ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), October 2006

Efficient Software Model Checking of Data Structure Properties

Paul T. Darga   Chandrasekhar Boyapati

Electrical Engineering and Computer Science Department
University of Michigan, Ann Arbor, MI 48109
{pdarga,bchandra}@eecs.umich.edu

Abstract

This paper presents novel language and analysis techniques that significantly speed up software model checking of data structure properties. Consider checking a red-black tree implementation. Traditional software model checkers systematically generate all red-black tree states (within some given bounds) and check every red-black tree operation (such as insert, delete, or lookup) on every red-black tree state. Our key idea is as follows. As our checker checks a red-black tree operation o on a red-black tree state s, it uses program analysis techniques to identify other red-black tree states s'_1, s'_2, ..., s'_k on which the operation o behaves similarly. Our analyses guarantee that if o executes correctly on s, then o will execute correctly on every s'_i. Our checker therefore does not need to check o on any s'_i once it checks o on s. It thus safely prunes those state transitions from its search space, while still achieving complete test coverage within the bounded domain. Our preliminary results show *orders of magnitude improvement* over previous approaches. We believe our techniques can make model checking significantly faster, and thus enable checking of much larger programs and complex program properties than currently possible.

Categories and Subject Descriptors
D.2.4 [*Software Engineering*]: Program Verification
D.2.5 [*Software Engineering*]: Testing and Debugging
F.3.1 [*Logics*]: Specifying and Verifying Programs

General Terms
Verification, Reliability, Languages

Keywords
Software Model Checking, Program Analysis

1. Introduction

This paper presents novel language and analysis techniques that significantly speed up software model checking [1, 2, 4, 6, 7, 12, 15, 48, 21, 39] of data structure properties. Model checking is a formal verification technique that exhaustively tests a circuit/program on all possible inputs (sometimes up to a given size) to handle input nondeterminism and on all possible nondeterministic schedules to handle scheduling nondeterminism. For hardware, model checkers have been successfully used to verify fairly complex finite state control circuits with up to a few hundred bits of state information; but not circuits in general that have large data paths or memories. Similarly, for software, model checkers have been primarily used to verify event sequences with respect to temporal properties; but not much work has been done to verify programs that manipulate rich complex data with respect to properties that depend on the data. This paper deals with such data oriented programs. In particular, it focuses on verifying properties of data structures.

Consider checking that a red-black tree [8] implementation maintains the red-black tree invariants. Previous model checking approaches such as JPF [48, 28], CMC [39, 38], Korat [2], or Alloy [26, 27] systematically generate all red-black trees (up to a given size n) and check every red-black tree operation (such as insert or delete) on every red-black tree. Since the number of red-black trees with at most n nodes is exponential in n, these systems take time exponential in n for checking a red-black tree implementation. Our key idea is as follows. Our checker detects that any red-black tree operation such as insert or delete touches only one path in the tree from the root to a leaf (and perhaps some nearby nodes). Our checker then determines that it is sufficient to check every operation on every unique tree path (and some nearby nodes), rather than on every unique tree. Since the number of unique red-black tree paths is polynomial in n, our checker takes time polynomial in n. This leads to orders of magnitude speedups over previous approaches.

In general, our system works as follows. Consider checking a file system implementation, as another example. As our checker checks a file system operation o (such as reading, writing, creating, or deleting a file or a directory) on a file system state s, it uses program analyses to identify other file system states s'_1, s'_2, ..., s'_k on which the operation o behaves similarly. Our analyses guarantee that if o executes correctly on s, then o will also execute correctly on every s'_i. Our checker therefore does not need to check o on any s'_i once it checks o on s. It thus safely prunes all those state transitions from its search space, while still achieving complete test coverage within the bounded domain.

We call this the *glass box* approach to model checking of data oriented programs because our checker analyzes the behavior

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
OOPSLA'06 October 22–26, 2006, Portland, Oregon, USA.
Copyright © 2006 ACM 1-59593-348-4/06/0010 ...$5.00

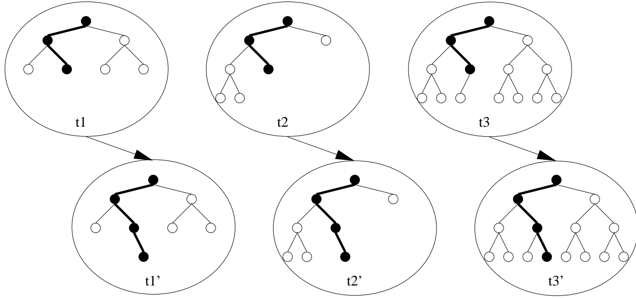

Figure 1. Three red-black trees before and after an `insert` operation. The tree path touched by the operation is highlighted in each case. Once our glass box checker checks the `insert` operation on tree `t1`, it determines that it is redundant to check the same operation on `t2` and `t3`.

```
1 class Queue {
2   private Stack front = new Stack();
3   private Stack back  = new Stack();
4
5   public boolean repOk() {
6     return (back != null) && back.repOk() && (back != front)
7         && (front != null) && front.repOk();
8   }
9
10  //              ------------     ------------
11  // dequeue <---    front    |   |    back     <--- enqueue
12  //              ------------     ------------
13
14  public void enqueue(Object o) {
15    back.push(o);
16  }
17  public Object dequeue() throws EmptyQueueException {
18   if (front.isEmpty()) moveBackToFront();
19    if (front.isEmpty()) throw new EmptyQueueException();
20    return front.pop();
21  }
22  private void moveBackToFront() {
23    while (!back.isEmpty()) front.push(back.pop());
24  }
25 }
```

Figure 2. `Queue` implemented using two `Stacks`.

of an operation to prune large portions of the search space. While there is much research on state space reduction techniques for model checkers such as partial order reduction [13, 15, 16] and tools based on predicate abstraction [19] such as Slam [1], Blast [21], or Magic [4], none of these techniques seem to be effective in reducing the state space of data oriented programs. For example, predicate abstraction relies on alias analysis that is often too imprecise. Thus, our approach is in contrast to the traditional *black box* approach to model checking of data oriented programs that checks every operation on every state, treating the operation as a black box. Depending on the strength of the analyses, a glass box checker can be significantly more efficient than a black box checker in exploring the same search space. Indeed, our preliminary results show *orders of magnitude improvement* over previous approaches. We therefore believe that our techniques can make model checking significantly faster, and thus enable checking of much larger programs and broader class of program properties than currently possible.

The rest of this paper is organized as follows. Section 2 illustrates our approach with examples. Section 3 describes our glass box model checker. Section 4 presents experimental results. Section 5 discusses related work. Section 6 concludes.

2. Examples

This section illustrates our key idea with examples.

2.1 Red-Black Tree Example

Consider the red-black tree example from Section 1. That is, consider checking that a red-black tree implementation maintains the red-black tree invariants. As we discussed in Section 1, a black box checker (such as JPF [48, 28], CMC [39, 38], Korat [2], or Alloy [26, 27]) systematically generates all red-black trees (up to a given size n) and checks every red-black tree operation (such as insert or delete) on every red-black tree. Since the number of red-black trees with at most n nodes is exponential in n, a black box checker takes time exponential in n for the checking.

Our glass box checker works as follows. Consider checking the `insert` operation on tree `t1` in Figure 1. The tree `t1'` depicts the state of the tree after the operation. (For simplicity, the figure only shows the tree structures and does not show the color of the nodes, or the keys or values stored in the nodes.) As our checker checks the `insert` operation on `t1`, it detects that the operation touches only one path in the tree from the root to a leaf. This path is highlighted in the figure. That means, assuming deterministic execution, the `insert` operation will behave similarly on all trees, such as `t2` or `t3`, where the highlighted path remains the same. Our checker determines that it is redundant to check the same `insert` operation on trees such as `t2` or `t3` once it checks the `insert` operation on tree `t1`. Our checker safely prunes those state transitions from its search space, while still achieving complete test coverage within the bounded domain. Our checker thus ends up checking every red-black tree operation on every unique tree path (and some nearby nodes), rather than on every unique tree. Since the number of unique red-black tree paths (in trees with at most n nodes) is polynomial in n, our checker takes time polynomial in n to check a red-black tree implementation. This leads to orders of magnitude speedups over the black box approach.

2.2 Queue Example

This section illustrates our approach with a more detailed example. Figure 2 presents a `Queue` that is implemented using two `Stack` objects `front` and `back`. The `enqueue` method inserts an item at the back of a `Queue` by pushing it onto `back`. The `dequeue` method removes and returns the item at the front of a `Queue` by popping and returning the top item of `front`. If `front` is empty, `dequeue` first moves all the items from `back` to `front`. If `front` is still empty, `dequeue` throws an `EmptyQueueException`. (One possible implementation of `Stack` is shown in Figure 7.)

`Queue`'s class invariant is described by its `repOk` method, as good programming practice suggests [32]. The class invariant of an object must hold before and after every public method of the object. That is, the class invariant is both a precondition and a postcondition of every public method.

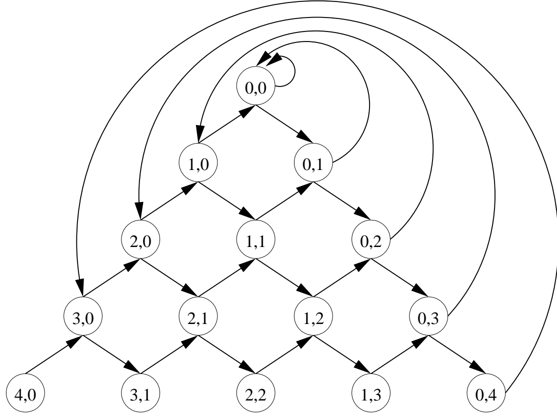

Figure 3. State space of `Queue` with at most $n = 4$ items. State (f,b) has f items in `front Stack` and b in `back`. A black box checker checks $\Omega(n^2)$ state transitions.

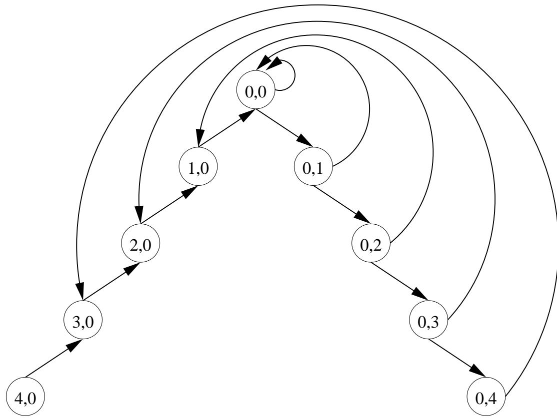

Figure 4. A glass box checker generates only $O(n)$ states and checks only $O(n)$ transitions, yet achieves complete coverage within the bounded domain.

The `repOk` method returns true iff the current state (or representation) of an object satisfies its class invariant. The class invariant of `Queue` holds iff its subobjects `front` and `back` are different and not null, and their invariants hold.

Consider checking that every public method of `Queue` preserves its class invariant. That is, consider checking that if the class invariant holds before a method, then the class invariant holds after the method, and the method either returns normally or by throwing one of its declared exceptions. We assume all `Queue` methods execute deterministically. (Otherwise, one must expose the nondeterminism points to the model checker to check every possibility.)

A black box checker such as JPF [48] or CMC [39, 38] starts from an empty `Queue` state and recursively invokes and checks every `Queue` operation on every successive `Queue` state (within a bounded domain). A stateful checker stores all the checked states in a hashtable to avoid redundantly checking the same operation on the same state more than once. Suppose there is exactly one concrete state represent-

```
1  class ReachabilityDemo {
2    private boolean x, y, z;
3    public boolean repOk() { return !z || x && y; }
4
5    public void setX() { x = true; }
6    public void setY() { y = true; }
7    public void setZ() { if (x && y) z = true; }
8  }
```

Figure 5. A class with three boolean variables `x`, `y`, `z`.

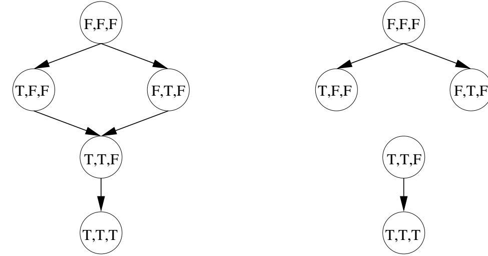

Figure 6. State space of code in Figure 5 (excluding self loops). (b_1,b_2,b_3) implies $\texttt{x} = b_1$, $\texttt{y} = b_2$, $\texttt{z} = b_3$. The figures on the left and right show the state transitions executed by a black box and glass box checker respectively.

ing a `Stack` of size n. Then there are $n + 1$ concrete states representing a `Queue` of size n. Figure 3 shows the state space of `Queue` with at most $n = 4$ items. State (f,b) has f items in `front` and b in `back`. Edges represent `enqueue` and `dequeue` operations. E.g., the edge from (1,1) to (1,2) represents an `enqueue`. The edges from (1,2) to (0,2) and (0,2) to (1,0) represent `dequeue` operations. A black box checker executes $\Omega(n^2)$ state transitions to explore this space.

Our glass box checker works as follows. Consider the transition from (0,0) to (0,1) using the `enqueue` method. This operation terminates normally and the class invariant holds after the method. As our checker checks this operation, its dynamic analysis detects that the `enqueue` method does not read the `front Stack`. That means, if the state of the `front Stack` were different, the `enqueue` method would still execute similarly. Our checker then determines that if `enqueue` executes successfully on (0,0), then it will execute successfully on (i,0) for any i. Our checker therefore safely prunes all those state transitions from its search space. In particular, if `Queue` has at most $n = 4$ items, our checker prunes the `enqueue` edges from (0,0), (1,0), (2,0), (3,0), and (4,0) once it successfully checks `enqueue` on (0,0).

Similarly, checking `enqueue` on (0,1), (0,2) and (0,3) results in pruning `enqueue` operations on all (i,1), (i,2) and (i,3). Checking `dequeue` on (1,0), (2,0), and (3,0) results in pruning `dequeue` operations on all (1,i), (2,i), and (3,i). Figure 4 presents the same state space as Figure 3 except that it only shows the transitions that our checker executes. Our glass box checker executes only $O(n)$ state transitions to explore the state space, while still achieving complete test coverage within the bounded domain. Moreover, our checker never generates states from which all transitions have been pruned. For example, our checker never generates any state (i,j) where $i \neq 0$ and $j \neq 0$. Thus, our checker generates only

```
1  public class Stack {
2    private static class Node {
3      Node   next;
4      Object value;
5      Node(Node n, Object v) { next = n; value = v; }
6    }
7
8    private Node head;
9
10   public boolean repOk() {
11     Set  visited = new java.util.HashSet();
12     for (Node n = head; n != null; n = n.next) {
13       if (!visited.add(n)) return false;
14     }
15     return true;
16   }
17
18   public void push(Object value) {
19     head = new Node(head,value);
20   }
21   public Object pop() {
22     if (head == null) return null;
23     Object v = head.value;  head = head.next; return v;
24   }
25 }
```

Figure 7. Stack implemented using a linked list.

```
1  public class Stack {
2    private static class Node {
3      tree Node next;
4      Object    value;
5      Node(Node n, Object v) { next = n; value = v; }
6    }
7
8    private tree Node head;
9
10   public boolean repOk() { return true; }
11
12   public void push(Object value) {
13     head = new Node(head,value);
14   }
15   public Object pop() {
16     if (head == null) return null;
17     Object v = head.value;  head = head.next; return v;
18   }
19 }
```

Figure 8. Stack in Figure 7 with its invariant rewritten using the tree annotation (Line 3). The repOk (Line 10) then has no additional constraints to specify.

$O(n)$ states and checks only $O(n)$ transitions, compared to $O(n^2)$ states and $O(n^2)$ transitions in a black box approach. This results in significant speedups.

Note that for simplicity, we implicitly assumed that there is only one possible argument to `enqueue`, so there is only one `enqueue` transition from each state. But suppose there are n different items that can be passed as arguments to `enqueue`, so there are n `enqueue` transitions from each state. Then, for checking a `Queue` of size n, a black box checker actually executes an exponential number of transitions. Our glass box checker still executes $O(n)$ transitions.

3. Glass Box Model Checking

This section presents our glass box model checker. While the basic idea illustrated in the previous section is simple, one has to overcome several technical challenges to make it work well in practice. This section describes our approach. Section 3.1 describes the search space of a glass box model checker, Section 3.2 describes the search space representation, and Section 3.3 describes the search process.

3.1 Search Space

This section describes the search space of a glass box checker.

3.1.1 Search Space Organization

Consider the `Stack` example from Figure 7. The `Stack` is implemented using a linked list. Its class invariant (`repOk`) checks that the list is acyclic. Consider checking that the `Stack` implementation preserves the `Stack` invariant.

One way to systematically test the `Stack` implementation is to start from the initial empty `Stack` state, and recursively invoke and check every `Stack` operation on every successive `Stack` state (within a bounded domain). Some black box checkers such as JPF [48] or CMC [39] use this approach. The *stateful* black box checkers store (a hash of) every checked state in a hashtable to avoid redundantly checking the same operation on the same state more than once.

The above technique, however, is not a suitable way for a glass box checker to organize its search space. The example in Figure 5 illustrates why. Figure 6 shows the corresponding state space (excluding self loops). A black box checker using the above technique starts from the initial state and reaches all five states by recursively invoking methods on successive states. However, as a glass box checker checks the `setX` method on state (F,F,F), its analyses detect that `setX` behaves similarly on state (F,T,F). Therefore, the glass box checker prunes that edge from its state space. Similarly, as a glass box checker checks `setY` on (F,F,F), it prunes `setY` from (T,F,F). But this disconnects the state space graph. A glass box checker thus cannot depend on reachability of the state space to reach the state (T,T,F).

Instead, our glass box checker uses a different approach. Our system requires programmers to specify the class invariants of data structures. For example, in Figure 7, the `repOk` method describes the class invariant of the `Stack`. The search space of a glass box checker checking a data structure is defined to consist of all type-correct states (within some finite bounds) that satisfy its class invariant. Note that this is different from the search space of a black box checker, which is defined to consist of all states (within some finite bounds) that are reachable from the initial state by performing a sequence of data structure operations. Sections 3.1.2 and 3.1.3 discuss the implications of this difference.

A glass box checker exhaustively checks every operation on every state within the search space, but does so efficiently by detecting redundancies in the search space and pruning away large portions of the search space without explicitly checking them. Each time our checker checks an operation, it verifies that (i) the operation either terminates normally or throws one of its declared exception, (ii) the invariant holds after the operation, and (iii) any additional properties specified by programmers (e.g., as assert statements) hold.

```
1  public class RedBlackTree {
2    private static class Node {
3      tree Node left;
4      tree Node right;
5      Node      parent
6      boolean   color;
7
8      public boolean repOkLocal() {
9        assert((left  == null) || (left.parent  == this));
10       assert((right == null) || (right.parent == this));
11
12       if (color == RED) {
13         assert((left  == null) || (left.color  == BLACK));
14         assert((right == null) || (right.color == BLACK));
15       }
16
17       return true;
18     }
19   }
20
21   private tree Node root;
22
23   public boolean repOk() {
24     // Return true iff the number of black nodes in every
25     // path from the root to a leaf is the same.
26   }
27   ...
28 }
```

Figure 9. Partial implementation of RedBlackTree, excluding keys and values from Nodes.

```
1  public class RedBlackTree {
2    private static class Node {
3      tree Node left;
4      tree Node right;
5      Node      parent
6      boolean   color;
7      ghost int blackHeight;
8
9      public boolean repOkLocal() {
10       assert((left  == null) || (left.parent  == this));
11       assert((right == null) || (right.parent == this));
12
13       if (color == RED) {
14         assert((left  == null) || (left.color  == BLACK));
15         assert((right == null) || (right.color == BLACK));
16       }
17
18       int x = blackHeight - ((color == BLACK) ? 1 : 0);
19       assert(blackHeight >= 0);
20       assert((left  == null) || (left.blackHeight  == x));
21       assert((right == null) || (right.blackHeight == x));
22       if (x > 0) assert((left != null) && (right != null));
23
24       return true;
25     }
26   }
27
28   private tree Node root;
29
30   public boolean repOk() { return true; }
31   ...
32 }
```

Figure 10. RedBlackTree in Figure 9 with its invariant rewritten using a ghost field (Line 7) and thus converting global constraints (Lines 24-25 in Figure 9) into local constraints (Lines 18-22 in this figure).

3.1.2 Programming Overhead

One of the main advantages of the black box model checking approach is that it requires minimal programmer assistance. For checking application independent properties (such as null pointer dereferences or memory leaks), it requires no programmer assistance. For checking application dependent properties, it only requires a specification of the properties to be checked in an executable form (e.g., using asserts).

Compared to black box checking, glass box checking sometimes involves extra programming effort because programmers have to additionally specify class invariants of data structures as described above. However, because glass box checking is orders of magnitude faster than black box checking, writing the class invariants is often worth the effort. Note that the effort required to write class invariants is proportional to the size and complexity of the data declarations, *not* the size of the code. For example, `java.util.TreeMap` (a red black tree implementation) has 1670 lines of code, whereas its invariant takes less than 1% as many lines. Also, if the goal includes checking that a data structure implementation preserves its invariants, then programmers have to specify the invariants for both black box and glass box checking, in which case there is no additional overhead involved in glass box checking. In addition, if programmers make a mistake in writing the invariants, our system provides concrete counterexamples to help them correct the invariants, as we describe in the next section.

Glass box model checking thus involves slightly more programming overhead compared to black box model checking, but significantly less overhead compared to other formal verification techniques using theorem provers (that require extensive programmer assistance—either as invariants for loops and recursive functions, or as guidance to interactive theorem provers). On the other hand, glass box checking is significantly faster than black box checking but could be slower than techniques using theorem provers, if the theorem provers are provided sufficient manual assistance. Glass box model checking thus presents an interesting trade-off in the design space of software verification techniques.

3.1.3 Handling Errors in Invariants

Programmers can make two kinds of errors in writing the class invariant of a data structure. Let S_R be the set of states (within some finite bounds) that are reachable from the initial state by performing a sequence of operations. Let S_I be the set of states (within the same finite bounds) that satisfy the invariant. We say an invariant is *unsound* if there is a state in S_R that is not in S_I, and *incomplete* if there is a state in S_I that is not in S_R.

If an invariant is unsound, then (assuming the initial state is in S_I) there must exist a transition from states s_1 to s_2, where both s_1 and s_2 are in S_R, but only s_1 is in S_I and s_2 is not. Our glass box checker will eventually check such a transition (or a transition similar to it) and detect that the transition does not preserve the invariant. It will then present the transition as a concrete counterexample to the user. The user can either fix the invariant, or alternately, if the bug is in the code, the user can fix the code.

If an invariant is incomplete, then either (i) the checker detects a false positive, that is, a state s that is in S_I but not S_R on which some operation fails to check—in which

```
1  public Finitization checkStack(int h, int nObjects) {
2      Finitization f = new Finitization("Stack");
3      f.setOperations("push", "pop");
4      f.setMaxTreeHeight(h);
5
6      Set objects = f.createObjects("Object", nObjects);
7      objects.add(null);
8      f.setFieldDomain("Node.value", objects);
9      f.setArgumentDomain("push", "value", objects);
10
11     return f;
12 }
```

Figure 11. Finitization description for code in Figure 8.

Field	Domain
operation	{push, pop}
head	{N0, null}
N0.next	{N1, null}
N1.next	{N2, null}
N2.next	{null}
N0.value	{O0, O1, O2, null}
N1.value	{O0, O1, O2, null}
N2.value	{O0, O1, O2, null}
push.value	{O0, O1, O2, null}

Figure 12. Search space for `checkStack(3,3)`.

case the user can strengthen the invariant by examining the concrete counterexample s, or (ii) the checker successfully checks the program—in which case the checker would have verified the program not only on all reachable states but on some unreachable states as well.

Thus, even though our glass box checker depends on invariants to cover all states, it is sound in that it does not miss any errors in the program that a black box checker would detect, even if programmers make a mistake in specifying the invariants.

3.1.4 Specifying Invariants

One way programmers can specify a class invariant is by writing a `repOk` method [32]. The `repOk` method returns true iff the current state (or representation) of an object satisfies its class invariant. E.g., the `repOk` method of `Stack` in Figure 7 checks that there are no cycles in the linked list.

Our system also allows programmers to specify invariants (as well as other properties to be checked) using a declarative language, such as a subset of JML [30], as long as the declarative specifications can be automatically translated into executable code. For example, a large subset of JML can be automatically translated to Java using the JML tool set [30].

In addition to the above, our system provides a stylized way for specifying certain kinds of invariants that makes it both convenient for programmers to write the invariants, and faster for a glass box checker to check programs using the invariants. Our approach is premised on the observation that most data structures are tree-based. The next three subsections describe this approach.

3.1.4.1 Specifying a Tree Backbone

Given a tree-based data structure, our system allows programmers to specify the tree backbone of the data structure using the keyword `tree` as a field modifier [36]. For example, in Figure 8, the keyword `tree` on Line 3 specifies that the linked list has no cycles along the `next` fields. Note how this is far more convenient to write than the executable specification in Lines 11-15 of Figure 7. In general, if object x has a `tree` field fd that contains a pointer to object y, we say that there is a `tree` edge fd from x to y. x is the tree-parent of y and y is a tree-child of x. The meaning of the `tree` specification is that (before and after every public method) the graph induced by the set of all `tree` edges in the heap is a forest of trees (that is, it has no directed or undirected cycles). Programmers can use the `tree` keyword to specify the tree backbone of any tree-based data structure. This includes singly linked lists, doubly linked lists, trees with parent pointers, threaded trees, balanced search trees, etc. Note that these data structures can have other non-tree pointers that can contain cycles, as long as their tree pointers do not contain cycles. The partial implementation of a `RedBlackTree` in Figure 9 provides another example. The `tree` keyword on Lines 3-4 specify that the `left` and `right` fields form the tree backbone of the data structure.

```
1  public Finitization checkRedBlackTree(int h) {
2      Finitization f = new Finitization("RedBlackTree");
3      f.setOperations(...);
4      f.setMaxTreeHeight(h);
5      return f;
6  }
```

Figure 13. Finitization description for code in Figure 10.

Field	Domain
operation	{...}
root	{N0, null}
N0.left	{N1, null}
N0.right	{N2, null}
N1.left	{N3, null}
N1.right	{N4, null}
N2.left	{N5, null}
N2.right	{N6, null}
N3.left, N3.right, N4.left, N4.right, N5.left, N5.right, N6.left, N6.right	{null}
N0.color, N1.color, N2.color, N3.color, N4.color, N5.color, N6.color	{RED, BLACK}
N0.parent, N1.parent, N2.parent, N3.parent, N4.parent, N5.parent, N6.parent	{N0, N1, N2, N3, N4, N5, N6, null}

Figure 14. Search space for `checkRedBlackTree(3)`.

3.1.4.2 Specifying Local Invariants

Consider the `RedBlackTree` in Figure 9. One of its invariants is that for every node N, (N.`left`≠`null`) ⟹ (N.`left`-.`parent`=N). We say that such invariants, that involve only (the fields of) an object and (the fields of) its tree-children, are local invariants. Another example of a local invariant in `RedBlackTree` is (N.`color`=`RED` ∧ N.`left`≠`null`) ⟹ (N.`left.color`≠`RED`) for all nodes N.

Our system allows programmers to specify local invariants using the `repOkLocal` method. Lines 8-19 in Figure 9 provide an example. To check the local invariant on a particular instance of a data structure, our system traverses the tree

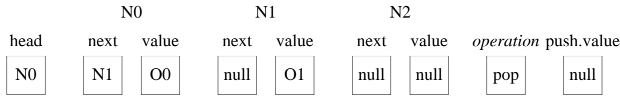

Figure 15. A valid element of the search space in Figure 12 representing the pop operation on a Stack with two items O1 an O2.

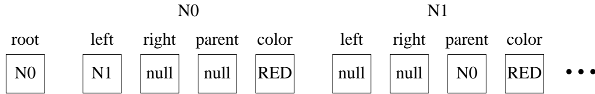

Figure 16. (A portion of) an invalid element of the search space in Figure 14, with the root and its left child both being red.

backbone of the data structure and checks that `repOkLocal` returns `true` on every tree node. The advantage of specifying local invariants this way (as opposed to specifying them as global invariants using the `repOk` method) is that programmers do not have to write code to perform the tree traversal. Another advantage is that it makes glass box model checking faster, as we explain later in the paper.

3.1.4.3 Specifying Non-Local Invariants

In addition to specifying local invariants using `repOkLocal`, programmers can specify other non-local invariants using the `repOk` method. The `repOk` method in Figure 9 provides an example. It checks that the number of black nodes in every path from the root to a leaf is the same.

It is often possible to convert non-local invariants into local ones by adding extra fields [35]. For example, the above non-local invariant can be converted into a local invariant by adding a field `blackHeight` to every node (which stores the number of black nodes in any path from that node to a leaf). This is illustrated in Lines 18-22 of Figure 10. Note that in Line 7 of the figure, `blackHeight` is declared to be a `ghost` field [14]. A `ghost` field exists only during model checking, but otherwise does not exist when the data structure is used in a program. A `ghost` field is thus part of the specification (and not implementation) of a data structure and it does not slow down the performance of the data structure.

3.1.5 Specifying Bounds on Search Space

In any model checker that checks data structure properties, programmers must specify finite bounds on the search space. In our glass box checker, programmers must specify the following: (i) for the tree back-bone (of a tree-based data structure), the maximum height of the tree backbone; (ii) for objects not on the tree backbone, the maximum number of objects of each class; (iii) the domain of every method argument and non-tree field. Our checker then checks the program on every possible state in this finite space.

Figure 11 presents an example finitization description that is *automatically* generated by our system from the type declarations in Figure 7. The `setOperations` method specifies that the checker must check the two public methods `push` and `pop`. The `setMaxTreeHeight` sets the maximum height of the tree backbone. The `createObjects` method specifies that a state can contain at most `nObjects` number of `Objects`. The `setFieldDomain` and `setArgumentDomain` methods specify that the field `value` and the argument to `push` can either contain `null` or an `Object`.

Once our system generates a finitization, programmers can specialize it; e.g., they can make `checkStack` take a single argument `n` and set both `h` and `nObjects` to `n`. We provide several helper functions for easy domain construction.

Figure 13 presents another example finitization description for the code in Figure 9. If the domain of a non-tree field of type T is not explicitly set by the finitization, then our system sets the domain to be the set of all values of type T. For example, for Figure 13, our system sets the domain of `color` to `true` and `false` (representing `BLACK` and `RED`).

3.1.6 Search Space

Suppose our checker is invoked using `checkStack(3,3)` in Figure 11. Our system then constructs the search space in Figure 12. Our system first allocates the specified number of objects: one `Stack`, three `Nodes`, and three `Objects`. It then sets the domain of each object field and method argument as described in the finitization. Finally, it includes the two public methods of `Stack` in the operations to be checked.

The search space consists of all possible assignments to the above fields, where each field gets a value from its corresponding domain. Every element of this search space is a state transition consisting of a concrete `Stack` state, a method to invoke on the state, and the method arguments. For example, Figure 15 corresponds to invoking `pop` on a `Stack` with two items `O0` and `O1`. In Figure 12, there are four fields with four elements in their domains and four with two, so the size of this search space in $4^4 * 2^4$. In general, when our checker is invoked with `checkStack(n,n)`, the size of the search space is $(2n+2)^{n+1}$. Note that some elements of a search space may be invalid because the corresponding structure does not satisfy the class invariant. For example, the element in Figure 16 (for the search space in Figure 14) is invalid because the root and its left child are both red.

3.1.7 Search Algorithm

Figure 17 presents the glass box search algorithm. Given a class to check and a finitization, our system first initializes the search space `S` to the set `I` of all elements that satisfy the invariant of the class. It then systematically explores the space `S` by repeatedly selecting a transition `t` from `S`, check-

```
1  void search(Finitization f) {
2    F = Set of all elements in f
3    I = Set of all elements in F that satisfy the invariant
4    S = I
5    while (S is not empty) {
6      t = Any transition in S
7      Check t
8      T = Set of all transitions similar to t (including t)
9      S  = S - T
10   }
11 }
```

Figure 17. Pseudo-code for the search algorithm.

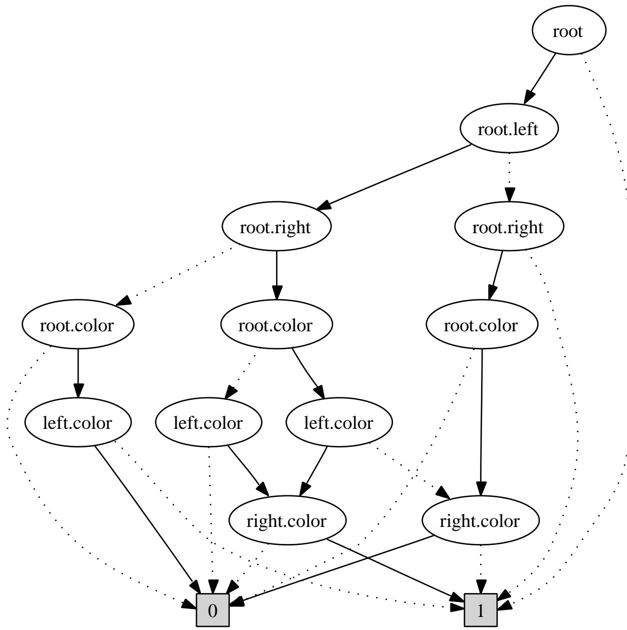

Figure 18. BDD representing the set of all `RedBlackTrees` of maximum height 2 that satisfy the invariant in Figure 10.

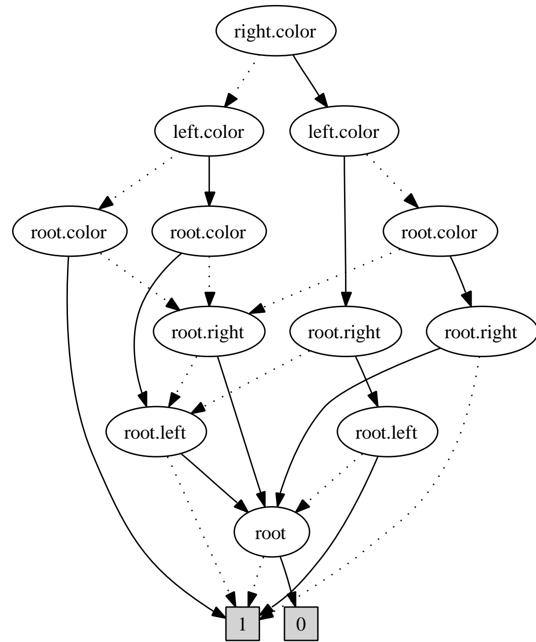

Figure 19. BDD representing the same set as in Figure 18, with the order of the fields in the BDD reversed.

ing `t`, running its analyses to identify the set `T` of transitions similar to `t` (including `t`), and pruning `T` from `S`. Sections 3.2 and 3.3 describe how to perform the above search efficiently.

3.2 Search Space Representation

Consider checking the `RedBlackTree` (Figure 10) with `checkRedBlackTree(`h`)`. Say, $n = 2^h$. Our checker generates $O(n^2)$ states and checks $O(n^2)$ transitions to cover this search space (as we show in Section 4). But the size of the search space is exponential in n. Also, the size of the set `I` in Figure 17 is exponential in n. If we are not careful, then search space management itself could take exponential time, thus defeating most of the advantage gained by glass box checking. To avoid this, we compactly represent the search space using *reduced ordered binary decision diagrams* [3], or BDDs.

Figure 18 presents an example, where the BDD represents the set `I` (in Figure 17) of all `RedBlackTrees` of maximum height 2 that satisfy the invariant in Figure 10. Each node in the BDD represents one bit. A solid line from the node represents the bit being 1 and a dotted line 0. For the fields `root`, `left`, and `right`, 1 represents that the field is non-null and 0 `null`; for `color`, 1 represents `BLACK` and 0 `RED`. (We use more than one bit for fields whose domain contains more than two elements.) Any path in the BDD from the initial node to the node 1 represents (one or more) elements of the set `I`, that is, data structures that satisfy the invariant. For example, the following are elements of `I`: {`root=null`}, {`root`$\neq$`null`, `root.color=BLACK`, `root.left=null`, `root.right=null`}. Figure 19 presents another example, with a BDD representing the same set as in Figure 18, but with the order of the fields in the BDD reversed. Figure 20 presents a BDD representing all `RedBlackTrees` of maximum height 3 that satisfy the invariant in Figure 10.

Note that in Figures 18, 19, and 20, we do not include the field `parent` in the BDD. This is because our analyses detect that `parent` is a *derived* field. That is, given any reachable node in a tree, there is exactly one possible value of `parent` that satisfies the invariant. It is therefore unnecessary to store the values of the `parent` fields in the set `S` in Figure 17, because given any element of `S`, one can reconstruct the values of all the `parent` fields. Similarly, we do not include the ghost field `blackHeight` (Line 7 in Figure 10) in the BDDs, because it is a derived field.

A good field ordering is the key to keeping the BDD size small. In Figures 19 and 20, we order the fields in the BDDs based on a post-order traversal of the tree backbone of the data structure. In general, this is ordering of fields we use in our system. For objects not on the tree backbone, we include them in the order in which we encounter them as we build the set `I` (as we describe in Section 3.3.3). The above field ordering keeps the fields connected by invariants together in the BDD, which seems to naturally induce a good field ordering and thus compact BDDs.

The above field ordering also makes the search efficient. The reason is as follows. In the BDD package we use (and in the BDD packages we know of), all the BDDs are immutable. A BDD node once created cannot be modified. But different BDDs can share nodes. To make a change to a BDD, the implementation constructs a new BDD by copying all the BDD nodes above the place where the change happens. That means, making a change to the bottom of a BDD is more expensive because there is more copying involved, whereas making a change to the top of a BDD is cheaper. It is therefore better to keep fields that change frequently at the

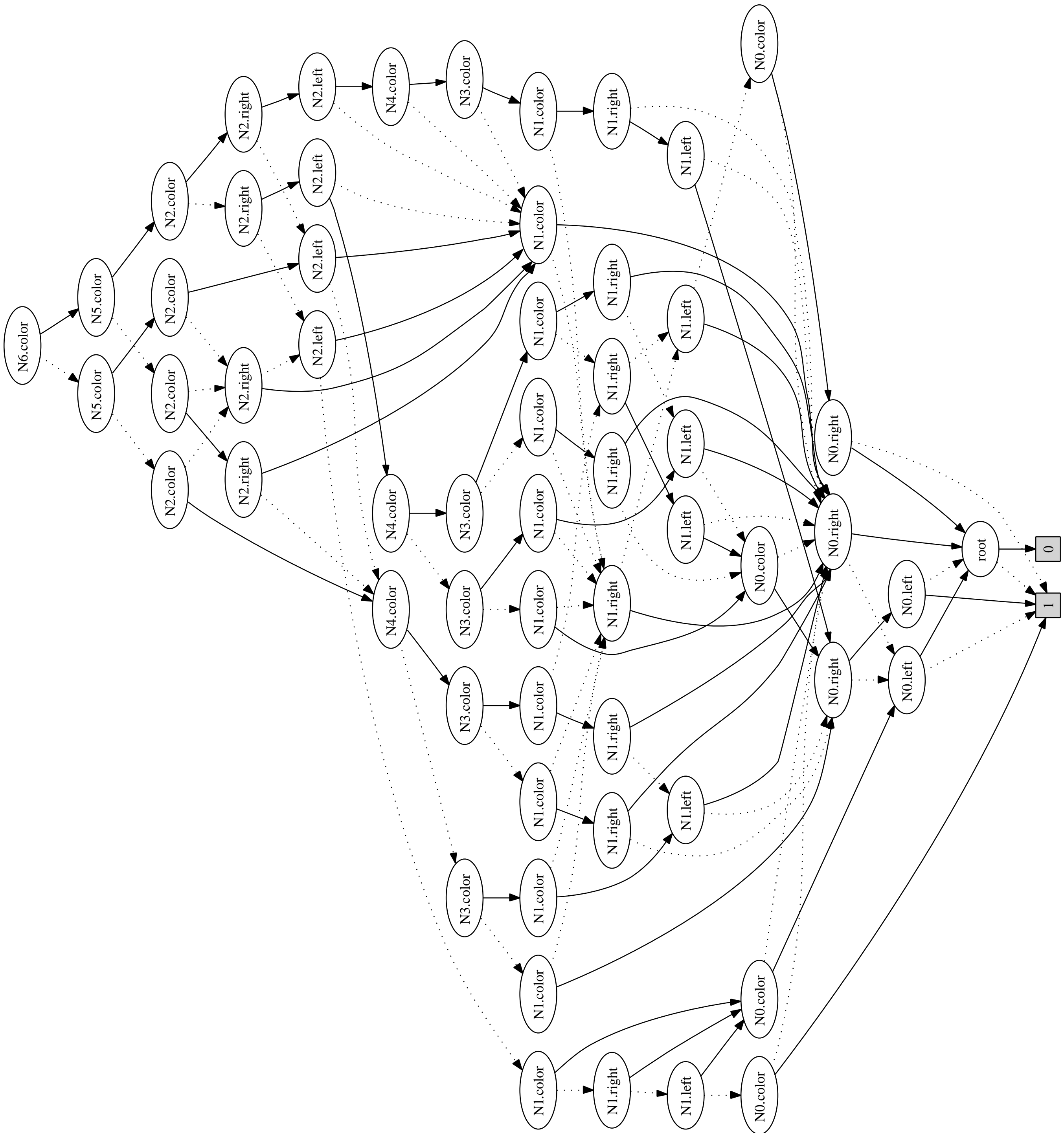

Figure 20. BDD representing the set of all `RedBlackTrees` in the search space in Figure 14 (with maximum height 3) that satisfy the invariant in Figure 10.

Max. Height	Set Size	BDD Size
1	2	0
2	7	12
3	34	58
4	597	226
5	238526	744
6	42715248230	2367
7	1123387228727905854061	7359

Figure 21. For a given maximum height, Column 2 above presents the number of `RedBlackTrees` that satisfy the class invariant in Figure 10, and Column 3 presents the size of the BDD representing the set of all such `RedBlackTrees`.

top of the BDD. In the context of checking a tree-based data structure, such as the `RedBlackTree` in Figure 10, the value of the `root` field is non-null for all valid `RedBlackTrees` except one. Hence the value of the `root` field does not change much as the search progresses. Therefore it is best to keep `root` at the bottom of the BDD. Similarly it is best to keep fields that are near the root of the tree towards the bottom of the BDD. We therefore use the post-order tree traversal ordering for fields as in Figures 19 and 20.

Figure 21 presents the sizes of BDDs representing the set `I` (in Figure 17) for `RedBlackTrees` with different maximum heights. The numbers indicate that as the height h increases, assuming $n = 2^h$, the size of the BDD grows as $O(n \log n)$. The number of elements in `I` however grows exponentially in n, because there are exponentially (w.r.t. n) many `RedBlackTrees` of a given maximum height h. This illustrates how a BDD can provide a compact representation for a very large set of related data structures.

In general, if all the invariants of a tree-based data structure are local (that is, the invariants are specified using only `repOkLocal` and without using `repOk`), then it is easy to see that the size of the BDD representing the set `I` in Figure 17 is always $O(n \log n)$ (where n is the maximum size of the data structure), even though the number of elements in `I` could be exponential in n. This is part of the reason why it is advantageous to use local invariants as much as possible for glass box model checking.

3.3 Search

Figure 17 presented the search algorithm. This section describes how to perform each step of the algorithm efficiently.

3.3.1 Manipulating the Search Space

This section describes how to execute Lines 5, 6, and 9 in Figure 17 efficiently. We described in Section 3.2 how we use BDDs to represent our search space. Given that, Line 5, checking if a set is empty, is a constant time BDD operation. Line 9, computing the difference of two sets, is usually an efficient BDD operation. In particular, even if the two sets contain exponentially many elements, operations on the sets can be performed efficiently using their compact BDD representations. Line 6, choosing an element from a non-empty set, takes time linear in the number of fields in the BDD. If the set has more than one element, our system chooses the lexicographically least element of the set. That ensures that the search space remains contiguous and structured as much as possible, which in turn leads to smaller BDD representations of the search space as the search progresses.

3.3.2 Pruning the Search Space

This section describes how to execute Lines 7 and 8 in Figure 17 efficiently. The key to making glass box model checking efficient is to identify as large a set `T` as possible in Figure 17, that is, given a transition `t`, to identify as many transitions similar to `t` as possible, so that they can be pruned away without explicitly checking them. This section describes how we monitor the program as we check a transition `t` (Line 7) and how we use the results of the monitoring to construct the set `T` of transitions similar to `t` (Line 8).

3.3.2.1 Tracking Fields Read

Consider the Stack example in Figure 8. Consider checking that the transition in Figure 15 preserves the `Stack` invariant. As our checker runs the `pop` method, it monitors the set of fields that `pop` reads. In this case, `pop` reads `head`, `N0.value`, and `N0.next`. That means, regardless of the values of the remaining fields, `pop` will still behave similarly. Our system then determines (as we explain below) that regardless of the values of the remaining fields, if the invariant holds before `pop`, then the invariant holds after `pop`. Our system therefore prunes all elements of the search space where `head=N0`, `N0.value=O0`, `N0.next=N1`, and *operation*=`pop`.

The above technique, in effect, detects *don't care* fields in a transition `t`, and suggests that all transitions `t'` that differ from `t` only at the don't care fields be pruned from the search space. However, we need additional mechanisms to ensure that the system is sound. To see why the above technique alone is unsound, consider the following example:

```
1  class SoundnessDemo {
2    private boolean x, y;
3    public  boolean repOk() { return !x || y; }
4    public  void    flipX() { x = !x;         }
5  }
```

The invariant `repOk` returns true iff `x` implies `y`. Suppose we invoke `flipX` on `x=false` and `y=true`. The invariant holds before and after the transition. `flipX` reads only `x`; `y` is a don't care. The above technique suggests that `flipX` will perhaps verify on all states where `x=false` (and therefore those elements be pruned from the search space). But the suggestion is incorrect because `flipX` does not verify on `x=false` and `y=false`. The invariant holds before the transition, but not after. The above technique fails on this example because a field that is read (`x`) is related by the invariant to a field that is a don't care (`y`).

3.3.2.2 Identifying Don't Care Fields

To correctly identify the don't care fields of a transition `t` and to soundly prune the search space, our system works as follows. Let $\mathtt{F_t}$ and $\mathtt{F'_t}$ be the set of fields read by `t` and modified by `t` respectively. After executing `t`, our system checks if the class invariant is preserved. If `t` has a postcondition in addition to the class invariant, our system checks the postcondition as well. Let $\mathtt{F_P}$ be the set of fields read by the additional postcondition, if any.

If a data structure invariant only specifies that the tree backbone of the data structure must remain a tree (that is, `repOk` and `repOkLocal` always return true), then our system computes the set of relevant fields F_R as $F_t \cup F_P$. All the fields not in F_R are don't cares. In particular, if F_P is an empty set, then F_R reduces to F_t. Thus, for checking that the `Stack` in Figure 8 preserves its invariant, the technique described in Section 3.3.2.1 is actually sound.

If a data structure invariant specifies a local invariant in addition to the tree backbone (that is, `repOk` always return true but `repOkLocal` does not), then our system works as follows. Recall from Section 3.1.4.2 that to check the local invariant, our system traverses the tree backbone of the data structure and checks that `repOkLocal` returns `true` on every tree node N_i. Also recall that `repOkLocal` is restricted to read only the fields of a node and the fields of its tree children—our system enforces this restriction using dynamic checking. Let F_{N_i} be the set of fields read when our system invokes `repOkLocal` on a tree node N_i, after the execution of `t`. Let F_L be the smallest set such that: (i) if $f \in F_t$ then $f \in F_L$, and (ii) if $f_1 \in F'_t$ and $f_1, f_2 \in N_i$ for any i then $f_2 \in F_L$. Our system computes the set of relevant fields F_R as $F_L \cup F_P$. Finally, all the fields not in F_R are don't cares.

If a data structure invariant specifies a global invariant as well (that is, `repOk` does not always return true), then our system conservatively treats all the fields read by `repOk` as relevant fields. (This is another reason, besides the one at the end of Section 3.2, why using local invariants as much as possible is advantageous.) Let F_G be the set of fields read by `repOk`, after the execution of `t`. Then our system computes the set of relevant fields F_R as $F_L \cup F_G \cup F_P$, where F_L is computed as above. All the fields not in F_R are don't cares.

3.3.2.3 Tracking Information Flow to Improve Precision

This section describes an optimization to more precisely compute the above sets F_t, F_{N_i}, F_G, and F_P. The above algorithm computes the sets by tracking the fields read by the corresponding methods. But sometimes, even though a method reads a field, it does not *depend* on it. That is, the method's return value and its side effects do not depend on the field. The `Point` class below provides an example. Suppose the method returns false on Line 5 because `x==y`. The above analysis assumes that because the method read all three fields `_x`, `_y`, `_z`, the return value depends on all the fields—even though it depends only on `_x` and `_y`.

```
1  class Point {
2    private int _x, _y, _z;
3    public  boolean isSkewed() {
4      int x=_x, y=_y, z=_z;
5      if (x == y) return false;
6      if (y == z) return false;
7      if (z == x) return false;
8      return true;
9  }}
```

To make our analysis more precise, we use dynamic information flow tracking. Consider `Stack` in Figure 12. There are nine fields. For every value v the program computes, our system also computes a nine-bit shadow value v' that tracks the input fields from which there is information flow to v. Given an execution trace of the postcondition of a method (such as `repOkLocal`, `repOk`, or any additional asserts) we say that the postcondition only depends on the set of fields from which there is information flow to the value returned by postcondition. Given an execution trace of a method `m` being checked, we say that `m` only depends on the set of fields from which there is information flow to the value returned by the postcondition that is run after `m`'s execution.

One thing we must be careful about is that information flow analysis [10, 40] is different from dynamic slicing [29, 53], as the following example shows.

```
1  class InfoFlowDemo {
2    private boolean b;
3    public  boolean m() {
4      boolean x = false;
5      if (b) x = true;
6      return x;
7  }}
```

There is information flow from `b` to `x` above. But if `b` is `false`, then `x` is not control or data dependent on `b` because the branch on Line 5 is not taken. If we use dynamic slicing, then on running the method with `b=false` we would incorrectly conclude that the method does not depend on `b` and that the method always returns false.

To avoid such incorrect conclusions and the consequent incorrect pruning, our analysis conservatively assumes that after any join point in the control flow graph, all variables that appear on the left hand side of an assignment on either side of the branch depend on the corresponding branch conditional. Thus the return value `x` above depends on `b`.

3.3.2.4 Pruning Isomorphic Structures

Compare the `Stack` in Figure 15 with a `Stack` where `head=N0`, `N0.next=N1`, `N0.value=O2`, `N1.next=null`, and `N1.value=O1`. The two are isomorphic. Clearly, once we check `pop` on the first `Stack`, it is redundant to check `pop` on the second `Stack`. Our checker avoids checking isomorphic structures as follows. After checking the transition in Figure 15, the analyses in the previous sections conclude that the `pop` operation on all `Stacks` with `head=N0`, `N0.next=N1`, and `N0.value=O0` can be pruned. Our isomorphism analysis then determines that `pop` can also be pruned from all structures that satisfy the following formula: ($\texttt{head}=\texttt{N0} \wedge \texttt{N0.next}=\texttt{N1} \wedge \texttt{N0.value}\neq\texttt{null}$).

In general, to construct the formula, our system traverses all the relevant fields of a transition `t` in the order in which they were read by `t`. Each time it encounters a fresh object `o` that a field points to, it includes (in the formula) all other transitions `t'` where the fields read by the traversal so far have the same values except that instead of `o` in `t` there is another fresh object `o'` in `t'`. Our system then prunes all transitions denoted by the formula using efficient BDD operations. The above technique is sound if `t` is deterministic.

Note that some black box checkers also prune isomorphs using heap canonicalization [24, 37]. The difference is, in heap canonicalization, once a checker *visits* a state, it canonicalizes the state and checks if the state has been previously visited. In our isomorphism pruning, once our checker checks a

Benchmark	Max Size	Transitions	BDD Nodes Created				Time (seconds)		
			Total	Initialization	Checking	Max BDD Size	Total	Initialization	Checking
Stack	1	**4**	9	4	5	2	0.016	0.014	0.002
	2	**5**	14	6	8	3	0.016	0.014	0.002
	3	**5**	16	8	8	3	0.017	0.015	0.002
	4	**5**	18	10	8	3	0.017	0.015	0.002
	5	**5**	20	12	8	3	0.017	0.015	0.002
	6	**5**	22	14	8	3	0.017	0.015	0.002
	7	**5**	24	16	8	3	0.017	0.014	0.002
	8	**5**	26	18	8	3	0.017	0.015	0.002
	...								
	16	**5**	42	34	8	3	0.018	0.016	0.002
	32	**5**	74	66	8	3	0.018	0.015	0.002
	64	**5**	140	130	10	3	0.020	0.017	0.003
	128	**5**	268	258	10	3	0.023	0.020	0.003
Queue	1	**5**	17	6	11	4	0.019	0.017	0.002
	2	**7**	36	10	26	8	0.020	0.017	0.002
	3	**8**	46	14	32	10	0.020	0.018	0.002
	4	**9**	61	18	43	10	0.020	0.017	0.003
	5	**10**	68	22	46	10	0.021	0.017	0.004
	6	**11**	75	26	49	10	0.021	0.017	0.003
	7	**12**	101	30	71	10	0.021	0.017	0.004
	8	**13**	114	34	80	10	0.022	0.018	0.004
	...								
	16	**21**	272	66	206	18	0.028	0.018	0.010
	32	**37**	834	130	704	34	0.050	0.019	0.030
	64	**69**	2873	258	2615	66	0.086	0.021	0.064
	128	**133**	12292	514	11778	130	0.118	0.028	0.090
HeapArray	1	**4**	9	4	5	2	0.016	0.015	0.001
	2	**7**	58	16	42	10	0.016	0.014	0.002
	3	**11**	245	72	173	40	0.018	0.015	0.002
	4	**17**	565	76	489	71	0.019	0.015	0.004
	5	**24**	2053	350	1703	226	0.022	0.015	0.006
	6	**34**	4398	597	3801	470	0.029	0.016	0.013
	7	**45**	9032	959	8073	960	0.035	0.016	0.018
	8	**57**	16621	672	15949	2467	0.045	0.016	0.029
	9	**70**	55382	3092	52290	6274	0.080	0.018	0.062
	10	**86**	81364	3963	77401	9105	0.114	0.018	0.096
	11	**103**	145832	5387	140445	13656	0.198	0.019	0.179
	12	**122**	254985	7299	247686	19472	0.318	0.020	0.298
	13	**142**	556949	10407	546542	37432	0.647	0.022	0.625
	14	**165**	1344220	17506	1326714	107759	2.098	0.026	2.072
	Max Height								
RedBlackTree	1	**6**	28	12	16	5	0.022	0.019	0.003
	2	**28**	493	139	354	25	0.028	0.020	0.007
	3	**108**	3860	403	3457	88	0.066	0.021	0.045
	4	**366**	24400	1645	22755	340	0.134	0.022	0.112
	5	**1094**	128314	3906	124408	1311	0.340	0.027	0.313
	6	**2968**	781369	9352	772017	4868	1.543	0.035	1.508
	7	**7524**	6263228	23095	6240133	17434	10.340	0.055	10.285
	Max Height / Max Degree								
FileSystem	2 2	**62**	784	58	726	21	0.035	0.019	0.015
	3 2	**135**	1764	42	1722	14	0.057	0.019	0.038
	4 2	**240**	3728	131	3597	41	0.075	0.019	0.056
	5 2	**380**	6531	162	6369	40	0.096	0.020	0.076
	6 2	**558**	9984	218	9766	56	0.109	0.020	0.089
	7 2	**777**	13467	87	13380	18	0.125	0.020	0.105
	8 2	**1040**	21755	350	21405	86	0.147	0.021	0.126
	9 2	**1350**	31031	392	30639	79	0.167	0.020	0.146
	10 2	**1710**	40954	509	40445	105	0.203	0.020	0.183
	2 3	**102**	1132	152	980	31	0.051	0.020	0.031
	3 3	**270**	2754	108	2646	20	0.076	0.020	0.055
	4 3	**560**	6707	1035	5672	93	0.115	0.021	0.093
	5 3	**1005**	12115	1587	10528	95	0.158	0.022	0.135
	6 3	**1638**	20789	3650	17139	170	0.234	0.025	0.209
	7 3	**2492**	21131	496	20635	27	0.266	0.029	0.237
	8 3	**3600**	44059	12226	31833	348	0.583	0.037	0.546
	2 4	**182**	1903	442	1461	49	0.065	0.020	0.045
	3 4	**675**	5446	273	5173	21	0.114	0.021	0.093
	4 4	**1840**	24672	12443	12229	289	0.324	0.034	0.289
	5 4	**4130**	55614	27500	28114	349	0.935	0.053	0.882
	2 5	**342**	3818	1420	2398	83	0.089	0.022	0.067
	3 5	**1890**	13109	765	12344	26	0.243	0.032	0.211

Figure 23. Experimental results for glass box model checking.

```
1  I = F
2
3  // Enforce local constraints
4
5  for (all nodes N in post-order traversal of tree backbone) {
6    I' = I
7    I = Empty set
8    while (I' is not empty) {
9      t = Any transition in I'
10     r = repOkLocal holds for t on node N
11     T = Set of all transitions with same repOkLocal behavior
12     I' = I' - T
13     if (r) I = I + T
14   }
15 }
16
17 // Enforce global constraints
18
19 I' = I
20 I = Empty set
21 while (I' is not empty) {
22   t = Any transition in I'
23   r = repOk holds for t
24   T = Set of all transitions with same repOk behavior
25   I' = I' - T
26   if (r) I = I + T
27 }
```

Figure 22. Pseudo-code for initializing the search space. This is the expanded version of Line 3 in Figure 17.

transition `t`, it computes a formula `F` denoting (often an exponentially large number of) transitions isomorphic to `t`, and prunes `F` from the search space (often with a small number of BDD operations). Our checker *never visits* `F`'s transitions.

In addition to heap symmetries, our checker also handles other symmetries. In particular, if the actual values of integers in a program do not matter but only their relative ordering matters, our checker prunes states which are symmetric in the above respect using efficient BDD operations.

3.3.3 Initializing the Search Space

This section describes how to execute Lines 2, 3, and 4 in Figure 17 efficiently. Line 2 builds a BDD `F` that represents the search space described by a given finitization (e.g., the search spaces in Figures 12 and 14). It takes linear time. Lines 3 and 4 initialize the search space with the set `I` of all structures that satisfy the invariant. The pseudo code for constructing the set `I` is shown in Figure 22. First, it initializes `I` to `F`. It then performs a post-order traversal of the tree backbone of the data structure. Each time it encounters a node N, it constrains `I` with the local constraints specified by `repOkLocal` (Lines 6-14 in Figure 22). Finally, it further constrains `I` with the global constraints specified by `repOk` (Lines 19-27 in Figure 22). Note that the above algorithm for initializing the search space is similar to the algorithm for performing the search (Figure 17).

The last part of the above algorithm (Lines 19-27 in Figure 22) is also similar to our previous work on Korat [2] for generating all structures satisfying a given global invariant (`repOk`), except that in this paper we use information flow tracking to improve the precision of the analysis (Section 3.3.2.3) and we use BDDs to represent the search space which leads to better pruning. (Recall that Korat imposes a linear order on the search space and keeps all unexplored elements contiguous at the end of the linear order. While this makes the search space management efficient, it also means that Korat can only prune a subset of elements its analyses identify, so that all unexplored elements remain contiguous at the end. Our checker can prune all the elements its analyses identify because it uses BDDs.)

The main difference between Korat and our glass box model checker, however, is that Korat ultimately works like a black box model checker. That is, Korat generates every valid state (within a bounded domain) and checks every operation on every state. Our glass box checker, on the other hand, detects redundancies in the state space and prunes away a large number of states and operations on states without explicitly checking them. We present experimental results comparing Korat with glass box model checking in Section 4.

4. Experimental Results

This section presents our preliminary experimental results. We implemented a rudimentary glass box model checker as described in this paper. We extended the Polyglot [41] compiler framework to automatically instrument programs to perform our dynamic analysis (described in Section 3.3.2), and to automatically generate the finitization descriptions (described in Section 3.1.5). We used JavaBDD [50] for BDDs, which is built on top of the BuDDy package [31]. We performed all our experiments on a Linux Fedora Core 4 machine with a Pentium 4 3.2 GHz processor and 1 GB memory using Sun's Java 1.4.2_08.

We present results for the following benchmarks: (a) `Stack` shown in Figure 8, with methods *push* and *pop*; (b) `Queue` shown in Figure 2, implemented using the Stack in Figure 8, with methods *enqueue* and *dequeue*; (c) `HeapArray` [8], an array based implementation of a binary heap to represent a priority queue, with methods *insert* and *extractMin*; (d) `RedBlackTree` [8], from java.util.TreeMap, with methods *get*, *put*, and *remove*; and (e) `FileSystem`, adopted from the Daisy file system benchmark [11], with methods *lookup*, *create*, *unlink*, *mkdir*, and *rmdir*. For each benchmark, we ran the model checking tools to check that the implementation preserves the data structure invariants.

We checked each benchmark on states up to a maximum size, where: a Stack of maximum size n has at most n nodes and at most n different non-null values and possibly some null values; a Queue of maximum size n has at most n nodes in the `front` Stack, at most n nodes in the `back` Stack, and at most n different non-null values and possibly some null values; a HeapArray of maximum size n has at most n nodes and at most n different non-null values; a RedBlackTree of maximum size h has at most h height, at most $2^h - 1$ different keys, and at most $2^h - 1$ different non-null values and possibly some null values; and a FileSystem of maximum size (h, d) has at most h height and at most d degree, i.e., each directory has at most d entries.

Figure 23 presents our experimental results. It reports the following numbers for glass box model checking. It shows the number of transitions that are explicitly checked by our checker (that is, the number of times the loop in Lines 5-10

Benchmark	Max Size	Glass Box **Transitions**	Glass Box BDD Nodes	Glass Box Time (s)	JPF **Transitions**	JPF States	JPF Time (s)
`Stack`	1	**4**	9	0.016	**33**	15	0.533
	2	**5**	14	0.016	**141**	83	0.669
	3	**5**	16	0.017	**1033**	687	1.349
	4	**5**	18	0.017	**10949**	7819	7.233
	5	**5**	20	0.017	**149313**	111983	96.529
	6	**5**	22	0.017	**2471943**	1922551	1661.628
	7	**5**	24	0.017			timeout
	8	**5**	26	0.017			timeout
	9	**5**	28	0.017			timeout
	10	**5**	30	0.017			timeout
	...						
	16	**5**	42	0.018			timeout
	32	**5**	74	0.018			timeout
	64	**5**	140	0.020			timeout
	128	**5**	268	0.023			timeout
`Queue`	1	**5**	17	0.019	**601**	299	1.121
	2	**7**	36	0.020	**89756**	53852	47.598
	3	**8**	46	0.020			timeout
	4	**9**	61	0.020			timeout
	5	**10**	68	0.021			timeout
	6	**11**	75	0.021			timeout
	7	**12**	101	0.021			timeout
	8	**13**	114	0.022			timeout
	...						
	16	**21**	272	0.028			timeout
	32	**37**	834	0.050			timeout
	64	**69**	2873	0.086			timeout
	128	**133**	12292	0.118			timeout
`HeapArray`	1	**4**	9	0.016	**19**	7	0.434
	2	**7**	58	0.016	**133**	67	0.521
	3	**11**	245	0.018	**1816**	1090	1.464
	4	**17**	565	0.019	**39565**	26377	15.184
	5	**24**	2053	0.022			timeout
	6	**34**	4398	0.029			timeout
	7	**45**	9032	0.035			timeout
	8	**57**	16621	0.045			timeout
	9	**70**	55382	0.080			timeout
	10	**86**	81364	0.114			timeout
	11	**103**	145832	0.198			timeout
	12	**122**	254985	0.318			timeout
	13	**142**	556949	0.647			timeout
	14	**165**	1344220	2.098			timeout
	Max Height						
`RedBlackTree`	1	**6**	28	0.022	**49**	19	0.617
	2	**28**	493	0.028			timeout
	3	**108**	3860	0.066			timeout
	4	**366**	24400	0.134			timeout
	5	**1094**	128314	0.340			timeout
	6	**2968**	781369	1.543			timeout
	7	**7524**	6263228	10.340			timeout
	Max Height / Max Degree						
`FileSystem`	2 2	**62**	784	0.035	**12901**	7482	6.883
	3 2	**135**	1764	0.057			timeout
	4 2	**240**	3728	0.075			timeout
	5 2	**380**	6531	0.096			timeout
	6 2	**558**	9984	0.109			timeout
	7 2	**777**	13467	0.125			timeout
	8 2	**1040**	21755	0.147			timeout
	9 2	**1350**	31031	0.167			timeout
	10 2	**1710**	40954	0.203			timeout
	2 3	**102**	1132	0.051			timeout
	3 3	**270**	2754	0.076			timeout
	4 3	**560**	6707	0.115			timeout
	5 3	**1005**	12115	0.158			timeout

Figure 24. Comparing glass box model checking to JPF.

Benchmark	Max Size	Glass Box **Transitions**	Glass Box BDD Nodes	Glass Box Time (s)	Black Box **Transitions**	Black Box States	Black Box Time (s)	Black Box With Abstraction **Transitions**	Black Box With Abstraction States	Black Box With Abstraction Time (s)
Stack	1	**4**	9	0.016	**9**	3	0.008	**4**	2	0.010
	2	**5**	14	0.016	**32**	8	0.009	**6**	3	0.009
	3	**5**	16	0.017	**115**	23	0.012	**8**	4	0.009
	4	**5**	18	0.017	**450**	75	0.030	**10**	5	0.008
	5	**5**	20	0.017	**1946**	278	0.067	**12**	6	0.009
	6	**5**	22	0.017	**9240**	1155	0.137	**14**	7	0.009
	7	**5**	24	0.017	**47655**	5295	0.437	**16**	8	0.009
	8	**5**	26	0.017	**264420**	26442	2.242	**18**	9	0.009
	9	**5**	28	0.017	**1566587**	142417	15.648	**20**	10	0.009
	10	**5**	30	0.017	**9851844**	820987	105.304	**22**	11	0.009
	...									
	16	**5**	42	0.018			timeout	**34**	17	0.012
	32	**5**	74	0.018			timeout	**66**	33	0.026
	64	**5**	140	0.020			timeout	**130**	65	0.041
	128	**5**	268	0.023			timeout	**258**	129	0.098
Queue	1	**5**	17	0.019	**27**	9	0.011	**8**	4	0.011
	2	**7**	36	0.020	**356**	89	0.044	**18**	9	0.012
	3	**8**	46	0.020	**6610**	1322	0.161	**32**	16	0.014
	4	**9**	61	0.020	**176430**	29405	2.126	**50**	25	0.016
	5	**10**	68	0.021	**6330912**	904416	92.334	**72**	36	0.019
	6	**11**	75	0.021			timeout	**98**	49	0.025
	7	**12**	101	0.021			timeout	**128**	64	0.029
	8	**13**	114	0.022			timeout	**162**	81	0.036
	...									
	16	**21**	272	0.028			timeout	**578**	289	0.069
	32	**37**	834	0.050			timeout	**2178**	1089	0.211
	64	**69**	2873	0.086			timeout	**8450**	4225	1.231
	128	**133**	12292	0.118			timeout	**33282**	16641	11.148
HeapArray	1	**4**	9	0.016	**4**	2	0.007	**4**	2	0.007
	2	**7**	58	0.016	**18**	6	0.008	**18**	6	0.008
	3	**11**	245	0.018	**96**	24	0.013	**96**	24	0.013
	4	**17**	565	0.019	**550**	110	0.035	**550**	110	0.035
	5	**24**	2053	0.022	**3984**	664	0.088	**3984**	664	0.088
	6	**34**	4398	0.029	**31605**	4515	0.284	**31605**	4515	0.284
	7	**45**	9032	0.035	**333568**	41696	2.382	**333568**	41696	2.382
	8	**57**	16621	0.045	**3101139**	344571	25.261	**3101139**	344571	25.261
	9	**70**	55382	0.080			timeout			timeout
	10	**86**	81364	0.114			timeout			timeout
	11	**103**	145832	0.198			timeout			timeout
	12	**122**	254985	0.318			timeout			timeout
	13	**142**	556949	0.647			timeout			timeout
	14	**165**	1344220	2.098			timeout			timeout
	Max Height									
RedBlackTree	1	**6**	28	0.022	**12**	3	0.013	**6**	2	0.012
	2	**28**	493	0.028	**936**	52	0.084	**99**	11	0.020
	3	**108**	3860	0.066	**18143370**	259191	431.682	**11781**	561	0.298
	4	**366**	24400	0.134			timeout			timeout
	5	**1094**	128314	0.340			timeout			timeout
	6	**2968**	781369	1.543			timeout			timeout
	7	**7524**	6263228	10.340			timeout			timeout
	Max Height / Max Degree									
FileSystem	2 2	**62**	784	0.035	**570**	19	0.045	**210**	7	0.032
	3 2	**135**	1764	0.057	**14820**	247	0.201	**900**	15	0.063
	4 2	**240**	3728	0.075	**552900**	5529	6.267	**3100**	31	0.117
	5 2	**380**	6531	0.096			timeout	**9450**	63	0.226
	6 2	**558**	9984	0.109			timeout	**26670**	127	0.569
	7 2	**777**	13467	0.125			timeout	**71400**	255	1.682
	8 2	**1040**	21755	0.147			timeout	**183960**	511	5.088
	9 2	**1350**	31031	0.167			timeout	**460350**	1023	15.066
	10 2	**1710**	40954	0.203			timeout	**1125850**	2047	40.609
	2 3	**102**	1132	0.051	**222670**	3181	2.573	**5110**	73	0.147
	3 3	**270**	2754	0.076			timeout	**851955**	4369	18.740
	4 3	**560**	6707	0.115			timeout			timeout
	5 3	**1005**	12115	0.158			timeout			timeout

Figure 25. Comparing glass box model checking to black box model checking.

		Glass Box			Korat		
Benchmark	Max Size	**Transitions**	BDD Nodes	Time (s)	**Transitions**	States Considered	Time (s)
Stack	1	**4**	9	0.016	**6**	7	0.001
	2	**5**	14	0.016	**9**	12	0.001
	3	**5**	16	0.017	**12**	18	0.000
	4	**5**	18	0.017	**15**	25	0.001
	5	**5**	20	0.017	**18**	33	0.001
	6	**5**	22	0.017	**21**	42	0.001
	7	**5**	24	0.017	**24**	52	0.001
	8	**5**	26	0.017	**27**	63	0.002
	9	**5**	28	0.017	**30**	75	0.002
	10	**5**	30	0.017	**33**	88	0.003
	...						
	16	**5**	42	0.018	**51**	187	0.007
	32	**5**	74	0.018	**99**	627	0.018
	64	**5**	140	0.020	**195**	2275	0.053
	128	**5**	268	0.023	**387**	8643	0.259
Queue	1	**5**	17	0.019	**12**	15	0.001
	2	**7**	36	0.020	**27**	39	0.001
	3	**8**	46	0.020	**48**	78	0.002
	4	**9**	61	0.020	**75**	135	0.005
	5	**10**	68	0.021	**108**	213	0.007
	6	**11**	75	0.021	**147**	315	0.010
	7	**12**	101	0.021	**192**	444	0.015
	8	**13**	114	0.022	**243**	603	0.019
	...						
	16	**21**	272	0.028	**867**	3315	0.059
	32	**37**	834	0.050	**3267**	21219	0.363
	64	**69**	2873	0.086	**12675**	149955	3.972
	128	**133**	12292	0.118	**49923**	1123203	57.416
HeapArray	1	**4**	9	0.016	**4**	4	0.000
	2	**7**	58	0.016	**18**	19	0.001
	3	**11**	245	0.018	**96**	106	0.001
	4	**17**	565	0.019	**550**	643	0.006
	5	**24**	2053	0.022	**3984**	4606	0.023
	6	**34**	4398	0.029	**31605**	36692	0.128
	7	**45**	9032	0.035	**333568**	370714	0.879
	8	**57**	16621	0.045	**3101139**	3579511	10.852
	9	**70**	55382	0.080	**36626580**	41004532	95.155
	10	**86**	81364	0.114	**429394636**	483881209	1356.158
	11	**103**	145832	0.198			timeout
	12	**122**	254985	0.318			timeout
	13	**142**	556949	0.647			timeout
	14	**165**	1344220	2.098			timeout
	Max Height						
RedBlackTree	1	**6**	28	0.022	**8**	14	0.017
	2	**28**	493	0.028	**144**	460	0.035
	3	**108**	3860	0.066	**16044**	105779	0.858
	4	**366**	24400	0.134	**155496600**	1236548801	16023.741
	5	**1094**	128314	0.340			timeout
	6	**2968**	781369	1.543			timeout
	7	**7524**	6263228	10.340			timeout

Figure 26. Comparing glass box model checking to Korat.

in Figure 17 is executed). It shows the number of BDD nodes created, as a measure of the search space management overhead. It also shows the time taken by our checker. Note that we did not yet optimize the execution time of our checker, but we report it here nonetheless to provide a rough idea. For the number of BDD nodes created and the time taken, the figure shows the totals as well as the numbers separately for the initialization phase (Lines 2-4 in Figure 17) and the checking phase (Lines 5-10 in Figure 17). Finally, the figure also shows the maximum size of the BDD representing the search space (set `S` in Figure 17).

Note that in Figure 23, for checking the `Stack`, our glass box checker checks only $O(1)$ transitions regardless of the size of the `Stack`. This is because `push` and `pop` touch only a constant number of fields at the beginning of the linked list. For checking the `Queue`, our glass box checker checks $O(n)$ transitions, as explained in Section 2.2. For the `HeapArray` and the `RedBlackTree`, the growth in the number of transitions appears to be roughly $O(n^2)$ (where n is the maximum size of the `HeapArray` and $h = \log n$ is the maximum height of the `RedBlackTree`). However, for the `HeapArray`, the search space management overhead dominates the cost. We are currently exploring other search space representation techniques (e.g., an incremental SAT solver) to see if our search space management overhead can be further reduced.

Figure 24 presents results of comparing the performance of our glass box model checker with JPF [48] (version 4). For JPF, we wrote a test harness for each benchmark and we marked all the methods in the benchmarks to be *atomic* (because the benchmarks are all single threaded programs). We report the number of transitions explicitly checked by JPF, as well as the number of unique states visited within the finite bounds (as a measure of the space overhead, as JPF is a stateful checker). We timeout if the tool runs out of memory or if it takes too long. The results show how JPF takes exponentially more time as the size of the structures increases. Our glass box checker scales much better.

While running experiments with JPF, we noticed that JPF sometimes does not detect that two states are isomorphic, perhaps because of their different memory layouts. It therefore visits a lot more states than necessary. To make for a fairer comparison, we implemented our own black box checker that accurately detects heap isomorphisms. Figure 25 presents results of comparing the performance of our glass box model checker with our black box model checker. The results clearly indicate that glass box model checking scales significantly better than black box model checking.

We also extended our black box model checker such that if programmers implement an abstraction function [51, 43], then our checker treats all concrete states that map to the same abstract state as isomorphic (in addition to detecting heap isomorphisms). (Abstraction functions can thus speed up model checking but require manual assistance and are error-prone.) We handcoded an abstraction function for each of our benchmarks. For `Stack`, `Queue`, and `RedBlackTree`, the abstraction function ignores the `value` fields because the invariants of the above data structures do not depend on the `values`. For the `FileSystem` benchmark, the general purpose heap isomorphism detector does not work well because `FileSystem` uses array indices instead of pointers—our abstraction function for `FileSystem` maps all such isomorphic concrete states into the same abstract state. Figure 25 also presents results of comparing the performance of our glass box model checker with our black box model checker with abstraction functions. Once again, the results clearly indicate that glass box model checking scales much better.

Finally, Figure 26 presents the results of comparing our glass box model checker with Korat [2]. We report the number of transitions explicitly checked by Korat. We also report the number of candidate states considered by Korat, which includes both transitions checked by Korat (such as Figure 15) and invalid states considered by the Korat (such as Figure 16). We could not run Korat on the `FileSystem` benchmark because the Korat infrastructure does not yet support multidimensional arrays. The results show that our glass box checker scales much better that Korat.

A version of JPF [28] uses lazy initialization of fields to essentially simulate the Korat algorithm. Its asymptotic performance is similar to that of Korat. However, because JPF is a general purpose model checker, it has higher overhead and is slower than Korat. We therefore expect our glass box checker to similarly scale better than [28].

5. Related Work

There are many model checking tools that exhaustively test a program on all possible inputs up to a given size (to handle input nondeterminism) and on all possible nondeterministic schedules (to handle scheduling nondeterminism). Verisoft [15] is a stateless model checker for C programs. Java PathFinder (JPF) [48, 28] is a stateful model checker for Java programs. XRT [20] checks Microsoft CIL programs. Bandera [7] and JCAT [9] translate Java programs into the input language of model checkers like SPIN [22] and SMV [34]. Bogor [12] provides an extensible framework for building software model checkers. CMC [39] is a stateful model checker for C programs that has been used to test large pieces of software including the Linux implementation of TCP/IP and the ext3 file system [38]. However, most of the above work on applying model checking to software focuses on control oriented programs and properties, primarily to verify event sequences with respect to temporal properties. This paper, in contrast, focuses on data oriented (and single threaded) programs and properties.

There has been much research on techniques for reducing the state space of a model checker. Tools such a Slam [1], Blast [21], and Magic [4] use heuristics to construct and check an abstraction of a program (usually predicate abstraction [19]). Abstractions that are too coarse generate false positives, which are then used to refine the abstraction and redo the checking. This technique is known as Counter Example Guided Abstraction and Refinement, or *CEGAR*. There are also many static [15, 16] and dynamic [13] partial order reduction systems for concurrent programs. There are many other symmetry-based reduction techniques as well (e.g., [25]). However, it is unclear how any of the above tech-

niques can be used to significantly reduce the state space of data oriented programs such as a file system implementation or a balanced tree implementation. We believe CEGAR, partial order reduction, and other techniques are complimentary to our glass box approach.

There is a large body of research on specification-based testing. An early paper [18] emphasizes its importance. Many projects automate test case generation from specifications, such as Z specifications [23], UML statecharts [42], or ADL specifications [5]. These specifications typically do not consider data structures that use pointers, and the tools do not generate Java test cases.

Tools such as Alloy [26, 27] and Korat [2, 33] systematically generate all inputs that satisfy a given precondition. A version of JPF [28] uses lazy initialization of fields to essentially simulate the Korat algorithm. However, these tools work as black box checkers because they generate and test every valid state, unlike our glass box checker. (Section 4 compares the performance of such systems with our glass box checker.) Jalloy [47] translates a Java program and its specifications into a SAT formula and uses a constraint solver to check the program. [28] extends Korat by treating integers symbolically. Symstra [52] also treats some integers symbolically.

Tools such as CUTE [45, 17] and a version of JPF [49] use constraint solvers to obtain complete branch coverage (or complete path coverage on paths up to a given length) for testing data structures. However, this approach does not guarantee that an implementation works correctly on all data structures up to a given size. For example, a buggy tree insertion method that does not rebalance the tree might work correctly on a set of test cases that exercise complete branch (or finite path) coverage, but fail on a different test case that makes the tree unbalanced. Therefore, it seems to us that this approach is more suitable for checking control dependent properties rather than data dependent properties.

ESC/Java [14] uses a theorem prover to verify absence of such errors as null pointer dereferences and array bounds violations. Static analyses such as TVLA [44] and PALE [36] offer a promising approach for verifying shape properties of data structures. However, none of the above techniques are currently practical enough to verify, say, the correctness of implementations of *balanced* trees, such as red-black trees. Software model checking, on the other hand, is a general approach that can verify any decidable property, but for inputs bounded by a given size.

6. Conclusions

This paper presents a novel approach to software model checking of data structure properties. Most previous work on software model checking focuses on control oriented programs and properties, primarily to verify event sequences with respect to temporal properties. This paper, in contrast, focuses on data oriented programs and properties. In particular, it deals with verifying properties of data structures. While there is much research on state space reduction techniques for model checkers such as partial order reduction [13, 15, 16] and tools based on predicate abstraction [19] such as Slam [1], Blast [21], or Magic [4], none of these techniques seem to be effective in reducing the state space of data oriented programs. The paper presents novel techniques for detecting similarities in the state space of data structures, and for soundly pruning large numbers of redundant states and operations without explicitly checking them. It also presents novel techniques for efficiently managing extremely large sets of data structures. This results in dramatic speedups. We do not know of any other model checker that scales nearly as well for checking linked data structures. We believe our techniques can make software model checking significantly faster, and thus enable checking of much larger programs and complex program properties than currently possible.

Acknowledgments

We thank Sarfraz Khurshid, Darko Marinov, Madan Musuvathi, and the anonymous referees for their useful comments.

References

[1] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Programming Language Design and Implementation (PLDI)*, June 2001.

[2] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *International Symposium on Software Testing and Analysis (ISSTA)*, July 2002. Winner of an ACM SIGSOFT distinguished paper award.

[3] R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys 24(3)*, 1992.

[4] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *International Conference on Software Engineering (ICSE)*, June 2003.

[5] J. Chang and D. J. Richardson. Structural specification-based testing: Automated support and experimental evaluation. In *Foundations of Software Engineering (FSE)*, September 1999.

[6] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[7] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *International Conference on Software Engineering (ICSE)*, June 2000.

[8] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1991.

[9] C. DeMartini, R. Iosif, and R. Sisto. A deadlock detection tool for concurrent Java programs. *Software—Practice and Experience (SPE) 29(7)*, June 1999.

[10] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. In *Communications of the ACM (CACM) 20(7)*, July 1977.

[11] Daisy file system. Joint CAV/ISSTA Special Event on Specification, Verification, and Testing of Concurrent Software. http://research.microsoft.com/~qadeer/cav-issta.htm.

[12] M. Dwyer, J. Hatcliff, M. Hoosier, and Robby. Building your own software model checker using the Bogor extensible model checking framework. In *Computer Aided Verification (CAV)*, January 2005.

[13] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Principles of Programming Languages (POPL)*, January 2005.

[14] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Programming Language Design and Implementation (PLDI)*, June 2002.

[15] P. Godefroid. Model checking for programming languages using VeriSoft. In *Principles of Programming Languages (POPL)*, January 1997.

[16] P. Godefroid. Partial-order methods for the verification of concurrent systems—An approach to the state-explosion problem. *Lecture Notes in Computer Science (LNCS) 1032*, Springer-Verlag, January 1996.

[17] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Programming Language Design and Implementation (PLDI)*, June 2005.

[18] J. Goodenough and S. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering (TSE) SE-1(2)*, June 1975.

[19] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Computer Aided Verification (CAV)*, June 1997.

[20] W. Grieskamp, N. Tillmann, and W. Shulte. XRT—Exploring runtime for .NET: Architecture and applications. In *Workshop on Software Model Checking (SoftMC)*, July 2005.

[21] T. A. Henzinger, R. Jhala, and R. Majumdar. Lazy abstraction. In *Principles of Programming Languages (POPL)*, January 2002.

[22] G. Holzmann. The model checker SPIN. *Transactions on Software Engineering (TSE) 23(5)*, May 1997.

[23] H.-M. Horcher. Improving software tests using Z specifications. In *International Conference of Z Users*, September 1995.

[24] R. Iosif. Symmetry reduction criteria for software model checking. In *SPIN workshop on Model Checking of Software (SPIN)*, April 2002.

[25] C. N. Ip and D. Dill. Better verification through symmetry. In *Computer Hardware Description Languages*, April 1993.

[26] D. Jackson. Alloy: A lightweight object modeling notation. *Transactions on Software Engineering and Methodology (TOSEM) 11(2)*, April 2002.

[27] S. Khurshid and D. Marinov. TestEra: Specification-based testing of Java programs using SAT. In *Automated Software Engineering (ASE)*, November 2001.

[28] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, April 2003.

[29] B. Korel and J. Laski. Dynamic program slicing. In *Information Processing Letters (IPL) 29(3)s*, October 1988.

[30] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report TR 98-06i, Department of Computer Science, Iowa State University, May 1998.

[31] J. Lind-Nielsen. BuDDy. http://sourceforge.net/projects/buddy.

[32] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. MIT Press, 1986.

[33] D. Marinov. Automatic testing of software with structurally complex inputs. Ph.D. thesis, Massachusetts Institute of Technology, February 2005.

[34] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[35] S. McPeak and G. C. Necula. Data structure specification via local equality axioms. In *Computer Aided Verification (CAV)*, January 2005.

[36] A. Moeller and M. I. Schwartzbach. The pointer assertion logic engine. In *Programming Language Design and Implementation (PLDI)*, June 2001.

[37] M. Musuvathi and D. Dill. An incremental heap canonicalization algorithm. In *SPIN workshop on Model Checking of Software (SPIN)*, August 2005.

[38] M. Musuvathi and D. R. Engler. Using model checking to find serious file system errors. In *Operating System Design and Implementation (OSDI)*, December 2004. Winner of the best paper award.

[39] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. Dill. CMC: A pragmatic approach to model checking real code. In *Operating System Design and Implementation (OSDI)*, December 2002.

[40] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Principles of Programming Languages (POPL)*, January 1999.

[41] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *Compiler Construction (CC)*, April 2003.

[42] J. Offutt and A. Abdurazik. Generating tests from UML specification. In *International Conference on the Unified Modeling Language*, October 1999.

[43] C. Pasareanu, R. Pelanek, and W. Visser. Test input generation for red black trees using abstraction. In *Automated Software Engineering (ASE)*, November 2005.

[44] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *Transactions on Programming Languages and Systems (TOPLAS) 20(1)*, January 1998.

[45] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *European Software Engineering Conference and Foundations of Software Engineering (ESEC/FSE)*, September 2005.

[46] D. Suwimonteerabuth, S. Schwoon, and J. Esparza. jMoped: A Java bytecode checker based on Moped. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, April 2005.

[47] M. Vaziri and D. Jackson. Checking properties of heap-manipulating procedures using a constraint solver. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, April 2003.

[48] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Automated Software Engineering (ASE)*, September 2000.

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

[50] J. Whaley. JavaBDD. http://javabdd.sourceforge.net/.

[51] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Automated Software Engineering (ASE)*, September 2004.

[52] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, April 2005.

[53] X. Zhang and R. Gupta. Cost effective dynamic program slicing. In *Programming Language Design and Implementation (PLDI)*, June 2004.