An Empirical Study of Structural Constraint Solving Techniques

Junaid Haroon Siddiqui and Sarfraz Khurshid

The University of Texas at Austin
Austin, TX 78712
{jsiddiqui,khurshid}@ece.utexas.edu

Abstract. Structural constraint solving allows finding object graphs that satisfy given constraints, thereby enabling software reliability tasks, such as systematic testing and error recovery. Since enumerating all possible object graphs is prohibitively expensive, researchers have proposed a number of techniques for reducing the number of potential object graphs to consider as candidate solutions. These techniques analyze the structural constraints to prune from search object graphs that cannot satisfy the constraints. Although, analytical and empirical evaluations of individual techniques have been done, comparative studies of different kinds of techniques are rare in the literature. We performed an experiment to evaluate the relative strengths and weaknesses of some key structural constraint solving techniques. The experiment considered four techniques using: a model checker, a SAT solver, a symbolic execution engine, and a specialized solver. It focussed on their relative abilities in expressing the constraints and formatting the output object graphs, and most importantly on their performance. Our results highlight the tradeoffs of different techniques and help choose a technique for practical use.

Keywords: Empirical comparison, software testing tools, model checking, symbolic execution, SAT, state space exploration, systematic testing.

1 Introduction

Generating test inputs for programs that manipulate structurally complex inputs like XML documents or red black trees is a complex operation. Manual generation of these tests is time consuming, error prone, and has fairly limited ability to find bugs whereas systematic testing, which is effective at finding bugs, is not straightforward as there are no simple enumerators for structurally complex inputs.

Automated generation of structurally complex test inputs can be done in two basic ways: using generator functions [52, 51] and by solving constraints [5, 38]. Generator functions are functions that perform basic operations to construct and build structures (e.g., constructors or mutator methods in Java). Automated testing using generator functions typically uses different orderings of generator functions to produce test inputs. This can however result in the same structures repeated, i.e., redundant tests, and some kinds of structures may never be

K. Breitman and A. Cavalcanti (Eds.): ICFEM 2009, LNCS 5885, pp. 88–106, 2009.

produced. Generator functions are mostly applied for generating larger inputs effectively.

Automated testing by solving structural constraints [5,38] enables *systematic testing* where the program is tested against all test inputs within given bounds. Even though doing so is feasible only for small bounds, it has been shown to give high code coverage and find faults in programs with structurally complex inputs [32, 38, 49]. Also, by writing constraints we can conveniently describe a whole class of structurally complex test inputs. In this paper, we discuss the techniques that can be used for systematic testing based on structural constraint solving.

The structural constraints used by systematic testing techniques are usually written either as declarative constraints or as imperative constraints. Alloy [30] (one of the techniques discussed here) uses declarative constraints written in relational logic using quantified formulas. The other three techniques that we evaluate use imperative constraints. We call them *imperative* in contrast to *declarative* as they use constraints written in an imperative language (C or Java in our case). We note that these imperative constraints are required to be free of side-effects and hence are declarative in nature (even though they are written in an imperative language).

The contribution of this paper is a controlled experiment for performance analysis of different constraint solving techniques. It also performs an analysis to quantify the tradeoffs of these techniques in writing constraints and in processing outputs. Our results show that even though generic techniques like model checkers and symbolic execution can be used to solve structural constraints, specialized solvers are faster in solving and need the least tweaking of code to work.

The rest of the paper is organized as follows. We provide an overview of the problem of constraint solving in the following subsection, give a background on different techniques and how they solve structural constraints in Section 2. Section 3 describes our experiment; the subjects, analysis strategy, and threats to validity. We discuss experimental results and our analysis in Section 4 and summarize and conclude in Section 5.

1.1 Related Work

The idea of using constraints for representing test inputs has been used for at least three decades [11, 28, 35, 43] and implemented in EFFIGY [35], TEST-GEN [36], and INKA [24] among other tools. However most of this work was to solve constraints on primitive data like integers and not structural constraints.

Goodenough and Gerhart [23] discuss the importance of specification based testing. Test case generation has been automated from specifications by many tools. Some examples are from Z specifications [15], UML statecharts [41], ADL specifications [9], and AsmL specifications [25]. However these specifications are also targeted to primitive types and not structurally complex inputs.

Constraints on complex structures require very different constraint solving techniques, which have only been explored more recently. Directions of research include using model checkers [20, 50], SAT solvers [47], symbolic execution [21, 44], and specialized solvers [5]. Section 2 discusses each of these techniques, their background and recent advancements.

One common problem faced while generating complex structures is isomorphism [45]. Two structures are defined to be isomorphic if they only differ in object identities. For example, if all elements in two nodes of a tree are swapped and all references to these nodes are swapped too, the resulting structure is identical to the original except that pointer values in some nodes would be different. Since, most programs are not concerned with the actual pointer *values* and only with *where* they are pointing, generating isomorphic structures is considered redundant and the algorithms attempt isomorph breaking procedures to reduce generated structures.

For the purpose of comparison and explaining how constraints are written in different approaches, we will take red-black tress [3, 26] as our running example. We pick this representative example as it is one of the more complex structures, one of the structures commonly used for evaluation in previous work, and one that is likely to be familiar.

2 Background of Subject Tools

2.1 JPF — Model Checker

Model checking [10] has traditionally been applied to software [2, 13, 50, 27] for checking event sequences, specified in temporal logic or as a finite state machine of API usage rules. If a program is checked successfully, no input and execution can lead it to an error. Thus model checking provides a strong guarantee. However these techniques did not consider checking properties and validity of complex structures. The model checkers BLAST and SLAM are also used for white-box test input generation [4] targeting to cover specific predicates. The two are also not applied to solving complex structural constraints.

Generalized Symbolic Execution [34] introduced the idea of using a model checker for solving structural constraints. As an enabling technology, the JPF (Java Path Finder) model checker [50] was used. JPF is an explicit-state model checker for Java programs that has been used to find errors in a number of complex systems [42, 6, 1]. It is built on top of a custom Java Virtual Machine (JVM). Therefore it handles all standard Java features and in addition allows nondeterministic choices written as annotations. These annotations are added by method calls to class `Verify`. The following methods in this class are important:

- `randomBool()` returns a nondeterministic boolean value
- `random(n)` returns a nondeterministic integer in [0,`n`]
- `ignoreIf(cond)` makes JPF backtrack if `cond` is true

Generalized symbolic execution provides a source-to-source translation of a Java program such that it can be symbolically executed by any standard model checker that supports non-deterministic choice. The technique of generalized

```
class RedBlackTree {
    ...
    static Node[] nodes;
    static int maxNode = 0;
    boolean header_accessed = false;
    Node header;
    Node header() {
        if (!header_accessed) {
            header_accessed = true;
            if (maxNode < nodes.length - 1) {
                maxNode++;
                int r = Verify.random(maxNode);
                if( r != maxNode )
                    maxNode--;
                header = nodes[r];
            } else header = nodes[ Verify.random( maxNode ) ];
        }
        return header;
    }
    boolean repOk() {
        if (header() == null)
            return false;
        Set<Node> visited = new java.util.HashSet<Node>();
        visited.add(header());
        LinkedList<Node> workList = new LinkedList<Node>();
        workList.add(header());
        while (!workList.isEmpty()) {
            Node current = workList.removeFirst();
            if (current.left() != null) {
                if (!visited.add(current.left()))
                    return false;
                workList.add(current.left());
            }
            if (current.right() != null) {
                if (!visited.add(current.right()))
                    return false;
                workList.add(current.right());
            }
        }
        if (visited.size() != size() || size() < LOWER_BOUND )
          return false;
        return repOkColors() && repOkKeys();
    }
}
```

Fig. 1. Parts of Red Black Tree predicate written for JPF

symbolic execution is based on *lazy initialization*, i.e. it initializes fields when they are first accessed during symbolic execution of a method. Due to this lazy initialization, the algorithm only executes program paths on non-isomorphic inputs. This can be used for systematic generation of structurally complex inputs by symbolically executing a predicate checking structural constraints.

Figure 1 shows parts of Red Black Tree predicate written for JPF. Note that all accesses to structure variables are through accessors functions. One accessor function for `header` is also shown. It non-deterministically picks one of the nodes that have already been used or one of the new nodes.

Recently, this technique has been optimized by making modifications to Java Path Finder [19]. However these optimizations are specific to one model checker, whereas the original technique can be used on any model checker.

2.2 Alloy — Using a SAT Solver

SAT solvers solve boolean formulas. To use SAT solvers for solving structural constraints, we thus need a language for writing structural constraints, a compiler to translate that language into a boolean formula, and a mapping from the solution of the boolean formula into a solution to the structural constraint.

Alloy [29] provides a declarative language for writing these constraints. It is based on parts of the Z specification [48]. The Alloy Analyzer [31] provides a fully automated tool to solve these constraints using a SAT solver. The latest version of Alloy Analyzer (4.1.10) works with many state-of-the-art solvers like BerkMin [22], MiniSat [47], SAT4J (Java implementation of MiniSat), and ZChaff [40]. Alloy analyzer provides a translation from the declarative language of Alloy with quantifiers to a boolean formula when given bounds. It then translates the solution back to the declarative language.

TestEra [33] builds on Alloy to translate the solutions further back into actual Java structures. TestEra also adds a layer on top of Alloy language to facilitate writing preconditions and postconditions, and allows test case generation based on preconditions and function validation using its postconditions as an oracle. However for the purpose of constraint solving alone, Alloy is sufficient. The Alloy to Java translator component of TestEra can be used to translate Alloy solutions into Java structures. The translation time is insignificant in comparison to the constraint solving time.

We show class invariant for red-black trees modeled in Alloy in Figure 2. Note that this completely models red black trees. Addition of a few more syntactic sugar like definition of `Node` etc is all that is needed to generate all possible red black trees within given bounds. This concise representation is one of the key benefits of using a declarative language. However the learning curve of declarative

```
all e: rbt.root.*(left+right) |
      // BT: distinct children
      ( no e.(left+right) || e.left != e.right ) &&
      // BT: acyclic
      ( e ! in e.^(left+right) ) &&
      // BT: distinct parent
      lone e.~(left + right) &&
      // BST: ordered
      lt[ e.left.*(right+left).key, e.key ] &&
      gt[ e.right.*(right+left).key, e.key ] &&
      // RBT: red node has black children
      ( e.color in Red && some e.(left + right)
        => e.(left + right).color in Black )

all e, f: rbt.root.*(left+right) |
      // RBT: all paths from root to NIL have same # of black nodes
      (no e.left || no e.right) && (no f.left || no f.right) =>
        #{ p: rbt.root.*(left+right) |
          e in p.*(left+right) && p.color in Black } =
        #{ p: rbt.root.*(left+right) |
          f in p.*(left+right) && p.color in Black }
```

Fig. 2. Red Black Tree constraint written for Alloy

programming for programmers used to program in imperative languages often offsets this benefit. The bounds for Alloy are written as below:

```
run test for 1 rbt, exactly 3 Node
```

The class invariant requires the tree to satisfy binary search tree properties and the additional properties of red-black trees mentioned in comments in Figure 2. The reader is referred to Jackson [29] for detailed discussion of Alloy operators and syntax and to Guibas [26] for red-black tree properties.

2.3 CUTE — Symbolic Execution

The idea of symbolic execution dates back at least three decades [35]. Traditional symbolic execution is a combination of static analysis and theorem proving. In symbolic execution, operations are performed on symbolic variables instead of actual data. On branches, symbolic execution is forked with opposite constraints on symbolic variables in each forked branch. At times, the constraints on symbolic variables can become unsatisfiable signaling unreachable code. Otherwise, end of the function is reached and a formula on symbolic variables is formed. A solution to this formula will give a set of values that will direct an actual execution along the same path.

Renewed interest in symbolic execution is seen in the last decade [7, 12, 18]. Generalized Symbolic Execution [34] extended the concept to concurrent programs and complex structures.

The main problem with symbolic execution is that for large or complex units, it is computationally infeasible to maintain and solve the constraints required for test generation. Larson and Austin [37] combined symbolic execution with concrete execution to overcome this limitation. Their approach was primitive as they used symbolic execution to make the path constraint of a concrete execution and find other input values that can lead to errors along the same path.

DART (Directed Automated Random Testing) [21] is one of the first tools to systematically combine symbolic execution and concrete execution. Similar to previous approach, they formed a path constraint during concrete execution. However after the execution, they backtrack on the path constraint by negating clauses, solve the new constraints, and re-run concrete execution expecting it to follow a new path. When it is not feasible to solve the modified constraints, they substitute random concrete values. Another simultaneous effort was EGT (Execution Guided Test Cases) [8] using a similar approach. Lastly, CUTE (Concolic Unit Testing Engine for C) [44], another tool using similar approach, is the tool that we will be using here. It is the only tool that can handle pointers and complex structures.

The idea of using CUTE to generate test cases has been briefly discussed but not evaluated [44]. There, the authors considered `prev` pointers in a doubly linked list and discussed the order (big O) of candidates CUTE and Korat (discussed below) explore to find answers. In our evaluations we thoroughly cover this example among others. In particular, we discuss the constants involved (time of

```
int repOk( struct bintree* b ) {
  struct listnode* visited=0, *worklist=0;
  int NODES = 0;
  if( b->root == 0 )
      return 0;
  visited = newnode( b->root, visited );
  ++NODES;
  worklist = newnode( b->root, worklist );
  while( worklist ) {
    struct node* current = worklist->data;
    worklist = worklist->next;
    if( current->left ) {
      if( !addunique( visited, current->left ))
        return 0;
      ++NODES;
      worklist = newnode( current->left, worklist );
    }
    if( current->right ) {
      if( !addunique( visited, current->right ))
        return 0;
      ++NODES;
      worklist = newnode( current->right, worklist );
    }
    if( NODES > UPPER_BOUND )
      return 0;
  }
  if( b->size != vcount || NODES < LOWER_BOUND)
    return 0;
  return repOkColors(b) && repOkKeys(b);
}
```

Fig. 3. Parts of Red Black Tree predicate written for CUTE

exploring one candidate) and constraint rewriting requirements to understand which approach is likely better in practical usage.

We show parts of the red-black tree constraint written in C for use in CUTE in Figure 3. The `NODES` variable is introduced to keep a count of nodes used. We break the loop when more than `UPPER_BOUND` nodes have been touched and `return false` if less than `LOWER_BOUND` nodes were touched during the execution. This is how we control the desired number of objects when generating structures in CUTE. Rest of the constraint is similar to what was shown in Figure 1.

2.4 Korat — A Specialized Solver

Korat [5] is a framework for automated generation of structurally complex test inputs. It performs *specification based testing*. By using a Java predicate that represents properties of desired inputs, Korat uses backtracking search and explores the input space of the predicate and enumerates inputs for which the predicate returns true. Each enumerated inputs is a desired structurally complex test input. Korat performs *bounded exhaustive testing*: it generates all non-isomorphic test cases within given bounds. Bounded exhaustive testing has been used to successfully find bugs in a fault-tree analyzer [49], a resource discovery architecture, and an XPath compiler.

Korat performs a dynamic analysis of the predicate. It prunes huge portions of the input space by monitoring field accesses during predicate execution. It

```
public boolean repOK() {
    if (root == null)
        return false;
    Set<Node> visited = new HashSet<Node>();
    visited.add(root);
    LinkedList<Node> workList = new LinkedList<Node>();
    workList.add(root);
    while (!workList.isEmpty()) {
        Node current = workList.removeFirst();
        if (current.left != null) {
            if (!visited.add(current.left))
                return false;
            workList.add(current.left);
        }
        if (current.right != null) {
            if (!visited.add(current.right))
                return false;
            workList.add(current.right);
        }
    }
    if (visited.size() != size)
        return false;
    return repOkColors() && repOkKeys();
}
```

Fig. 4. Parts of Red Black Tree predicate written for Korat

```
IFinitization f = FinitizationFactory.create(RedBlackTree.class);

IClassDomain entryDomain = f.createClassDomain(Node.class, numEntries);
IObjSet entries = f.createObjSet(Node.class, true);
entries.addClassDomain(entryDomain);

IIntSet sizes = f.createIntSet(minSize, maxSize);
IIntSet keys = f.createIntSet(-1, numKeys - 1);
IIntSet colors = f.createIntSet(0, 1);

f.set("root", entries);
f.set("size", sizes);
f.set("Node.left", entries);
f.set("Node.right", entries);
f.set("Node.color", colors);
f.set("Node.key", keys);
```

Fig. 5. Korat's specification of bounds for Red Black Tree

backtracks on the last field accessed and makes a non-deterministic assignment to that field. It then uses the new candidate to re-execute the predicate.

Korat, being a specialized solver, produces correct output for every predicate (`repOk`), however it is written. Although, some predicates would cause a faster execution (return after touching as few fields as possible) and some would be slower (return once after checking all checks that can be checked), none would result in an incorrect result. We here show a portion of red-black tree constraint written for Korat in Java in Figure 4. We also show how bounds are given for Red Black Tree in Korat's finitization in Figure 5.

The principle idea of Korat has been used in other applications. In particular, STARC [16] uses the Korat algorithm to repair huge complex structures by running the algorithm in neighborhood of the defective structure. Glass box testing [14] uses the method to be tested to prune Korat's generation. Thus it moves away from the pure black-box approach of Korat.

Korat has been optimized in a number of ways. Instead of running `repOk` from the start for every candidate, efficient backtracking optimization [17] can undo operations done in last execution and proceed from that point for the next candidate. This has shown improvements for STARC and also for Korat. Lastly, Korat has been parallelized for clusters of largely independent machines by random division of work [39] and for high bandwidth clusters by systematic division of work [46].

2.5 Research Questions

The effectiveness of bounded exhaustive testing (generating all test cases satisfying the constraints) has been previously shown in application to many real applications. Here we are concerned with different tools to generate these tests. Thus we are not concerned with the fault detecting capability of these tools, as this capability would be equal (given sufficient time) for all tools in our scenario. We are rather concerned with how to write the tests and interpret the output and most importantly how much time it takes to generate the tests.

We pose the following research questions for our experiment and analysis:

- What are the pros and cons of different tools in writing constraints and defining bounds?
- How is the output of a tool represented and how it can be converted into actual test inputs?
- What are the fastest tools for practical sizes of subject structures?
- How well do the tools perform with more and more complex constraints?
- What are the best tools in terms of time complexity?

Next we describe our experiment and its analysis.

3 The Experiment

3.1 Experimental Subjects

To evaluate the selected tools, we consider six complex structures: three list structures, and three tree structures. Note that these complex structures are the foundation of several data structures used in applications. For example, an XML document, a file system hierarchy, Java or C class hierarchies, expression trees, abstract syntax trees for compiler can all be viewed as trees and are likely to give similar performance to one of the tree structures we consider here. We evaluate the following six structures:

1. Binary Tree
2. Binary Search Tree
3. Red Black Tree
4. Singly Linked List
5. Doubly Linked List
6. Sorted Linked List

Note that a red-black-tree is a binary search tree which is in turn a binary tree. From this, we intend to learn the effect of increasing constraint complexity on tool performance.

To avoid any bias, we took constraints for the above subjects from previous work [5], where available. In some cases, we needed to change the constraints so that the tool under evaluation performs bounded exhaustive testing (as discussed in the previous section).

3.2 Experimental Design

The experiment focused on:

1. Structurally complex constraints (6 constraints of subjects given in previous section)
2. Bounds (we considered 4 bounds for each subject structure)
3. The constraint solver (one of the four constraint solvers discussed in this paper)

On each run, we measured:

1. Time taken to generate all tests
2. Candidates generated to see isomorphism pruning

We also measure qualitative results for:

1. How constraints needed to be converted to run the tool
2. How bounds needed to be converted to run the tool
3. How results from the tool needed to be converted to test cases

Results reported for the experiment were averages of 10 repeated measurements. Thus, for each subject structure and each constraint solver and each given bounds, we ran the tool 10 times and computed the average. The experiments were performed on a Linux machine with Intel Pentium 4 2.8Ghz processor and 4GB RAM.

3.3 Threats to Internal Validity

Threats to internal validity are influences that can affect dependent variables without researcher's knowledge. In this respect, our concerns include the way constraints are written and language differences. Constraints can be written to suit one tool and not the other. We have done our best effort is writing the constraints so that every tool can perform at its best. Language differences matter because one of the tools works in C while the rest work in Java. C implementations are inherently faster so the results of this tool would have a slight edge because of language. However this concern would have been more significant if this tool turned out to be the fastest which is not the case as we see below.

3.4 Threats to External Validity

Threats to external validity are conditions that limit us in generalizing the results of our experiment. Our biggest concerns in this area is that the subject programs might not be representative of complex constraints. To control this threat, we have studied literature regarding the tools and summarized the complex constraints previously studied, we have also studied structures discussed in algorithm books, and have found that the most commonly used complex structures are actually the basis of a large class of data structures. For example, B-trees, AVL trees, Sparse matrices, hash tables are all basically trees or a combination of trees and lists. We considered complex inputs of real programs like compilers (abstract syntax tree), XML parsers (XML Tree), web browser (HTML Tree), File system tree, Java class hierarchies, and expression trees. All of these share constraints with the basic structures we test here. Therefore we believe that our subjects are representative of complex constraints and can be used to evaluate constraint solvers.

3.5 Threats to Construct Validity

Threats to construct validity are situations where measurement instruments do not adequately capture concepts that they are supposed to capture. In this experiment, we measure performance and ease of writing constraints and using results. Measuring performance is always risky on todays multitasking machines. We controlled this threat with repeated measurements and with no sharing of resources. The quantitative analysis about constraint writing is more prone to this threat. We control this threat by providing raw data (how constraints are written, bounds given, results converted) and add our analysis on top of it.

3.6 Analysis Strategy

We summarize all the data first. We then make observations on this data and our observations on the three quantitative criteria of constraint writing, giving bounds, and using results. Finally, we show several comparisons between performance of different techniques in graphical form.

4 Data and Analysis

We provide performance comparison and its analysis followed by quantitative analysis.

4.1 Performance Comparison

Table 1 shows the results of our experiments. The first column lists the complex structures we chose. The next column specifies the size we are using. For Binary Tree, Singly Linked List, and Doubly Linked List, we generate structures up to

Table 1. Results of generating bounded exhaustive test cases for six subject structures by CUTE, Korat, Alloy, and JPF. Time out or tool limitations are represented by a hyphen (-).

Subject	Size	CUTE	Korat	Alloy	JPF
Binary Tree	3	1.761	0.507	0.880	16.349
	4	4.774	0.533	1.085	16.158
	5	15.104	0.567	1.779	16.678
	6	47.427	0.620	5.882	19.405
	7	156.368	0.720	41.866	24.197
	8	527.292	1.048	520.868	48.389
Search Tree	3	2.580	0.579	1.159	16.415
	4	8.240	0.495	1.423	16.478
	5	28.015	0.547	2.529	21.498
	6	95.764	0.746	3.032	43.905
	7	341.444	2.363	6.437	222.893
	8	-	17.515	26.456	1409.366
Red Black Tree	3	43.769	0.841	1.571	15.775
	4	82.905	0.875	1.450	17.139
	5	720.625	0.829	5.293	18.948
	6	-	1.018	4.132	28.186
	7	-	1.687	18.036	57.800
	8	-	5.250	85.277	170.962
Singly Linked List	10	0.855	0.389	8.452	16.661
	13	1.073	0.399	602.250	16.414
	50	4.136	0.481	-	18.015
	100	8.383	0.688	-	23.433
	200	17.273	2.110	-	48.625
	300	27.082	6.138	-	104.517
	400	36.811	13.939	-	200.062
	500	48.849	27.982	-	344.724
Doubly Linked List	10	1.167	0.408	7.408	16.221
	13	1.523	0.411	130.423	15.242
	50	5.657	0.537	-	18.511
	100	11.900	1.047	-	24.547
	200	25.538	4.987	-	63.614
	300	44.332	16.354	-	146.015
	400	67.828	36.503	-	285.589
	500	100.057	72.686	-	501.617
Sorted List	9	1.292	0.395	2.602	21.333
	11	1.557	0.457	7.409	36.900
	13	1.839	1.026	10.420	108.670
	15	2.110	2.286	21.874	439.063
	18	2.821	21.646	-	-
	20	2.797	102.609	-	-
	22	3.036	499.276	-	-

given size while we generate structures of exactly that size for the other three structures. The reason for this is that when generating structures with valid integer ranges of some data variables (e.g. Sorted List), then all tools except CUTE will produce all valid assignments while CUTE will provide a single valid assignment. This makes comparison difficult. We thus chose a fixed size and fixed range of integers such that only one valid assignment exists. The next four columns in the table list the times taken by each tool.

Alloy ran into solver limitations for sizes greater than about 15 nodes for all list structures. Similarly CUTE faced symbolic execution limitations for red black trees. Other numbers not available are time outs for the allocated 15 minutes.

Table 2. Isomorphic Candidates Produced

Subject	CUTE	Korat	Alloy	JPF
Binary Tree	NO	NO	**YES**	NO
Binary Search Tree	NO	NO	NO	NO
Red Black Tree	**YES**	NO	NO	NO
Singly Linked List	NO	NO	**YES**	NO
Doubly Linked List	NO	NO	**YES**	NO
Sorted List	NO	NO	NO	NO

Table 2 shows how well the candidate tools performed in terms of pruning isomorphic candidates. Korat and JPF never produced an isomorphic result. Also from their algorithm, they would never produce a normal isomorphic result according to the definition given previously. Note that their can be domain specific isomorphic results (e.g. isomorphic graphs) which no tool identifies as isomorphic. CUTE produced isomorphic candidates only when it ran into symbolic execution limitations. This happened in our case for red-black trees. Alloy produced isomorphic candidates most often. Its isomorphism pruning is most limited. For example, for a singly linked list, other than the root node and the tail node, it produces more than one isomorphic orderings of the middle nodes.

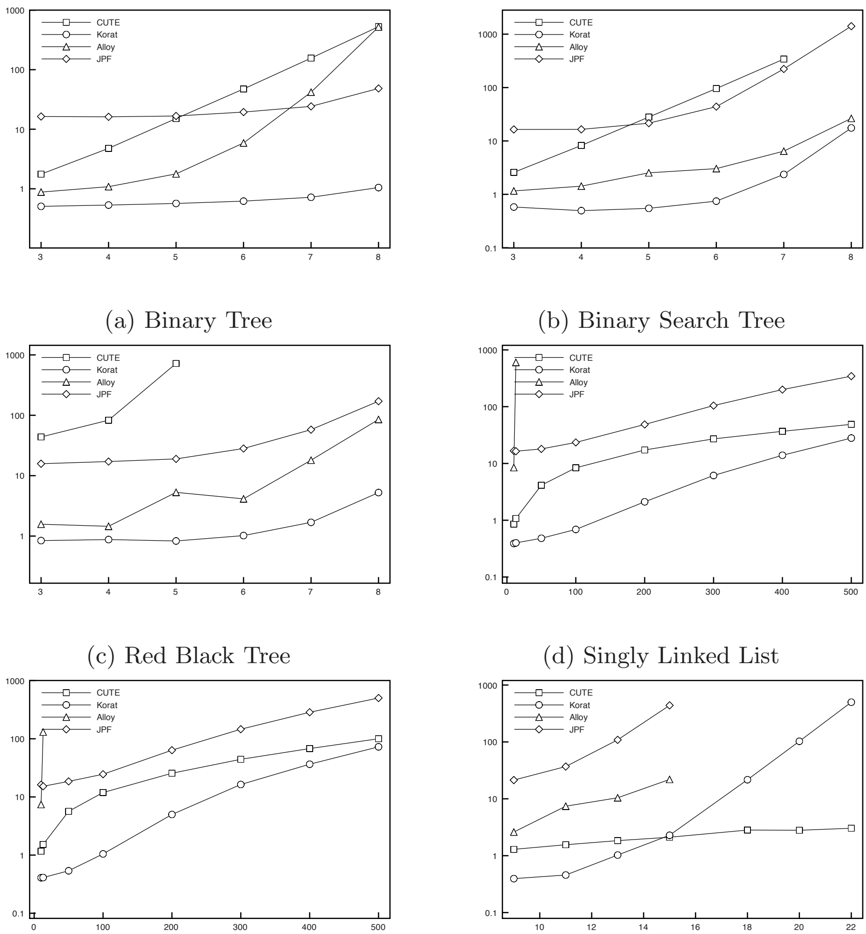
Lastly, Figure 6 shows six graphs, one for each subject structure and plots the performance of all four tools. The time axis is logarithmic since bounded exhaustive testing is an exponentially growing problem and a logarithmic scale better shows how the tools are performing.

We observe that other than sorted lists, Korat is the fastest tool within 1000s time. For binary tree and Red Black Trees, it also seems to grow the slowest. For Binary Trees and Binary Search Trees, CUTE is growing linear on a logarithmic scale which means it is slightly better in terms of time complexity but the actual problem size where it would take over Korat would be huge.

CUTE is the only tool that handles Sorted Lists successfully, It touches our 1000s limit for generating about 500 element lists. This huge difference is because the other tools internally generate all possible combinations (n!) whereas symbolic execution does not. This is also the motivation around some recent work on Korat and JPF to use symbolic execution for primitives and use the native algorithm for non-primitive fields [51].

Note also in all graphs that CUTE has the best time complexity. It grows exponentially (trees) and sub-exponentially (lists) except for red black trees where symbolic execution faced limitations. Thus when symbolic execution faces limitations and CUTE reverts to take help from concrete execution, we may not get results comparable to other tools. This is one of the key weak points of CUTE for bounded exhaustive generation.

Alloy shows an interesting behavior. It performs better for Binary Search Trees (more complex constraint) than Binary Trees. We believe that this is because SAT solvers solve the easiest clauses first and the former gives it a better chance at doing that. Red black tree performance is in the middle and is better for 4



(a) Binary Tree

(b) Binary Search Tree

(c) Red Black Tree

(d) Singly Linked List

(e) Doubly Linked List

(f) Sorted List

Fig. 6. Performance Comparison of techniques for all six subject structures. Y-axis shows time in seconds on a logarithmic scale. X-axis shows size of structure.

nodes than for 3 (and 6 nodes than for 5). We again believe this has to do with the formation of clauses.

If we carefully note, the graph of JPF is almost at a constant distance above Korat. Indeed, JPF structural constraint solving algorithm and the Korat algorithm principally make the same decisions. JPF is only burdened with running a model checking virtual machine and keeping a lot of additional state which Korat can do without. That is why they have similar time complexity but a different multiplier. Thus we can say that Korat is a much faster specialized implementation of what the JPF structural constraint solving algorithm does without the added overheads of model checking.

Table 3. Comparison of structural constraint solving techniques on non-performance metrics

	Constraints	Bounds	Output
CUTE	Imperative function: Some special care at branches to enable symbolic execution to visit both branches	For linear structures, giving a depth bound in invoking CUTE is enough; for others, special checks needed to be inserted inside the predicate	Each complex structure is available at end of testing function in a separate process
Korat	Imperative function: No special restrictions	An imperative function listing bounds for each object and predicate involved (*finitization*)	Each structure is available in a special function in the same single process
Alloy	Declarative predicate: In relational quantified logic	List of bounds for each object involved	Result is a list of solutions that can be translated into actual heap structures using Alloy to Java translator in TestEra [33]
JPF	Imperative function: Need to use special accessor functions (can be added automatically) that use model checker's non-determinism	Ranges can be specified in special accessor functions	Each complex structure is available at end of testing function in a separate process

4.2 Qualitative Comparison

One of the research goals of our experiment was to discuss some qualitative differences between subject tools. We give summarized results in Table 3 and give a more detailed discussion of each difference below.

Constraint Writing: All tools except Alloy required constraints written in an imperative language. Constraints are required to be free of side-effects. CUTE constraints needed some tweaking to allow symbolic execution to explore all paths. For example, a `return size == 0` statement has to be changed to a branch statement with separate returns. JPF and Korat can use an arbitrary imperative function that is free of side-effects. Alloy required declarative predicates. Declarative specifications are concise and can be significantly smaller than an equivalent imperative specification. The tradeoff is the learning curve of declarative language for programmers used to writing code in imperative languages.

Giving Bounds: Korat and Alloy were the easiest to provide bounds, which is not surprising since they are designed for specification-based, bounded exhaustive checking. They differed in that Alloy required bounds for each *type* whereas Korat was more explicit in requiring bounds for each *field* of each type. Also for primitives, Korat can use lower bounds and upper bounds whereas Alloy would need those bounds as part of specification and not as part of bounds. To limit structures generated by CUTE within bounds, we needed to tweak its imperative predicate. Providing bounds using the JPF approach was simple. In this approach the required arrays (universe of values) were constructed during the testing `Main` method. Values of these arrays are non-deterministically used by accessor functions (possibly automatically added).

Using Results: The JPF approach and CUTE approach produce each result, i.e. structure that represents a test input, in a separate execution (process). This result can directly be used for testing or saved for later use. Korat approach produces each result in the same execution (process). The result can be saved. Direct testing has to be careful about using a new process to avoid crashing of Korat due to faulty code. In previous work, these results have been distributed for parallel test execution [39]. Alloy produces solutions to declarative specifications. These need to be converted to the corresponding imperative language for actual test use. One tool in this area is Alloy to Java converter used in TestEra [33]. This tool can generate actual Java structures corresponding to Alloy output.

Treatment of primitive fields: While the key benefit of structural constraint solving is non-primitive fields (pointers to objects), primitive fields also pose a limitation. All the surveyed tools except CUTE try all possible values for a given primitive field. This often results in exponential or factorial amount of time. CUTE excels in this area by providing a single valid solution for such fields.

5 Summary and Conclusions

In this paper, we performed an empirical study of using four different techniques for constraint solving to perform bounded exhaustive testing. Bounded exhaustive testing has been previously shown effective at finding faults in real programs. Here, our goal is to compare the performance of these tools. We considered the CUTE tool based on symbolic execution, the JPF model checker, the Alloy tool based on SAT, and the specialized solver Korat . Our key results are:

– The fastest tool for most of the subjects of small size is Korat. However it degrades in performance when several constraints are on primitive fields.
– The JPF constraint solving approach using lazy initialization is effectively a slower Korat.
– Alloy provides the most concise way of writing predicates. For programmers knowledgeable in declarative languages, it can significantly reduce time to write or maintain specifications.
– CUTE provides better time complexity than most tools however the slope constant is fairly high. This is because of the symbolic execution overhead.
– CUTE requires some tweaking of class invariants to enable bounded exhaustive generation.
– No tool gives better non-isomorphic generation for exhaustive enumeration than the Korat algorithm (and likewise lazy initialization using JPF).
– All tools except CUTE provide bounded exhaustive checking by design and CUTE focuses on generating one input per path.

Our results also provide directions for future work. We see two main directions of research:

– Using symbolic execution to improve the specialized solver Korat.
– While Alloy provides an intuitive way to write specifications (after the learning curve), its solving capability is limited to smaller sizes (see list data structure) and can often produce isomorphic candidates. We believe using a combination of solvers, such as SAT, SMT, string constraint solvers, and set constraint solvers, is likely to provide significantly more efficient solving.
– Similar to parallelization for Korat [39, 46], we are working on parallel symbolic execution. Other tools, such as Alloy, can also gain from parallel execution, both on commodity parallel machines and bigger clusters.

Acknowledgements

We thank Darko Marinov for insightful and detailed discussions and comments. This material is based upon work partially supported by the NSF under Grant Nos. IIS-0438967, CCF-0702680, and CCF-0845628, and AFOSR grant FA9550-09-1-0351.

References

1. Artho, C., Drusinksy, D., Goldberg, A., Havelund, K., Lowry, M., Păsăreanu, C.S., Roşu, G., Visser, W.: Experiments with Test Case Generation and Runtime Analysis. In: Börger, E., Gargantini, A., Riccobene, E. (eds.) ASM 2003. LNCS, vol. 2589, pp. 87–107. Springer, Heidelberg (2003)
2. Ball, T., Rajamani, S.K.: Automatically Validating Temporal Safety Properties of Interfaces. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 103–122. Springer, Heidelberg (2001)
3. Bayer, R.: Symmetric Binary B-trees: Data Structure and Maintenance Algorithms. Acta informatica 1(4), 290–306 (1972)
4. Beyer, D., Chlipala, A.J., Majumdar, R.: Generating Tests from Counterexamples. In: Proc. 2004 Int. Conf. Softw. Eng (ICSE), pp. 326–335 (2004)
5. Boyapati, C., Khurshid, S., Marinov, D.: Korat: Automated Testing based on Java Predicates. In: Proc. 2002 Int. Symp. Softw. Testing and Analysis (ISSTA), pp. 123–133 (2002)
6. Brat, G., et al.: Experimental Evaluation of Verification and Validation Tools on Martian Rover Software. Form. Methods Syst. Des. 25(2-3), 167–198 (2004)
7. Bush, W.R., Pincus, J.D., Sielaff, D.J.: A Static Analyzer for Finding Dynamic Programming Errors. Softw. Pract. Exper. 30(7), 775–802 (2000)
8. Cadar, C., et al.: EXE: Automatically Generating Inputs of Death. In: Proc. 13th Conf. Comput. and Commun. Security (CCS), pp. 322–335 (2006)
9. Chang, J., Richardson, D.: Structural Specification-Based Testing: Automated Support and Experimental Evaluation. In: Nierstrasz, O., Lemoine, M. (eds.) ESEC 1999 and ESEC-FSE 1999. LNCS, vol. 1687, pp. 285–302. Springer, Heidelberg (1999)

10. Clarke, E.M.: The Birth of Model Checking. 25 Years of Model Checking: History, Achievements, Perspectives, 1–26 (2008)
11. Clarke, L.A.: A System to Generate Test Data and Symbolically Execute Programs. IEEE Trans. Softw. Eng. 2(3), 215–222 (1976)
12. Coen-Porisini, A., et al.: Using Symbolic Execution for Verifying Safety-critical Systems. In: Proc. 3rd joint meeting of the Euro. Softw. Eng. Conf. (ESEC) and Symp. Foundations of Softw. Eng. (FSE), pp. 142–151 (2001)
13. Corbett, J., et al.: Bandera: Extracting Finite-state Models from Java Source Code. In: Proc. 2000 Int. Conf. Softw. Eng. (ICSE), pp. 439–448 (2000)
14. Darga, P.T., Boyapati, C.: Efficient Software Model Checking of Data Structure Properties. In: Proc. 21st Annual Conf. Object Oriented Prog. Syst., Lang., and Applications (OOPSLA), pp. 363–382 (2006)
15. Donat, M.R.: Automating Formal Specification-Based Testing. In: Bidoit, M., Dauchet, M. (eds.) CAAP 1997, FASE 1997, and TAPSOFT 1997. LNCS, vol. 1214, pp. 833–847. Springer, Heidelberg (1997)
16. Elkarablieh, B., et al.: STARC: Static Analysis for Efficient Repair of Complex Data. In: Proc. 22nd Annual Conf. Object Oriented Prog. Syst., Lang., and Applications (OOPSLA), pp. 387–404 (2007)
17. Elkarablieh, B., Marinov, D., Khurshid, S.: Efficient Solving of Structural Constraints. In: Proc. 2008 Int. Symp. Softw. Testing and Analysis (ISSTA), pp. 39–50 (2008)
18. Flanagan, C., et al.: Extended Static Checking for Java. In: Proc. 2002 Conf. Prog. Lang. Design and Implementation (PLDI), pp. 234–245 (2002)
19. Gligoric, M., et al.: Optimizing Generation of Object Graphs in Java PathFinder. In: Proc. 2nd Int. Conf. Softw. Testing Verification and Validation (ICST), pp. 51–60 (2009)
20. Godefroid, P.: Model Checking for Programming Languages using VeriSoft. In: Proc. 24th Symp. Principles of Prog. Lang. (POPL), pp. 174–186 (1997)
21. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed Automated Random Testing. In: Proc. 2005 Conf. Prog. Lang. Design and Implementation (PLDI), pp. 213–223 (2005)
22. Goldberg, E., Novikov, Y.: BerkMin: A Fast and Robust SAT-solver. Discrete Appl. Math. 155(12), 1549–1561 (2007)
23. Goodenough, J.B., Gerhart, S.L.: Toward a Theory of Test Data Selection. In: Proc. Int. Conf. Reliable Softw. Technol., pp. 493–510 (1975)
24. Gotlieb, A., Botella, B., Rueher, M.: Automatic Test Data Generation using Constraint Solving Techniques. In: Proc. 1998 Int. Symp. Softw. Testing and Analysis (ISSTA), pp. 53–62 (1998)
25. Grieskamp, W., et al.: Generating Finite State Machines from Abstract State Machines. In: Proc. 2002 Int. Symp. Softw. Testing and Analysis (ISSTA), pp. 112–122 (2002)
26. Guibas, L.J., Sedgewick, R.: A Dichromatic Framework for Balanced Trees. In: Proc. 19th Annual Symp. Foundations of Comput. Sci. (FOCS), pp. 8–21 (1978)
27. Henzinger, T., et al.: Software Verification with BLAST. In: Proc. 10th Int. SPIN Workshop on Model Checking of Softw., pp. 235–239 (2003)
28. Huang, J.C.: An Approach to Program Testing. ACM Comput. Surv. 7(3), 113–128 (1975)
29. Jackson, D.: Alloy: A Lightweight Object Modelling Notation. ACM Trans. Softw. Eng. and Methodology 11(2), 256–290 (2002)
30. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press, Cambridge (2006)

31. Jackson, D., Schechter, I., Shlyakhter, I.: ALCOA: The Alloy Constraint Analyzer. In: Proc. 2000 Int. Conf. Softw. Eng. (ICSE), pp. 730–733 (2000)
32. Khurshid, S., Marinov, D.: Checking Java Implementation of a Naming Architecture Using TestEra. Electr. Notes Theor. Comput. Sci. 55(3) (2001)
33. Khurshid, S., Marinov, D.: TestEra: Specification-Based Testing of Java Programs using SAT. Automated Softw. Eng. J. 11(4), 403–434 (2004)
34. Khurshid, S., Păsăreanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 553–568. Springer, Heidelberg (2003)
35. King, J.C.: Symbolic Execution and Program Testing. Commun. ACM 19(7), 385–394 (1976)
36. Korel, B.: Automated Test Data Generation for Programs with Procedures. In: Proc. 1996 Int. Symp. Softw. Testing and Analysis (ISSTA), pp. 209–215 (1996)
37. Larson, E., Austin, T.: High Coverage Detection of Input-related Security Faults. In: Proc. 12th USENIX Security Symp., p. 9 (2003)
38. Marinov, D., Khurshid, S.: TestEra: A Novel Framework for Automated Testing of Java Programs. In: Proc. 16th Int. Conf. Automated Softw. Eng. (ASE), p. 22 (2001)
39. Misailovic, S., et al.: Parallel Test Generation and Execution with Korat. In: Proc. 6th joint meeting of the Euro. Softw. Eng. Conf. (ESEC) and Symp. Foundations of Softw. Eng. (FSE), pp. 135–144 (2007)
40. Moskewicz, M.W., et al.: Chaff: Engineering an Efficient SAT Solver. In: Proc. 38th Design Automation Conf. (DAC), pp. 530–535 (2001)
41. Offutt, J., Abdurazik, A.: Generating Tests from UML Specifications. In: Proc. 2nd Int. Conf. Unified Modeling Language, pp. 416–429 (1999)
42. Penix, J., et al.: Verification of Time Partitioning in the DEOS Scheduler Kernel. In: Proc. 2000 Int. Conf. Softw. Eng. (ICSE), pp. 488–497 (2000)
43. Ramamoorthy, C.V., Ho, S.B.F., Chen, W.T.: On the Automated Generation of Program Test Data. IEEE Trans. Softw. Eng. 2(4), 293–300 (1976)
44. Sen, K., Marinov, D., Agha, G.: CUTE: A Concolic Unit Testing Engine for C. In: Proc. 5th joint meeting of the Euro. Softw. Eng. Conf. (ESEC) and Symp. Foundations of Softw. Eng. (FSE), pp. 263–272 (2005)
45. Shlyakhter, I.: Generating Effective Symmetry-breaking Predicates for Search Problems. Discrete Appl. Math. 155(12), 1539–1548 (2007)
46. Siddiqui, J.H., Khurshid, S.: PKorat: Parallel Generation of Structurally Complex Test Inputs. In: Proc. 2nd Int. Conf. Softw. Testing Verification and Validation (ICST), pp. 250–259 (2009)
47. Sorensson, N., Een, N.: An Extensible SAT-solver. In: Proc. 6th Int. Conf. Theor. and Applications of Satisfiability Testing (SAT), pp. 502–518 (2003)
48. Spivey, J.M.: The Z Notation: A Reference Manual. Prentice-Hall, Inc., Englewood Cliffs (1989)
49. Sullivan, K., et al.: Software Assurance by Bounded Exhaustive Testing. In: Proc. 2004 Int. Symp. Softw. Testing and Analysis (ISSTA), pp. 133–142 (2004)
50. Visser, W., et al.: Model Checking Programs. In: Proc. 15th Int. Conf. Automated Softw. Eng. (ASE), p. 3 (2000)
51. Visser, W., Păsăreanu, C.S., Khurshid, S.: Test Input Generation with Java PathFinder. In: Proc. 2004 Int. Symp. Softw. Testing and Analysis (ISSTA), pp. 97–107 (2004)
52. Xie, T., Marinov, D., Notkin, D.: Rostra: A Framework for Detecting Redundant Object-Oriented Unit Tests. In: Proc. 29th Int. Conf. Automated Softw. Eng. (ASE), pp. 196–205 (2004)