Language and Analysis Techniques for 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 a combination of dynamic and static analyses 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 checking of data structure properties significantly faster, and thus enable checking of much larger programs than currently possible.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Program Verification;
D.2.5 [**Software Engineering**]: Testing and Debugging;
D.3.4 [**Programming Languages**]: Processors;
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, 5, 8, 9, 13, 16, 39, 19, 32] 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 [10] implementation maintains the red-black tree invariants. Previous model checking approaches such as JPF [39, 23], CMC [32, 31], Korat [2], or Alloy [22] 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 dynamic and static 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

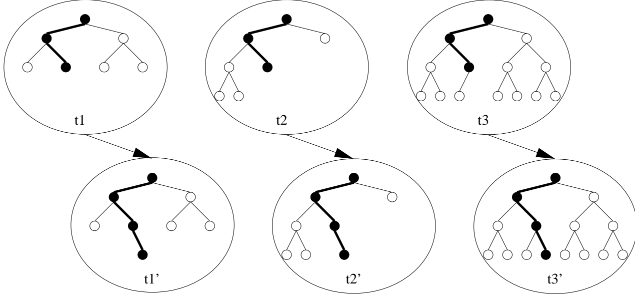

Figure 1: Three red-black trees before and after an `insert` operation. The tree path touched by the operation is highlighted in each case. Once our glass box checker checks the `insert` operation on tree `t1`, it determines that it is redundant to check the same operation on `t2` and `t3`.

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 [39, 23], CMC [32, 31], Korat [2], or Alloy [22]) 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   public boolean repOk() {
5     return (back != null) && back.repOk() && (back != front)
6         && (front != null) && front.repOk();
7   }
8
9   //              ------------     ------------
10  // dequeue <---    front    |   |    back     <--- enqueue
11  //              ------------     ------------
12
13  public void enqueue(Object o) {
14    back.push(o);
15  }
16  public Object dequeue() throws EmptyQueueException {
17    if (front.isEmpty()) moveBackToFront();
18    if (front.isEmpty()) throw new EmptyQueueException();
19    return front.pop();
20  }
21  private void moveBackToFront() {
22    assert(front.isEmpty());
23    back.reverse();  front = back;  back = new Stack();
24  }
25 }
```

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

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 5.)

`Queue`'s class invariant is described by its `repOk` method, as good programming practice suggests [26]. 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

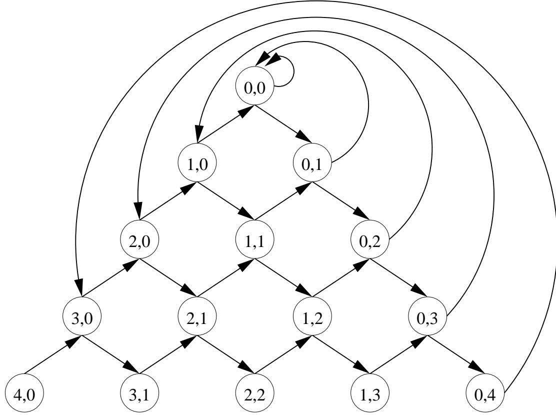

Figure 3: State space of `Queue` with at most $n = 4$ items. State (f,b) has f items in `front Stack` and b in `back`. A black box checker checks $\Omega(n^2)$ state transitions.

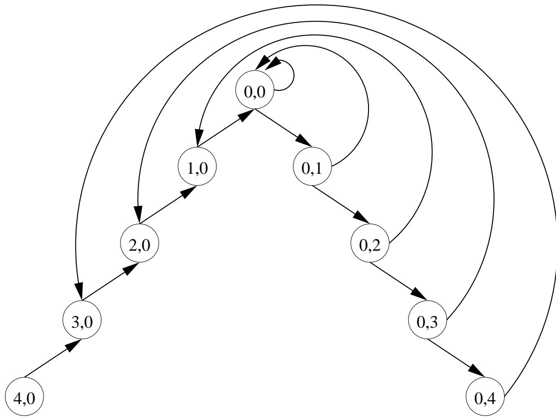

Figure 4: A glass box checker generates only $O(n)$ states and checks only $O(n)$ transitions, yet achieves complete coverage within the bounded domain.

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 [39] or CMC [32, 31] 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  public class Stack {
2    private static class Node<nOwner> {
3      Node<nOwner> next;
4      Object       value;
5      Node(Node<nOwner> n, Object v) { next = n; value = v; }
6    }
7
8    private Node<this> head;
9    public boolean repOk() {
10     Set  visited = new java.util.HashSet();
11     for (Node n = head; n != null; n = n.next) {
12       if (!visited.add(n)) return false;
13     }
14     return true;
15   }
16
17   public void push(Object value) {
18     head = new Node(head,value);
19   }
20   public Object pop() {
21     if (head == null) return null;
22     Object v = head.value;  head = head.next; return v;
23   }
24 }
```

Figure 5: Stack implemented using a linked list

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 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.

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

Figure 6: A class with three boolean variables `x, y, z`

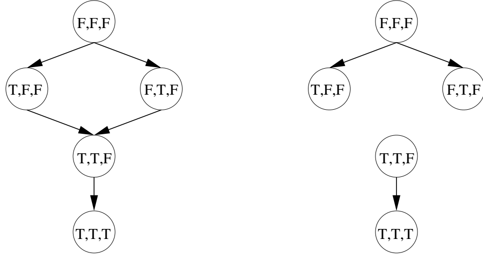

Figure 7: State space of code in Figure 6 (excluding self loops). (b_1,b_2,b_3) implies $\mathtt{x} = b_1$, $\mathtt{y} = b_2$, $\mathtt{z} = b_3$. The figures on the left and right show the state transitions executed by a black box and glass box checker respectively.

3 Glass Box Model Checker

This section presents our glass box model checker. Figure 5 shows a `Stack` that is implemented using a linked list. Its class invariant (`repOk`) checks that the list is acyclic. We use this as a running example. and show how we check that the `Stack` implementation preserves the `Stack` invariant.

3.1 Search

This section describes our basic search process.

3.1.1 Generating States from Invariants

Consider our running example `Stack` from Figure 5. 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 [39] or CMC [32] 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 6 illustrates why. Figure 7 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).

```
1  public Finitization checkStack(int nNodes, int nValues) {
2
3      Finitization f = new Finitization("Stack");
4
5      Set nodes  = f.createObjects("Node",   nNodes  - 1);
6      Set values = f.createObjects("Object", nValues - 1);
7      nodes.add(null);
8      values.add(null);
9
10     f.setFieldDomain("head",       nodes);
11     f.setFieldDomain("Node.next",  nodes);
12     f.setFieldDomain("Node.value", values);
13     f.setOperations("push", "pop");
14     f.setArgumentDomain("push", "value", values);
15
16     return f;
17 }
```

Figure 8: Finitization description for code in Figure 5

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

Figure 9: Search space for `checkStack(4,4)`

Instead, our glass box checker uses a different approach. In previous work on Korat [2], we developed a technique to systematically generate all the states (within a bounded domain) that satisfy a given class invariant. Our glass box checker uses a similar approach, and similarly relies on class invariants to cover every state. The main difference between Korat and a glass box checker is that Korat ultimately works like a black box 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 illustrated in Section 2). We also makes several improvements to the Korat technique itself, which we present later in this paper.

3.1.2 Finitization

In any model checker that checks data structure properties, programmers must specify finite bounds on the state space. In our glass box checker, programmers specify the maximum number of objects of each class, and the domain of every field and every method argument. Our checker then checks the program on every possible state in this finite space.

Figure 8 presents an example *finitization* description that is automatically generated by our system from the type declarations in Figure 5. The `createObjects` methods specify that a state can contain at most (`nNodes`-1) number of `Nodes` and (`nValues`-1) number of `Objects`. The `setFieldDomain` and `setArgumentDomain` methods specify that the fields `head` and `next` can either contain `null` or a `Node`, and the field `value` and the argument to `push` can either contain `null` or an `Object`. The `setOperations` method specifies that the checker must check the two public methods of `Stack`.

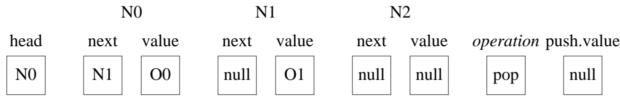

Figure 10: A valid element of the search space representing the pop operation on a Stack with two items O1 an O2

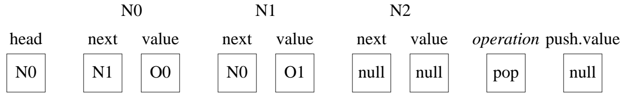

Figure 11: An invalid element of the search space with a cycle in the linked list

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

3.1.3 Search Space

Suppose our checker is invoked using `checkStack(4,4)` in Figure 8. Our system then constructs the search space in Figure 9. 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 10 corresponds to invoking `pop` on a `Stack` with two items `O0` and `O1`. In Figure 9, there are eight fields with four elements in their domains and one with two, so the size of this search space in $2 * 4^8$. In general, when our checker is invoked with `checkStack(n,n)`, the size of the search space is $2 * (n)^{2n}$. Note that many elements of this space are invalid because the corresponding `Stack` objects do not satisfy the class invariant. For example, the element in Figure 11 is invalid because the linked list has a cycle.

3.1.4 Search

Figure 12 presents the basic glass box search algorithm. Given a class to check and a finitization, our system first initializes the search space as we described in Section 3.1.3. It then systematically explores this space by repeatedly selecting a transition `t` from the space, checking `t`, running its analyses to identify other transitions similar to `t`, and pruning `t` and the similar transitions from the space.

3.1.5 Search Space Representation

Consider checking the `Stack` with `checkStack(n,n)`. Our checker generates $O(1)$ states and checks $O(1)$ transitions to cover this search space (as we show later). But the size of the search space is $2 * (n)^{2n}$. If we are not careful, then search space management itself could take exponential time. We avoid this by compactly representing the search space using *reduced ordered binary decision diagrams* [4], or BDDs. We use a BDD to represent the `Set searchSpace` in Figure 12. For the `Stack` example, the set initially contains all the $2 * (n)^{2n}$ elements in the search space. The size of the BDD representing this set is, however, $O(1)$. As the search proceeds, we progressively prune elements from this set by applying BDD operations. In theory, the size of the BDD could grow exponentially during this search process. In practice, we found that the size of the BDD usually remains small. For example, for `Stack`, experiments indicate that the BDD only grows to a maximum size of $O(\log n)$.

```
1  void search(Finitization f) {
2    Set searchSpace = GlassBoxChecker.initialize(f);
3    while (!searchSpace.isEmpty()) {
4      Transition t = searchSpace.getAnUncheckedTransition();
5      Set checkedTransitions = GlassBoxChecker.check(t);
6      searchSpace.prune(checkedTransitions);
7  }}
```

Figure 12: Pseudo-code for the search algorithm

The key to keeping the BDD size small is a good field ordering. We use a simple heuristic—fields that are read together by the code are kept together in the BDD. This seems to naturally induce a good field ordering and thus compact BDDs.

3.2 Dynamic Analysis

This section presents our dynamic analysis techniques that are key to making our glass box checker efficient.

3.2.1 Monitoring Fields Read

Consider the `Stack` example again. Given a state transition, our checker first checks if the precondition (`repOk`) returns true. If so, our checker runs the corresponding method, and then checks if the postcondition (`repOk`) returns true.

Case Where Precondition is False

Consider checking the state transition in Figure 11. As our checker runs `repOk`, it monitors the input fields that `repOk` reads. In this case, `repOk` reads `head`, `N0.next`, and `N1.next` and returns false (because it detects a cycle in the linked list). That means, no matter what the values of the remaining fields are, `repOk` will always return false (assuming deterministic execution). Our system therefore prunes all elements of the search space where `head=N0`, `N0.next=N1`, and `N1.next=N0`. This idea is similar to how we generated all inputs that satisfied a given precondition in Korat [2].

Case Where Precondition is True

Consider checking the state transition in Figure 10. The precondition is true this time. As our checker then runs the `pop` method, it once again monitors the input 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 static analysis then determines that regardless of the values of the remaining fields, if the precondition (`repOk`) holds before `pop`, then the postcondition (`repOk`) 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`.

This is the main idea that makes glass box checking fast. Note that regardless of the maximum length of the linked list, our checker verifies `pop` by running it on a constant number of states (assuming it also prunes isomorphic structures as we describe in Section 3.2.3). On the other hand, Korat (and every black box checker) runs `pop` on every valid (and nonisomorphic) state to verify it.

3.2.2 Monitoring Information Flow

The above analysis conservatively assumes that a `repOk` (or a method) depends on all the fields it reads. This assumption is usually true for naturally written programs, but not always. The `Rational` class below provides an example. Suppose `repOk` returns false on Line 5 because `d==0`. The above analysis assumes that because `repOk` read both `numerator` and `denominator`, the return value depends on both the fields—even though it depends only on `denominator`.

```
1  class Rational {
2    private int numerator, denominator;
3    public boolean repOk() {
4      int n=abs(numerator), d=denominator;
5      if (d <= 0) return false;
6      if (n>0 && gcd(n,d)>1) return false;
7      return true;
8  }}
```

To make our analysis more precise, we use dynamic information flow tracking. Consider `Stack` in Figure 9. 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. Note that information flow analysis [12, 33] is different from dynamic slicing [24], as the following example shows.

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

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 `repOk` with `b=false` we would incorrectly conclude that `repOk` 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;
```

For a `repOk`, our analysis tracks flow to the value returned by `repOk`. For a method `m`, our analysis tracks flow to the value returned by `m`, to all the values written by `m`, and to the predicates of all branches in the execution trace of `m`.

3.2.3 Pruning Isomorphic Structures

Compare the `Stack` in Figure 10 with a `Stack` where `head=N1`, `N1.next=N2`, `N1.value=O0`, `N2.next=null`, and `N2.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 10, our dynamic analysis concludes 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 all structures that satisfy the following formula can also be pruned:

$(\texttt{head}{\neq}\texttt{N0/null}) \vee (\texttt{head}{=}\texttt{N0} \wedge \texttt{N0.next}{\neq}\texttt{N1/null}) \vee (\texttt{head}{=}\texttt{N0} \wedge \texttt{N0.next}{=}\texttt{N1} \wedge \texttt{N0.value}{\neq}\texttt{O0/null})$

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 formula 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 [21, 30]. 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 Language Mechanisms

This section describes languages mechanisms. These are *not* necessary for glass box checking, but facilitate the process.

3.3.1 Ownership Types

Consider the `Queue` example in Figure 2. Suppose the `Stacks` are implemented using linked lists. Then the invariant of `Queue` must state that the two `Stacks` do not share list nodes. It is difficult to express such an invariant without exposing the representation of `Stack`. The problem gets worse if there are several implementations of `Stack`. Our system allows programmers to express such invariants elegantly using ownership types [3, 7]. Consider the `Stack` in Figure 5 for an example. Line 8 declares that the `Stack` *owns* the `head` node. Line 5 recursively declares that the `next` node in the list has the same owner as the `this` node. These declarations imply that all the nodes in a `Stack` are private to the `Stack`. Two `Stacks` therefore cannot share list nodes. Our finitization process (see Section 3.1.2) then exploits these ownership declarations to automatically generate a finitization description for `Queue` where the domains of `front` and `back` and the `next` fields of the nodes they point to do not overlap. Ownership types thus enable convenient expression of invariants and efficient generation of structures from invariants.

3.3.2 Special Asserts

Consider the code below from `repOk` of a doubly linked list:

```
if (this.next.prev != this) return false;
```

Suppose the domain of `prev` is $\mathtt{N}_0..\mathtt{N}_n$. Our system tries all possible values of `prev`, even though there is only one value for which `repOk` does not return false. Instead, programmers can write the above code as follows:

```
assertEqual(this.next.prev, this);
```

Our preprocessor then translates the above line into code that provides feedback to our checker. When `repOk` returns false because of the above line, our checker knows that it is unnecessary to try out all other values for `this.next.prev` except `this`. It thus avoids checking many states. Our system provides several such asserts. E.g., `assertGreater` and `assertGreaterOrEqual` are applicable to domains containing `Comparable` items, and are translated into efficient BDD operations. `assertTree` makes it convenient to specify the tree backbone of tree-based data structures; e.g., `repOk` of `Stack` can be simply written as `assertTree(head,"next*")`, and a binary tree as `assertTree(root,"{left+right}*")`. `assertTree` also speeds up the generation of tree-based data structures from invariants by avoiding non-tree states. A later version of Korat [27] also provides similar asserts.

3.4 Static Analysis

The dynamic analyses in Section 3.2 detect *don't care* fields in a transition t, and suggest that all transitions t' that differ with t only at the don't care fields can potentially be pruned from the search space. The goal of the static analysis is to prove that it is indeed safe to prune those transitions. To see why static analysis is necessary, consider the following:

```
1  class StaticAnalysisDemo {
2    private boolean a, b;
3    public  boolean repOk() { return !a || b; }
4    public  void    flipA() { a = !a;         }
5  }
```

`repOk` returns true iff `a` implies `b`. Suppose we invoke `flipA` on `a=false` and `b=true`. The pre and postconditions (both `repOk`) hold. `flipA` reads only `a`; `b` is a don't care. Our dynamic analysis suggests that `flipA` will perhaps verify on all states where `a=false` (and therefore those elements be pruned from the search space). But the suggestion is incorrect because `flipA` does not verify on `a=false` and `b=false`. The precondition holds then but not the postcondition.

Our static analysis works as follows. Consider checking an operation o on a state s. Suppose our dynamic analysis identifies fields $f_{1..k}$ as don't cares. Let v denote the values of the remaining fields in s, and v' in the state after executing o. The static analysis partially evaluates the pre and postconditions of o, say `pre` and `post`, with respect to v and v' respectively, to get functions $\mathtt{pre}_v(f_{1..k})$ and $\mathtt{post}_{v'}(f_{1..k})$. The analysis then attempts to prove that for all values of $f_{1..k}$ in the bounded domain, $\mathtt{pre}_v(f_{1..k})$ implies $\mathtt{post}_{v'}(f_{1..k})$. (Note that even if `pre` and `post` are the same function, v and v' could be different if the method performs mutations. So $\mathtt{pre}_v$ and $\mathtt{post}_{v'}$ could be different.)

Recall the `Queue` from Section 2.2 for an example. Consider the `enqueue` transition from (0,0) to (0,1) in Figure 4. Our dynamic analysis identifies `front` as a don't care. Our static analysis then performs a partial evaluation of the pre and postconditions (both `repOk`) with respect to `back`, as we show informally below assuming `front=`F and `back=`B before the operation and `back=`B' after the operation.

$\mathtt{pre}_B(F)$

$= \mathtt{Queue}\{\mathtt{front} = F, \mathtt{back} = B\}.\mathtt{repOk}()$

$= F \neq \mathtt{null} \wedge F.\mathtt{repOk}() \wedge B \neq \mathtt{null} \wedge B.\mathtt{repOk}() \wedge F \neq B$

$= F \neq \mathtt{null} \wedge F.\mathtt{repOk}() \wedge F \neq B$

because B is a constant, so any constraints on B can be statically determined to be true

$\mathtt{post}_{B'}(F)$

$= \mathtt{Queue}\{\mathtt{front} = F, \mathtt{back} = B'\}.\mathtt{repOk}()$

$= F \neq \mathtt{null} \wedge F.\mathtt{repOk}() \wedge B' \neq \mathtt{null} \wedge B'.\mathtt{repOk}() \wedge F \neq B'$

$= F \neq \mathtt{null} \wedge F.\mathtt{repOk}() \wedge F \neq B'$

because B' is a constant

$= F \neq \mathtt{null} \wedge F.\mathtt{repOk}() \wedge F \neq B$

because B and B' are the same Java object, even though their states are different

$\therefore \forall F.\ (\mathtt{pre}_B(F) \implies \mathtt{post}_{B'}(F))$

Note that after the partial evaluation above the pre and postconditions turn out to be identical, as they often do when the pre and postconditions are both the same class invariant. The implication holds trivially then.

However, partial evaluation of `repOks` written in object-oriented languages is non-trivial [36]. We instead use the following approach. We ask users to write class invariants in a declarative language similar to Alloy [22]. Usually, declarative specifications are more convenient to write than `repOks`. We automatically generate executable `repOks` from the declarative specifications for our dynamic analysis. We use the declarative specifications for the static analysis. We eliminate quantifiers using skolemization (as our domains are bounded) and then do the partial evaluation. We handle `assertTree` (described in Section 3.3.2) specially to improve analysis precision. The analysis is straightforward otherwise.

3.5 Programming Effort

Previous sections describe how our system verifies data structure invariants. Our system does not require any additional specifications from programmers other than the invariants themselves. The checking is therefore almost automatic.

In addition to verifying invariants, our checker also verifies any data structure properties that can be expressed as executable pre and postconditions. In that case, in addition to the specifications being checked, our system also requires class invariants from programmers. This is because, our checker depends on the invariants to explore all states. 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. Moreover, if users make a mistake in writing the invariants, our system provides concrete counter examples to help users correct the invariants.

		Black Box	Glass Box	
Benchmark	Max Size	Transitions	Transitions	BDD Size
Stack	1	9	8	8
	2	32	10	14
	3	115	10	11
	4	450	10	25
	5	1946	10	23
	6	9240	10	31
	7	47655	10	14
	8	264420	10	48
	9	1566587	10	43
	10	9851844	10	56
	15	timeout	10	17
	31	timeout	10	20
Queue	1	15	12	11
	2	132	18	38
	3	1815	24	30
	4	38838	30	97
	5	1175041	36	105
	15	timeout	96	312
	31	timeout	192	831

Figure 13: Comparing black box model checking with glass box model checking

		Black Box	Glass Box	
Benchmark	Max Size	Transitions	Transitions	BDD Size
HeapArray	1	6	6	6
	2	24	14	22
	3	120	27	31
	4	660	54	141
	5	4648	114	305
	6	36120	310	770
	7	375264	515	1592
	8	3445710	828	7706
RedBlackTree	1	24	10	15
	2	240	36	49
	3	1176	92	102
	4	15610	277	230
	5	145872	619	556
	6	2056320	1181	1203

Figure 14: Comparing black box model checking with glass box model checking

4 Experimental Results

This section presents our preliminary experimental results. We implemented both a black box model checker and a glass box model checker to study improvements obtained with glass box checking. Our black box checker starts from the initial state and recursively invokes and checks every operation on every successive state. We implemented all the relevant optimizations published in literature including stateful search and heap canonicalization. Our glass box checker works as described in this paper. We extended the Polyglot [34] compiler framework to automatically instrument programs to perform our dynamic analysis, and to automatically generate the finitization descriptions. We used JavaBDD [40] for BDDs, which is built on top of BuDDy [25].

We present results for the following benchmarks: (a) `Stack` shown in Figure 5, with methods *push* and *pop*; (b) `Queue` shown in Figure 2, implemented using the Stack in Figure 5, with methods *enqueue* and *dequeue*; (c) `HeapArray` [10], an array based implementation of a binary heap to represent a priority queue, with methods *insert* and *extractMin*, and (d) `RedBlackTree` [10], from java.util.TreeMap, with methods *get*, *put*, and *remove*;

We checked each benchmark on states up to a maximum size n, where: a Stack, a HeapArray, and a RedBlackTree of maximum size n has at most n nodes and n values; and a Queue of maximum size n has at most n nodes in the `front` Stack, n nodes in the `back` Stack, and n values.

We report the following numbers. For a black box checker, we report the number of state transitions executed by the checker. This number depends on the number of states visited by the checker and the number of transitions enabled on each state. For a glass box checker, we report the number of transitions considered by the checker. This includes both valid transitions executed by the checker (such as Figure 10) and invalid transitions considered by the checker (such as Figure 11). For a glass box checker, we also report the maximum size of the BDD as a measure of the search space management overhead. We do not report execution times because we did not yet optimize the checkers for that; e.g., for the black box checker, we implemented heap canonicalization, but not incremental heap canonicalization [30]. But to give an indication, our current glass box implementation checks `Queue` on states of size up to 31 in under 200 milliseconds, whereas our black box implementation takes about 15 seconds for checking `Queue` on states of size up to 5.

Figure 13 presents results for the Stack and Queue benchmarks. For checking the Stack, our glass box checker considers only $O(1)$ transitions regardless of the size of the Stack, because *push* and *pop* touch only a constant number of fields at the beginning of the linked list. The BDD size growth appears a bit erratic because BDDs whose fields have domain sizes that are powers of 2 tend to be smaller. However, on careful observation, one can see that the BDD size grows roughly as $O(\log n)$; e.g., observe the BDD size when the Stack size is 1,3,7,15,31. (Note that if a Stack has most n nodes, the domain of its `next` fields consists of n nodes and `null`, so the size of the domain is $n + 1$.) The $O(\log n)$ factor come in because it takes $O(\log n)$ bits to represent a domain of size n in a BDD. For checking the Queue, our glass box checker considers $O(n)$ transitions, as explained in Section 2.2. The BDD size grows roughly as $O(n \log n)$. A black box checker, on the other hand, executes an exponential number of transitions to check the Queue (and similarly the Stack), as explained at the end of Section 2.2.

Figure 14 present results for other benchmarks. The numbers demonstrate that glass box checking is significantly more efficient than black box checking.

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) [39, 23] is a stateful model checker for Java programs. XRT [18] checks Microsoft CIL programs. Bandera [9] and JCAT [11] translate Java programs into the input language of model checkers like

SPIN [20] and SMV [28]. Bogor [13] provides an extensible framework for building software model checkers. CMC [32] 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 [31]. 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 [39, 23], CMC [32, 31], Korat [2], and Alloy [22] 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. Moreover, our isomorphism pruning technique is different from heap canonicalization [21, 30] used in the previous checkers, as explained in Section 3.2.3.

Tools such a Slam [1], Blast [19], and Magic [5] use heuristics to construct and check an abstraction of a program (usually predicate abstraction [17]). 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] 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.

For systematically generating states from invariants we use an approach we developed in Korat [2]. The main difference between Korat and our checker is that Korat 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. This paper also improves on the Korat state generation technique by using dynamic information flow tracking (Section 3.2.2), special language constructs (Section 3.3), and BDDs (Section 3.1.5). In particular, Korat imposes a linear (lexicographic) 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 uses BDDs, so it can prune all the elements its analyses identify. [38] 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 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 [6] uses a theorem prover to verify resource bounds of applets. Static analyses such as TVLA [35] and PALE [29] 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 makes the following contributions.

It introduces the *glass box* approach to software model checking. Our checker detects similarities in the search space and prunes redundant states and operations without explicitly checking them. This results in significant speedups.

It shows how to compactly represent the search space of a glass box checker with BDDs. BDDs have been used for checking temporal properties of software before (e.g., in jMoped [37]). But they have not been used in this context for verifying data structure properties. The resulting state spaces and operations on the state spaces are different.

It presents an optimization to avoid checking isomorphic states. This technique is different from heap canonicalization, as explained in Section 3.2.3.

It presents dynamic and static analysis techniques and language mechanisms that make glass box checking efficient.

Finally, the paper presents preliminary experimental results to show the improvements obtained by glass box checking.

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] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *Principles of Programming Languages (POPL)*, January 2003.

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

[5] 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.

[6] A. Chander, D. Espinosa, N. Islam, P. Lee, and G. Necula. JVer: A Java verifier. In *Computer Aided Verification (CAV)*, January 2005.

[7] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1998.

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

[9] 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.

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

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

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

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

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

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

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

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

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

[23] 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.

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

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

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

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

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

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

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

[31] 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.

[32] 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.

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

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

[35] 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.

[36] U. P. Schultz, J. L. Lawall, and C. Consel. Automatic program specialization for Java. *Transactions on Programming Languages and Systems (TOPLAS) 25(4)*, July 2003.

[37] 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.

[38] 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.

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

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