree, we found that it admits structures whose root nodes are colored *red*, while this is not possible using method sequences. In other words, the constructor for the empty tree colors the root node *black*, and all methods (`insert`, `remove`, and `contains`) maintain the invariant that the root node be *black*. Stated yet differently, while the *specification* for red-black trees is *non-deterministic* and allows some nodes to have any color, each *implementation* is *deterministic* and chooses some specific color.

(2) Constraint-based generation of more structures can result in false alarms during testing. When we tested the `insert` method of the RedBlackTree using the structures generated by Korat, we encountered a `NullPointerException` failure. However, this was a false positive caused by a structure with a red root. Since our RedBlackTree implementation never created a red root, it never had to handle structures that have a red root.

(3) Slightly different implementations of the same data structure can behave differently in testing. While we were initially surprised with the above exception and thought that our `insert` implementation had a bug, our analysis convinced us it was a false alarm. But we then asked why we never encountered the same exception in (modified versions of) `java.util.TreeMap`, which is Sun's Java library code that implements maps using red-black trees. That code has been used in testing research for years, at least since TestEra [2]. We knew from the comments in that code that the developers used a previous edition of the same CLR(S) book as the reference. So, have we (and others) been missing some false alarm for almost ten years? Our inspection shows it was not the case. Rather, the `java.util.TreeMap` code can handle red roots, although method sequences cannot create them. Effectively, Sun's developers used defensive programming [21] to guard against the "impossible" case.

(4) Sequence-based generation may require sequences longer than n to generate all (reachable) structures with n nodes. For RedBlackTree and FibonacciHeap, since many structures cannot be generated with sequences of just `insert` methods, sequences longer than the structure size with both `insert` and `remove` methods were required to generate all reachable structures. As discussed above, sequence-based and constraint-based generation produce different structures for RedBlackTree. However, for FibonacciHeap, sequence-based generation, *with appropriately long sequences*, can produce all the structures produced by constraint-based generation. The problems are that (1) these sequences may need to be quite long (e.g., the generation for FibonacciHeap size 3 requires that lenOfSeq 7 to produce all 5 structures), (2) the length of the sequences to use is not obvious a priori, and (3) it is not even obvious what methods need to be used (e.g., is it necessary to use `remove`). In comparison, constraint-based generation can directly focus on structures of a given size.

In summary, the examples show four cases of relationship between constraint-based and sequence-based generation:

1) They generate the same structures of size n with sequences of length n (and hence only `insert` suffices). This is the case for AVLTree and HeapArray.
2) They can generate the same structures but longer sequences are required (and some `remove` methods may be necessary). This is the case for FibonacciHeap.
3) Constraint-based generates more structures but the code under test works fine for them. This is the case for `java.util.TreeMap` and DisjSet.
4) Constraint-based generates more structures that lead to false alarm(s) in testing. This is the case for our RedBlackTree.

The only remaining relationship would be that `repOk` constraints under-approximate the set of states reachable with sequences of method calls and thus testing with constraint-based generation could result in missed bugs. While we did not encounter this case among our examples, it could be possible for even more complex data structures where it is hard to characterize the set of reachable states or where one would not want to use bounded-exhaustive enumeration of the entire sets but to start sampling.

structure	size	sequence based		constraint based
		lenOfSeq	#structures	#structures
RedBlackTree	3	{3, 4}	**1**	3
		{5, 6, 7}	**2**	
	4	{4, 5, 6, 7, 8}	**4**	8
	5	{5, 6}	**6**	14
		{7, 8, 9}	**8**	
AVLTree	3	{3, 4, 5, 6, 7}	1	1
	4	{4, 5, 6, 7, 8}	4	4
	5	{5, 6, 7, 8, 9}	6	6
HeapArray	2	{2, 3, 4}	1	1
	3	{3, 4}	2	2
	4	{4}	2	2
FibonacciHeap	2	{2, 3}	**1**	2
		{4, 5}	2	
	3	{3, 4}	**1**	5
		{5, 6}	**3**	
		{7}	5	
	4	{4, 5}	**1**	14
		{6, 7}	**5**	
		{8}	**9**	
	5	{5, 6}	**1**	42
		{7}	**7**	
DisjSet	3	{3, 4, 5, 6}	**10**	16
	4	{4, 5, 6}	**65**	125
	5	{5, 6}	**436**	1296

Figure 4. Comparison of number of generated structures

IV. RELATED WORK

Generation of complex data structures is an active area of research in software testing. Two approaches include constraint-based [1], [2] and sequence-based [22]–[24] generation. We are not aware of any previous comparison of these approaches as presented in this paper.

We will now briefly review related work on sequence-based generation. Khurshid et al. [25] introduced generalized symbolic execution to generate complex data inputs with the goal of obtaining better coverage. Pasareanu, Visser, et al. [17], [26], [27] extended this work using model checking (in particular JPF) and abstractions, as applied to container data structures. Randoop [28] is a sequence-based method that generates random sequences of methods calls, and d'Amorim et al. [29] compared random generation and symbolic execution. Note that our comparison focused on bounded-exhaustive testing. Newer approaches are more sophisticated [30], [31]. For example, MSeqGen [30] mines code bases and generates a sequence of method calls based on the object on which the methods are called and the arguments used. While all these solutions strive for better code coverage, bounded-exhaustive sequence-based generation strives for better input space coverage.

V. CONCLUSION

We presented a comparison of constraint-based and sequence-based generation of complex input data structures. Although we used only five complex data structures, we obtained several interesting results that illustrate the relationship between these two approaches. In brief, the results show that constraint-based generation is highly valuable for bounded-exhaustive generation, but can result in false alarms during testing. Sequence-based generation does not require writing constraints but makes it hard to provide guarantees about bounded-exhaustive generation. In the future, we plan to work on combining these two approaches, e.g., if constraint-based generation finds some interesting test structures, can we use sequence-based generation to search for method sequences that create these structures.

ACKNOWLEDGEMENTS

This material is based upon work partially supported by the National Science Foundation under Grant Nos. CCF-0746856 and CCF-0916893. Milos Gligoric was supported by the Saburo Muroga fellowship.

References

[1] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated testing based on Java predicates," in *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, 2002.

[2] D. Marinov and S. Khurshid, "TestEra: A novel framework for automated testing of Java programs," in *Proc. 16th International Conference on Automated Software Engineering (ASE)*, 2001.

[3] B. Daniel, D. Dig, K. Garcia, and D. Marinov, "Automated testing of refactoring engines," in *Proc. 6th joint meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2007.

[4] K. Sullivan, J. Yang, D. Coppit, S. Khurshid, and D. Jackson, "Software assurance by bounded exhaustive testing," in *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, 2004.

[5] B. Elkarablieh, D. Marinov, and S. Khurshid, "Efficient solving of structural constraints," in *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, 2008.

[6] D. Marinov, A. Andoni, D. Daniliuc, S. Khurshid, and M. Rinard, "An evaluation of exhaustive testing for data structures," MIT CSAIL, Tech. Rep. MIT-LCS-TR-921, 2003.

[7] "Korat home page," http://mir.cs.illinois.edu/korat/.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*. The MIT Press, 2009.

[9] M. d'Amorim, S. Lauterburg, and D. Marinov, "Delta execution for efficient state-space exploration of object-oriented programs," in *Proc. International Symposium on Software Testing and Analysis (ISSTA 2007)*, 2007.

[10] A. Milicevic, S. Misailovic, D. Marinov, and S. Khurshid, "Korat: A tool for generating structurally complex test inputs," in *Proc. 30th International Conference on Software Engineering, Demo Papers (ICSE Demo)*, 2007.

[11] D. Marinov, "Automatic testing of software with structurally complex inputs," Ph.D. dissertation, MIT, 2004.

[12] R. Iosif, "Exploiting heap symmetries in explicit-state model checking of software," in *Proc. 16th International Conference on Automated Software Engineering (ASE)*, 2001.

[13] N. Sloane and S. Plouffe, *The Encyclopedia of Integer Sequences*. Academic Press, 1995, http://www.research.att.com/njas/sequences/.

[14] K. Stobie, "Model based testing in practice at microsoft," *Electronic Notes in Theoretical Computer Science*, vol. 111, 2005.

[15] S. Misailovic, A. Milicevic, N. Petrovic, S. Khurshid, and D. Marinov, "Parallel test generation and execution with Korat," in *Proc. 6th joint meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2007.

[16] W. Visser, C. S. Pasareanu, and S. Khurshid, "Test input generation with Java PathFinder," in *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, 2004.

[17] W. Visser, C. S. Pasareanu, and R. Pelanek, "Test input generation for red-black trees using abstraction," in *Proc. 20th International Conference on Automated Software Engineering (ASE)*, 2005.

[18] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *Automated Software Engineering*, vol. 10, no. 2, 2003.

[19] "JPF home page," http://babelfish.arc.nasa.gov/trac/jpf/.

[20] "JGraphT home page," http://jgrapht.sourceforge.net/.

[21] B. Liskov and J. Guttag, *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2000.

[22] C. Csallner and Y. Smaragdakis, "JCrasher: An automatic robustness tester for Java," *Software Practice and Experience*, vol. 34, no. 11, 2004.

[23] C. Pacheco and M. D. Ernst, "Eclat: Automatic generation and classification of test inputs," in *Proc. 19th European Conference on Object-Oriented Programming (ECOOP)*, 2005.

[24] T. Xie, D. Marinov, and D. Notkin, "Rostra: A framework for detecting redundant object-oriented unit tests," in *Proc. 19th International Conference on Automated Software Engineering (ASE)*, 2004.

[25] S. Khurshid, C. Pasareanu, and W. Visser, "Generalized symbolic execution for model checking and testing," in *Proc. 9th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2003.

[26] W. Visser, C. S. Pasareanu, and R. Pelanek, "Test input generation for red-black trees using abstraction." in *Proc. 20th International Conference on Automated Software Engineering (ASE)*, 2005.

[27] W. Visser, C. S. Pasareanu, and S. Khurshid, "Test input generation with Java PathFinder," in *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, 2004.

[28] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proc. 29th International Conference on Software Engineering (ICSE)*, 2007.

[29] M. d'Amorim, C. Pacheco, T. Xie, D. Marinov, and M. D. Ernst, "An empirical comparison of automated generation and classification techniques for object-oriented unit testing," in *Proc. 21st International Conference on Automated Software Engineering (ASE)*, 2006.

[30] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "MSeqGen: Object-oriented unit-test generation via mining source code," in *Proc. 7th joint meeting of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2009.

[31] P. Tonella, "Evolutionary testing of classes," in *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, 2004.