Efficient Software Model Checking of Data Structure Properties

Paul 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 software 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

Software model checking [1, 2, 4, 7, 8, 13, 16, 44, 21, 36] is a formal verification technique that exhaustively tests 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). Most previous work on software model checking focuses on scheduling nondeterminism to verify event sequences with respect to properties expressed in temporal logics. This paper deals with input nondeterminism. In particular, it focuses on verifying properties of linked data structures.

Consider checking that a red-black tree [9] implementation maintains the red-black tree invariants. Previous model checking approaches such as JPF [44, 26], CMC [36, 35], Korat [2], or Alloy [25] 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.

This paper presents novel language and analysis techniques that significantly speed up software model checking of programs with input nondeterminism. Our key idea is as follows. Consider checking the red-black tree implementation again on trees with at most n nodes. Our checker detects that any red-black tree operation such as insert or delete touches only one path in the tree from 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, 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 model checking 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 software model checking because our checker analyzes the behavior of an operation to prune large portions of the search space. This is in contrast to the traditional *black box* approach that checks

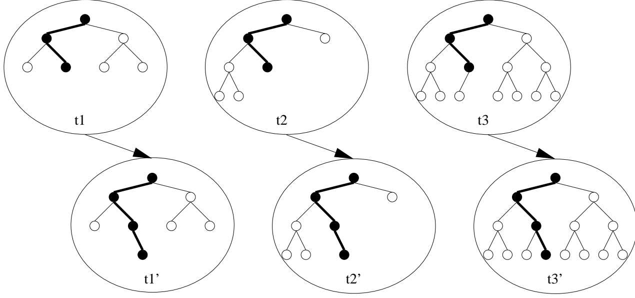

Figure 1: Three red-black trees before and after an `insert` operation. The tree path touched by the operation is highlighted in each case. Once our glass box checker checks the `insert` operation on tree `t1`, it determines that it is redundant to check the same operation on `t2` and `t3`.

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.

Our preliminary results show orders of magnitude improvement over previous model checking approaches. We believe that our techniques can make software model checking significantly faster, and thus enable checking of much larger programs 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 and lists our contributions.

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 [44, 26], CMC [36, 35], Korat [2], or Alloy [25]) 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,

```
 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 `Stack`s.

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, 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 implementaton of `Stack` is shown in Figure 7.)

`Queue`'s class invariant is described by its `repOk` method, as good programming practice suggests [30]. 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. 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

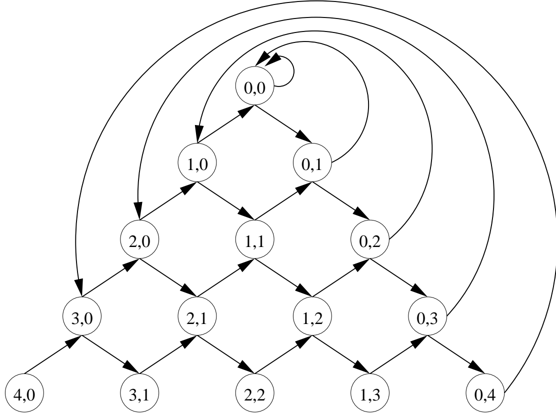

Figure 3: State space of `Queue` with at most $n = 4$ items. State (f,b) has f items in `front Stack` and b in `back`. A black box checker checks $\Omega(n^2)$ state transitions.

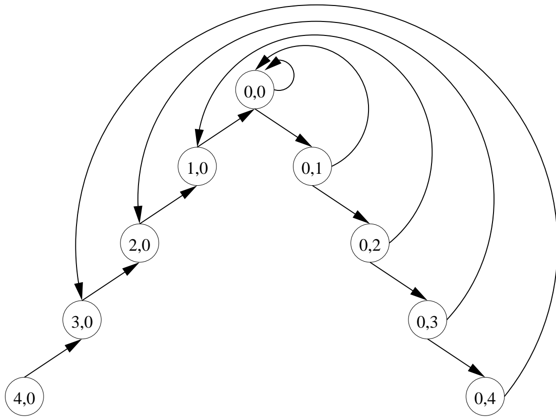

Figure 4: A glass box checker generates only $O(n)$ states and checks only $O(n)$ transitions, yet achieves complete coverage within the bounded domain.

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 [44] or CMC [36, 35] 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 representing 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.

```
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`.

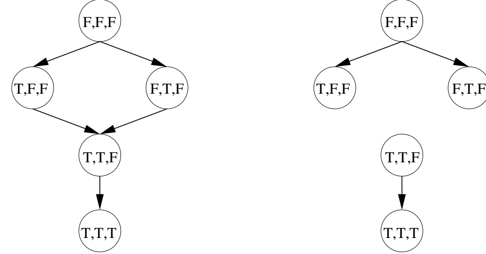

Figure 6: State space of code in Figure 5 (excluding self loops). (b_1,b_2,b_3) implies $\mathtt{x} = b_1$, $\mathtt{y} = b_2$, $\mathtt{z} = b_3$. The figures on the left and right show the state transitions executed by a black box and glass box checker respectively.

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 $O(n)$ 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.

For simplicity, we implicitly assumed in the above example 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

```
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.

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 Checker

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.

3.1 Search Space

This section describes how a glass box model checker organizes its search space.

3.1.1 Defining the Search Space

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 [44] or CMC [36] 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. The next two subsections 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) all the properties specified by programmers (e.g., as assert statements) hold.

3.1.1.1 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 model checking, glass box model checking sometimes involves extra programming effort because programmers have to additionally specify class invari-

```
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 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`.

ants of data structures as described above. However, because glass box checking is orders of magnitude faster than black box checking (as we show later in the paper), we believe writing the class invariants is 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 invariants 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 counter examples 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.

3.1.1.2 Handling Errors in Invariants

Programmers can make two kinds errors in writing the class invariant of a data structure. Let S_R be the set of states

```
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 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).

(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 and detect that the transition does not preserve the invariant. It will then present the transition as a concrete counter example 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 case the user can strengthen the invariant by examining the concrete counter example 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 some on unreachable states as well as.

Thus, even though our glass box checker depends on invariants to cover all states, it is sound in that in 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.

```
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)`.

3.1.2 Specifying Invariants

One way programmers can specify a class invariant is by writing a `repOk` method [30]. The `repOk` method returns true iff the current state (or representation) of an object satisfies its class invariant. The `repOk` method of `Stack` in Figure 7 is an example, which 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 [28], 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 [28].

3.1.3 Specifying Tree-Based Invariants

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 (as we explain later). Our approach is premised on the observation that most data structures are *tree-based.* The next three subsections describe this approach.

3.1.3.1 Specifying the 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 [33]. 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).

```
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)`.

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.

3.1.3.2 Specifying Local Invariants

Consider the `RedBlackTree` in Figure 9. One of its invariants is that for every node N, 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`) $\Longrightarrow$ (N.`left`.`color`$\neq$`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 invariants on a particular instance of a data structure, our system traverses the tree 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.

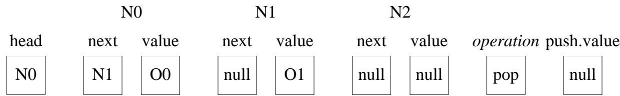

Figure 15: A valid element of the search space in Figure 12 representing the pop operation on a Stack with two items O1 an O2.

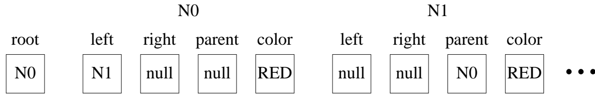

Figure 16: (A portion of) an invalid element of the search space in Figure 14, with the root and its left child both being red.

3.1.3.3 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 [32]. 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 [15]. 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.

3.1.4 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 specify 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`).

```
 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.

3.1.5 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 data 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.

3.1.6 Search

Figure 17 presents the basic 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`, checking `t`, running its analyses to identify the set `T` of other 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.

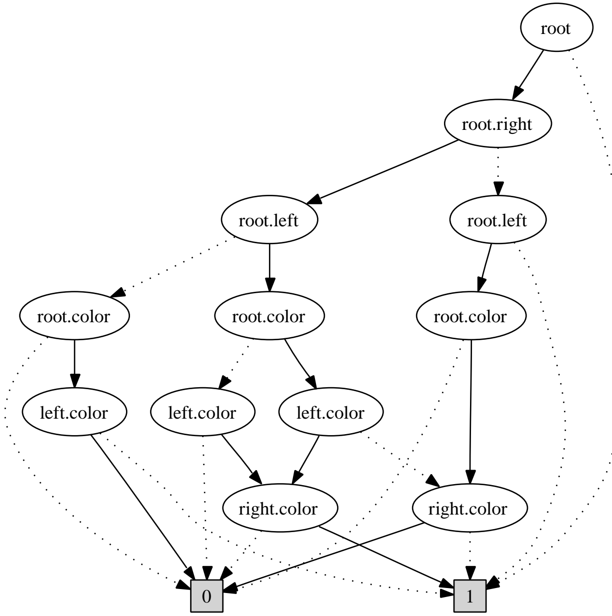

Figure 18: BDD representing the set of all `RedBlackTrees` of maximum height 2 that satisfy the invariant in Figure 10.

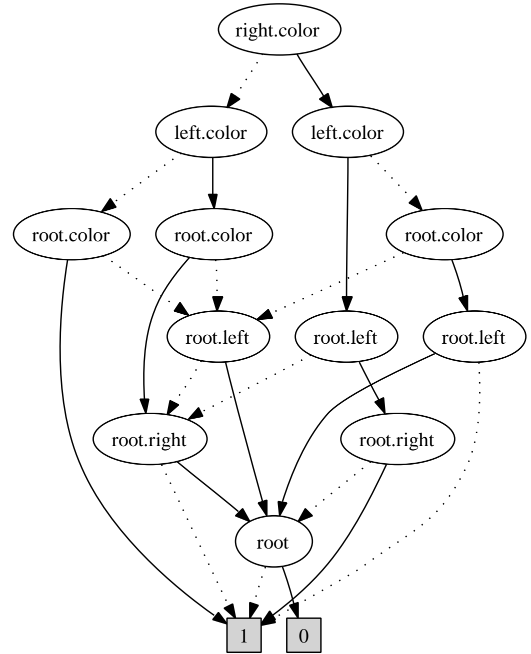

Figure 19: BDD representing the same set as in Figure 18, with the order of the fields in the BDD reversed.

3.2 Search Space Representation

Consider checking the `RedBlackTree` (Figure 10) with `checkRedBlackTree(`h`)`. Say, $n = 2^h$. Then, our checker generates $O(n^2)$ and checks $O(n^2)$ transitions to cover this search space (as we show later). 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`. 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≠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. In general, ghost fields are almost always derived fields.

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 described in Section 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 top of the BDD. In the context of our search (Figure 17), that means it is better to keep fields that are frequently accessed at the top of the BDD. Operations on tree-based

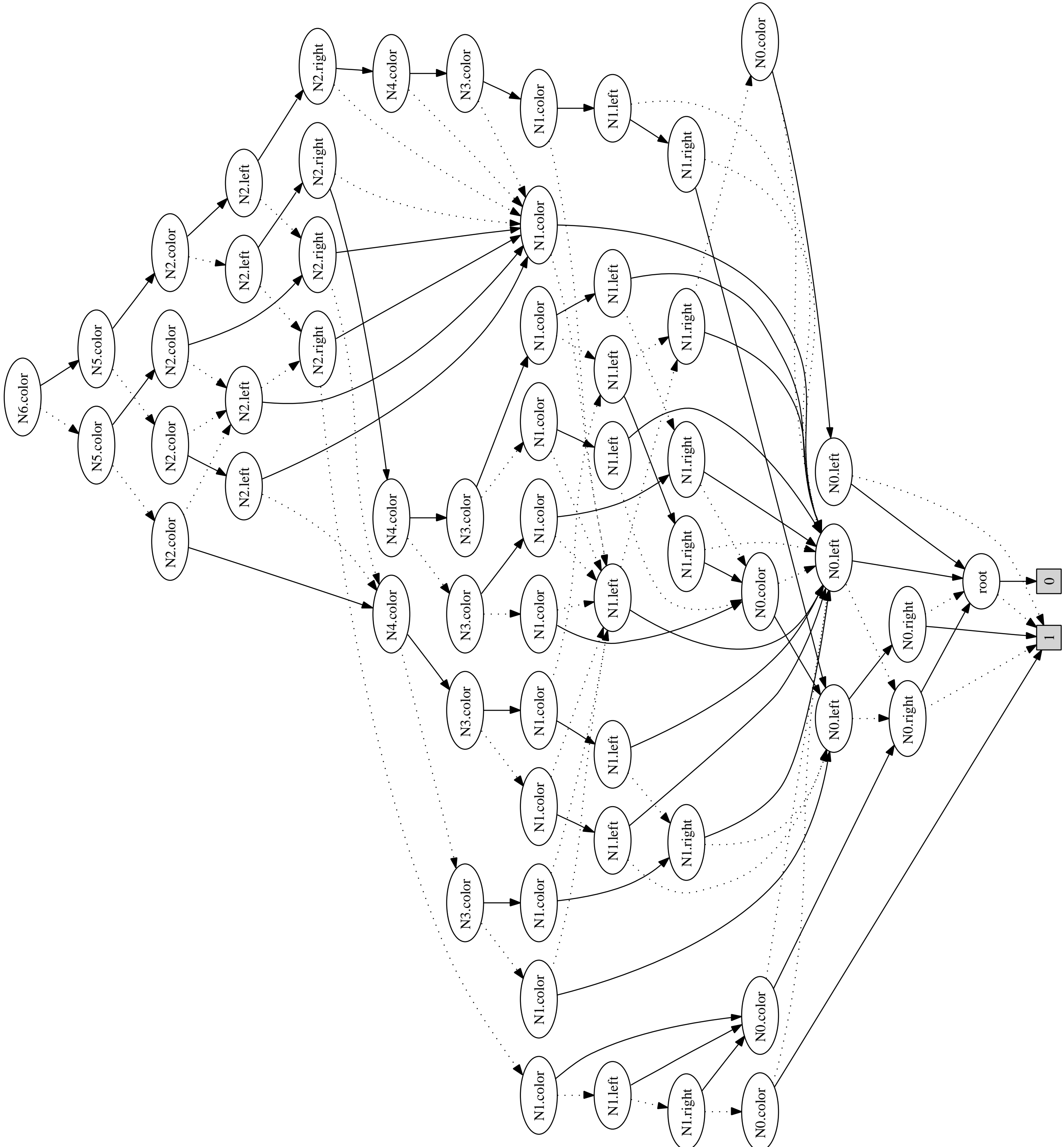

Figure 20: BDD representing the set of all `RedBlackTrees` in the search space in Figure 14 (with maximum height 3) that satisfy the invariant in Figure 10.

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`.

data structures access a node at the top of the tree more frequently than a node at the bottom of the tree. Hence we use the field ordering in Figure 19 rather than Figure 18.

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 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 Lines 5, 6, and 9 in Figure 17

We described in Section 3.2 how we use BDDs to represent our search space. Given that, the operations in Lines 5, 6, and 9 of Figure 17 can be performed efficiently. 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 Lines 7 and 8 in Figure 17

Section 2 illustrated the basic idea behind glass box checking. Section 3 so far described various pieces that are necessary to make glass box checking practical. This section, finally, describes the main glass box checking technique.

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 Pruning the Search Space

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 F_r 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 can potentially be pruned from the search space. However, we need some additional mechanisms to ensure that the system is sound. So 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 there is an invariant between a field that is modified (`x`) and a field that is a don't care (`y`).

To soundly prune the search space, our system therefore tracks (conservatively) the sets of fields that may be connected by invariants. It works as follows. If a data structure invariant only specifies that the tree backbone of the data structure must remain a tree (that is, both `repOk` and `repOkLocal` always return true), then the above technique is actually sound. The above technique thus works correctly on the `Stack` example in Figure 8.

If a data structure invariant only specifies local invariants in addition to the tree backbone (that is, `repOk` always return

true), our system knows that the only sets of fields that may be connected by an invariant are those that only include the fields of a node and the fields of its tree-children. Say, a transition `t` reads fields F_r and modifies fields F_m, and $S_{1..k}$ are sets of fields connected by invariants. Our system then computes the smallest set $F_{r'}$ such that: (i) if $f \in F_r$ then $f \in F_{r'}$, and (ii) if $f_1 \in F_r \cap F_m$ and $f_1, f_2 \in S_i$ for any $1 \leq i \leq k$ then $f_2 \in F_{r'}$. Our system treats any field not in $F_{r'}$ as a don't care, and prunes all transitions `t'` that differ from `t` only at the don't care fields.

If a data structure invariant specifies global invariants (that is, `repOk` does not always return true), then our system conservatively assumes that all the fields read by `repOk` are connected by an invariant. (This is another reason, in addition to the one at the end of Section 3.2, why using local invariants as much as possible is advantageous.) Our system includes the set S_{k+1} of all the fields read by `repOk` in the sets of fields connected by invariants, and computes the set $F_{r'}$ and uses it to prune the search space as described above. In the future, we plan to use static analysis to more precisely identify the sets of fields connected by invariants specified by a `repOk`.

3.3.2.2 Improving Precision By Tracking Information Flow

This section describes an optimization to more precisely compute the set F_r. The above algorithm sets F_r to the set of fields read by a method. But sometimes, even though a method reads a field, it does not *depend* on it. 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 to compute F_r. 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 an information flow to v. Given an execution trace of a method `m`, we set F_r to the set of fields from which there is information flow to the value returned by `m` and to all the values written by `m`.

One thing we must be careful about is that information flow analysis [11, 37] is different from dynamic slicing [27], as the following example shows.

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

```
 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.

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 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 always returns false. To avoid that, our analysis conservatively assumes that after any join point in the control flow graph, all variables depend on the corresponding branch conditional. Thus the return value `x` depends on `b`. However, in the following example, `q` on Line 2 does not depend on `p` because the branch on Line 1 always exits from the method, so there is no join point.

```
1    if (p) return true;
2    if (q) return true;
```

3.3.2.3 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: (`head`=`N0` $\wedge$ `N0.next`=`N1` $\wedge$ `N0.value`$\neq$`null`)

In general, to construct the formula, our isomorphism analysis traverses all the relevant fields of a transition 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 for-

		Transitions	BDD Nodes Created				Time (seconds)		
Benchmark	Max Size		Total	Initialization	Checking	Max BDD Size	Total	Initialization	Checking
Stack	1	4	9	4	5	2	0.470	0.468	0.002
	2	5	16	6	10	3	0.469	0.468	0.001
	3	5	18	8	10	3	0.469	0.467	0.002
	4	5	20	10	10	3	0.471	0.469	0.002
	5	5	22	12	10	3	0.470	0.469	0.001
	6	5	24	14	10	3	0.470	0.468	0.002
	7	5	26	16	10	3	0.488	0.487	0.001
	8	5	28	18	10	3	0.469	0.468	0.001
	…								
	16	5	44	34	10	3	0.474	0.472	0.002
	32	5	76	66	10	3	0.475	0.473	0.002
	64	5	140	130	10	3	0.477	0.475	0.002
	128	5	268	258	10	3	0.480	0.477	0.003
Queue	1	5	17	6	11	4	0.476	0.474	0.002
	2	7	34	10	24	8	0.475	0.473	0.002
	3	8	48	14	34	10	0.476	0.474	0.002
	4	9	57	18	39	10	0.477	0.475	0.002
	5	10	74	22	52	10	0.476	0.473	0.003
	6	11	81	26	55	10	0.476	0.473	0.003
	7	12	113	30	83	10	0.476	0.473	0.003
	8	13	122	34	88	10	0.477	0.473	0.004
	…								
	16	21	291	66	225	18	0.491	0.481	0.010
	32	37	934	130	804	34	0.546	0.513	0.033
	64	69	3211	258	2953	66	0.565	0.490	0.075
	128	133	11448	514	10934	130	0.588	0.496	0.092
HeapArray	1	4	4	2	0	1	0.490	0.488	0.002
	2	8	57	14	29	10	0.487	0.484	0.003
	3	13	262	70	122	37	0.492	0.487	0.005
	4	22	607	74	459	63	0.497	0.491	0.006
	5	32	2200	348	1504	190	0.495	0.487	0.008
	6	45	4653	595	3463	360	0.501	0.488	0.013
	7	59	9000	957	7086	759	0.512	0.486	0.026
	8	79	15416	670	14076	2143	0.522	0.486	0.036
	9	100	46695	3090	40515	5509	0.568	0.495	0.073
	10	124	71829	3961	63907	7058	0.583	0.489	0.094
	11	149	127326	5385	116556	10687	0.650	0.491	0.159
	12	176	231491	7297	216897	15036	0.755	0.492	0.263
	13	204	487375	10405	466565	28593	1.106	0.492	0.614
	14	235	1216192	17504	1181184	107758	2.216	0.496	1.720
	Max Height								
RedBlackTree	1	14	107	19	88	9	0.478	0.474	0.004
	2	48	667	155	512	51	0.489	0.474	0.015
	3	156	3973	576	3397	249	0.537	0.476	0.061
	4	478	31868	2558	29310	983	0.628	0.477	0.151
	5	1350	190363	7642	182721	3492	0.930	0.485	0.445
	6	3544	1008747	22832	985915	12066	2.679	0.496	2.183
	7	8804	6618185	68458	6549727	39339	16.272	0.541	15.731
	Height/ Degree								
FileSystem	2 2	62	630	58	572	21	0.495	0.481	0.014
	3 2	135	1431	42	1389	14	0.523	0.483	0.040
	4 2	240	3194	131	3063	41	0.542	0.481	0.061
	5 2	380	5666	162	5504	40	0.560	0.480	0.080
	6 2	558	8721	218	8503	56	0.574	0.481	0.093
	7 2	777	11710	87	11623	18	0.587	0.482	0.105
	8 2	1040	19723	350	19373	86	0.626	0.483	0.143
	9 2	1350	28322	392	27930	79	0.636	0.482	0.154
	10 2	1710	37544	509	37035	105	0.670	0.483	0.187
	2 3	102	987	152	835	31	0.514	0.482	0.032
	3 3	270	2469	108	2361	20	0.541	0.482	0.059
	4 3	560	6273	1035	5238	93	0.578	0.481	0.097
	5 3	1005	11430	1587	9843	95	0.626	0.484	0.142
	6 3	1638	19967	3650	16317	170	0.707	0.487	0.220
	7 3	2492	24288	496	23792	27	0.745	0.497	0.248
	8 3	3600	49240	12226	37014	348	1.063	0.504	0.559
	9 3	4995	65091	14698	50393	329	1.404	0.518	0.886
	10 3	6710	95751	27351	68400	511	1.993	0.527	1.466
	2 4	182	1767	442	1325	49	0.530	0.483	0.047
	3 4	675	5209	273	4936	21	0.579	0.484	0.095
	4 4	1840	25309	12443	12866	289	0.816	0.497	0.319
	5 4	4130	56441	27500	28941	349	1.458	0.527	0.931
	2 5	342	3691	1420	2271	83	0.555	0.484	0.071
	3 5	1890	13186	765	12421	26	0.717	0.499	0.218

Figure 23: Experimental results for glass box model checking.

mula using efficient BDD operations. The above technique is sound if the analysis traverses the fields in a fixed order.

Note that some black box checkers also prune isomorphs using heap canonicalization [24, 34]. 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 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. For example, 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 Lines 2, 3, and 4 in Figure 17

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. It works as follows. First, our system 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 constraints `I` with the local constraints specified by `repOkLocal` (Lines 6-14 in Figure 22). Finally, it further constraints `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 3.3.2.2) and we use BDDs to represent the search space which leads to better pruning. 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, prunes away a large number of states and operations on states without explicitly checking them (as described in the previous sections). 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 [38] 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 3.1.4). We used JavaBDD [45] for BDDs, which is built on top of the BuDDy package [29]. 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` [9], an array based implementation of a binary heap to represent a priority queue, with methods *insert* and *extractMin*; (d) `RedBlackTree` [9], from java.util.TreeMap, with methods *get*, *put*, and *remove*; and (e) `FileSystem`, adopted from the Daisy file system benchmark [12], with methods *lookup*, *create*, *unlink*, *mkdir*, and *rmdir*.

We checked each benchmark on states up to a maximum size, where: a Stack of maximum size n has at most n nodes and n values; a Queue of maximum size n has at most n nodes in the `front` Stack, n nodes in the `back` Stack, and n values; a HeapArray of maximum size n has at most n nodes and n values; a RedBlackTree of maximum size h has height at most h; and a FileSystem of maximum size (h, d) has height at most h and degree at most d, that is, 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 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 number of nodes in the tree and $h = \log n$ is the maximum height of the tree). However, for the `HeapArray`, the search space management overhead dominates the cost. We are currently exploring combining symbolic model checking with our glass box approach, by augmenting our search space with symbolic values, to reduce the search space management overhead.

Figure 24 presents results of comparing the performance of our glass box model checker with JPF [44]. For JPF, we report the number of transitions explicitly checked by the checker, as well as the number of unique states visited by the checker (as a measure of its space overhead, since JPF

		Glassbox			JPF		
Benchmark	Max Size	Transitions	BDD Nodes	Time (s)	Transitions	States	Time (s)
`Stack`	1	4	9	0.470	22	12	0.765
	2	5	16	0.469	170	66	0.766
	3	5	18	0.469	3864	1290	2.686
	4	5	20	0.471	335174	99368	174.0
	5	5	22	0.470			timeout
	...						
	8	5	28	0.469			timeout
	16	5	44	0.474			timeout
	32	5	76	0.475			timeout
	64	5	140	0.477			timeout
	128	5	268	0.480			timeout
`Queue`	1	5	17	0.476	24	18	0.663
	2	7	34	0.475	284	144	0.985
	3	8	48	0.476	1955	791	2.213
	4	9	57	0.477	86736	29197	60.00
	5	10	74	0.476			timeout
	...						
	8	13	122	0.477			timeout
	16	21	291	0.491			timeout
	32	37	934	0.546			timeout
	64	69	3211	0.565			timeout
	128	133	11448	0.588			timeout
`HeapArray`	1	4	4	0.490	170	85	0.604
	2	8	57	0.487	1558	627	1.983
	3	13	262	0.492	28826	9705	24.218
	4	22	607	0.497	795606	229455	900.0
	5	32	2200	0.495			timeout
	6	45	4653	0.501			timeout
	7	59	9000	0.512			timeout
	8	79	15416	0.522			timeout
	9	100	46695	0.568			timeout
	10	124	71829	0.583			timeout
	11	149	127326	0.650			timeout
	12	176	231491	0.755			timeout
	13	204	487375	1.106			timeout
	14	235	1216192	2.216			timeout
	Max Height						
`RedBlackTree`	1	14	107	0.478	47	29	0.752
	2	48	667	0.489	7585	3306	5.000
	3	156	3973	0.537			timeout
	4	478	31868	0.628			timeout
	5	1350	190363	0.930			timeout
	6	3544	1008747	2.679			timeout
	7	8804	6618185	16.272			timeout

Figure 24: Comparing glass box model checking to JPF.

		Glassbox			Blackbox		
Benchmark	Max Size	Transitions	BDD Nodes	Time (s)	Transitions	States	Time (s)
Stack	1	4	9	0.470	1	1	0.001
	2	5	16	0.469	9	3	0.001
	3	5	18	0.469	32	8	0.001
	4	5	20	0.471	115	23	0.007
	5	5	22	0.470	450	75	0.013
	6	5	24	0.470	1946	278	0.023
	7	5	26	0.488	9240	1115	0.066
	8	5	28	0.469	47655	5295	0.368
	9	5	30	0.470	264420	26442	2.333
	10	5	32	0.474	1566587	142417	17.587
	...						
	16	5	44	0.474			timeout
	32	5	76	0.475			timeout
	64	5	140	0.477			timeout
	128	5	268	0.480			timeout
Queue	1	5	17	0.476	15	5	0.001
	2	7	34	0.475	132	33	0.013
	3	8	48	0.476	1815	363	0.093
	4	9	57	0.477	38838	6473	0.517
	5	10	74	0.476	1175041	167873	18.609
	6	11	81	0.476			timeout
	7	12	113	0.476			timeout
	8	13	122	0.477			timeout
	...						
	16	21	291	0.491			timeout
	32	37	934	0.546			timeout
	64	69	3211	0.565			timeout
	128	133	11448	0.588			timeout
HeapArray	1	4	4	0.490	6	2	0.001
	2	8	57	0.487	24	6	0.001
	3	13	262	0.492	120	24	0.004
	4	22	607	0.497	660	110	0.034
	5	32	2200	0.495	4648	664	0.096
	6	45	4653	0.501	36120	4515	0.330
	7	59	9000	0.512	375264	41696	2.690
	8	79	15416	0.522	3445710	344571	29.973
	9	100	46695	0.568			timeout
	10	124	71829	0.583			timeout
	11	149	127326	0.650			timeout
	12	176	231491	0.755			timeout
	13	204	487375	1.106			timeout
	14	235	1216192	2.216			timeout
	Max Height						
RedBlackTree	1	14	107	0.478	6	2	0.024
	2	48	667	0.489	315	21	0.065
	3	156	3973	0.537	3713787	58949	81.395
	4	478	31868	0.628			timeout
	5	1350	190363	0.930			timeout
	6	3544	1008747	2.679			timeout
	7	8804	6618185	16.272			timeout

Figure 25: Comparing glass box model checking to an optimal black box model checking.

		Glassbox			Korat		
Benchmark	Max Size	Transitions	BDD Nodes	Time (s)	States Considered	Transitions	Time (s)
`Stack`	1	4	9	0.470	7	6	0.001
	2	5	16	0.469	12	9	0.001
	3	5	18	0.469	18	12	0.001
	4	5	20	0.471	25	15	0.001
	5	5	22	0.470	33	18	0.001
	6	5	24	0.470	42	21	0.002
	7	5	26	0.488	52	24	0.002
	8	5	28	0.469	63	27	0.002
	9	5	30	0.470	75	30	0.002
	10	5	32	0.474	88	33	0.002
	...						
	16	5	44	0.474	187	51	0.008
	32	5	76	0.475	627	99	0.019
	64	5	140	0.477	2275	195	0.054
	128	5	268	0.480	8643	387	0.258
`Queue`	1	5	17	0.476	20	15	0.001
	2	7	34	0.475	43	27	0.001
	3	8	48	0.476	77	42	0.002
	4	9	57	0.477	124	60	0.003
	5	10	74	0.476	186	81	0.006
	6	11	81	0.476	265	105	0.012
	7	12	113	0.476	363	132	0.015
	8	13	122	0.477	482	162	0.018
	...						
	16	21	291	0.491	2430	510	0.041
	32	37	934	0.546	14838	1782	0.195
	64	69	3211	0.565	102374	6630	1.805
	128	133	11448	0.588	757702	25542	26.162
`HeapArray`	1	4	4	0.490	13	12	0.001
	2	8	57	0.487	66	60	0.001
	3	13	262	0.492	364	330	0.004
	4	22	607	0.497	2185	1920	0.013
	5	32	2200	0.495	15250	13433	0.076
	6	45	4653	0.501	119658	105112	0.267
	7	59	9000	0.512	1168018	1058058	2.286
	8	79	15416	0.522	11370780	10050750	24.501
	9	100	46695	0.568			timeout
	10	124	71829	0.583			timeout
	11	149	127326	0.650			timeout
	12	176	231491	0.755			timeout
	13	204	487375	1.106			timeout
	14	235	1216192	2.216			timeout

Figure 26: Comparing glass box model checking to Korat.

is a stateful checker). The results clearly indicate how JPF takes exponentially more time as the size of the data structures increase. Our glass box checker, however, scales much better.

While running experiments with JPF, we noticed that JPF sometimes does not detect that two states are isomorphic. It therefore visits a lot more states than necessary. To make for a fairer comparison, we implemented our own black box checker. We implemented all the optimizations in published literature that we are aware of. We also handcoded a isomorphism detector for each data structure we tested, to simulate an optimal black box checker. Figure 25 presents results of comparing the performance of our glass box model checker with our optimal black box model checker. Once again, the results clearly indicate that glass box model checking scales significantly better than black box model checking.

Finally, Figure 26 presents the results of comparing the performance of our glass box model checker with Korat [2]. Korat ultimately works like a black box checker, so the results for Korat are similar to other black box checkers. Our glass box checker performs much better.

5 Related Work

There has been much research on software 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 [16] is a stateless model checker for C programs. Java PathFinder (JPF) [44, 26] is a stateful model checker for Java programs. XRT [20] checks Microsoft CIL programs. Bandera [8] and JCAT [10] translate Java programs into the input language of model checkers like SPIN [22] and SMV [31]. Bogor [13] provides an extensible framework for building software model checkers. CMC [36] 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 [35]. However, most of the above work on applying model checking to software focuses on scheduling nondeterminism to verify event sequences with respect to temporal properties. This paper deals with input nondeterminism. In particular, it focuses on verifying properties of linked data structures.

The main contribution of this paper is as follows. Consider checking a red-black tree implementation. Previous model checking approaches such as JPF [44, 26], CMC [36, 35], Korat [2], and Alloy [25] systematically generate all red-black trees up to a given size n and check every red-black tree operation on every red-black tree. Since the number of red-black trees is exponential in n, these checkers take exponential time. On the other hand, our checker detects similarities in the search space and infers that it is sufficient to check every red-black tree operation on every red-black tree path. Since the number of red-black tree paths is polynomial in n, our checker takes polynomial time. This leads to orders of magnitude speedups over the previous approaches. This paper also introduces several techniques to make glass box checking work well in practice.

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. The abstraction-based tools group several concrete program states into an abstract state and check the abstract state instead of checking several concrete states. Our glass box checker also in effect groups concrete states by using program analyses to identify states on which a given operation behaves similarly, and checks the operation on only one state from each group. One difference is that the abstraction-based tools use heuristics to group states, whereas our system groups states only if they are found to be similar with respect to an operation. However, the two techniques are complementary and can be combined.

There are many static [16, 17] and dynamic [14] partial order reduction systems. These systems are designed to handle scheduling nondeterminism and use techniques that are quite different from our techniques for checking data structure properties. In particular, the dynamic partial order reduction [14] works only in a stateless checker. Our checker is stateful in that it does not visit a state more than once.

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, 41], UML statecharts [39], or ADL specifications [6]. These specifications typically do not consider linked data structures, and the tools do not generate Java test cases.

For systematically generating states from invariants we previously developed a system called Korat [2]. The main difference between Korat and our checker is that Korat works like a black box checker. That is, generates every valid state (within a bounded domain) and checks every operation on every state. Our checker, on the other hand, prunes away a large number of states and operations on states without explicitly checking them. Figure 26 provides an experimental comparision of the two approaches. Also, Section 3.3.3 discusses other differences between Korat and this paper. [43] translates a program and its specifications into a SAT formula and uses a constraint solver to check the program. However, this approach does not seem to scale well probably because it generates huge formulas. Moreover, it introduces additional unsoundness by bounding the lengths of computations (e.g., 3 unrollings of loops).

ESC/Java [15] uses a theorem prover to verify absence of such errors as null pointer dereferences and array bounds violations. JVer [5] uses a theorem prover to verify resource bounds of applets. Static analyses such as TVLA [40] and PALE [33] offer a promising approach for verifying 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. Exhaustive testing, 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. While most previous work on software model checking focuses on scheduling nondeterminism to verify event sequences with respect to properties expressed in temporal logics, this paper focuses on verifying properties of linked data structures. The paper presents novel techniques for detecting similarities in the search space of data structures, and for soundly pruning 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 software model checker that scales nearly as well for checking linked data structures. This paper also presents several techniques that make glass box model checking work well in practice. 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 are grateful to Pratibha Permandla for helping us run some of the experiments. We also thank Sarfraz Khurshid, Darko Marinov, Madan Musuvathi, and Alexandru Salcianu for 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] A. Chander, D. Espinosa, N. Islam, P. Lee, and G. C. Necula. JVer: A Java verifier. In *Computer Aided Verification (CAV)*, January 2005.

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

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

[8] 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.

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

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

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

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

[13] M. B. 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.

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

[15] 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.

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

[17] 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.

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

[26] 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.

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

[28] 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.

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

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

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

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

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

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

[35] 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.

[36] 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.

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

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

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

[40] 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.

[41] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International, 2nd edition, 1992.

[42] 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.

[43] 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.

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

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