Glass Box Software Model Checking

by

Michael E. Roberson

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2011

Doctoral Committee:

Assistant Professor Chandrasekhar Boyapati, Chair
Professor Karem A. Sakallah
Associate Professor Marc L. Kessler
Associate Professor Scott Mahlke

ACKNOWLEDGEMENTS

This dissertation owes a great deal to my advisor, Chandrasekhar Boyapati. Chandra's steady help throughout the years had no small part in the honing of my research, writing, and presentation skills. He introduced me to new ideas and helped me to develop my own ideas. I am fortunate to have him as an advisor.

I also had the help of excellent colleagues. Paul Darga laid the groundwork that ultimately led to this dissertation. Melanie Harries provided valuable support for this project. I look back fondly on the days when Paul, Melanie, and I would collaborate and exchange ideas.

My family and friends lavished me with encouragement and support during my time in graduate school. I was propelled forward by the support of my wonderful wife Anne, my mother Mary, my father Peter, and my mother-in-law Cathy, as well as the rest of my family and close friends who know who they are and need not be enumerated. I am lucky to know so many genuinely kind people, and I will always be be grateful for their support.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

CHAPTER I

Introduction

This dissertation presents a technique for improving the reliability of software. Software drives nearly everything we do, including transportation, telecommunications, energy, medicine, and banking. As we increasingly depend on software for our infrastructure, it becomes ever more important that it works without error. Software failures can be costly, and in critical systems they can be catastrophic. Studies estimate that software bugs cost the US economy about $60 billion per year [57]. It is therefore an important challenge to develop tools and techniques to improve software reliability.

Model checking is one general strategy to improve software reliability. A software model checker is an automatic tool that exhaustively tests a program on all possible inputs (usually up to a given size) and on all possible nondeterministic schedules. Thus, unlike techniques based on branch coverage [63, 27], a model checker can guarantee total state coverage within its bounds, eliminating the possibility of unchecked error states. Unlike formal proof-based techniques [4, 40, 55], model checking is automatic, requiring little effort on the part of the user. However, even when the bounds on the inputs are small, the number of inputs and schedules that need to be checked can be very large. In that case, it is infeasible to simply enumerate and test all possible states. This has motivated much research in state space reduction techniques, which reduce the amount of work a model checker has to do while maintaining the full coverage guarantee.

One way to reduce the state space of a model checker is to create an abstraction of the program being checked by using a technique such as predicate abstraction [3, 32, 9]. This abstraction is much simpler than the original program and has fewer states to explore. The abstraction is sound in the sense that if the abstraction is shown to be free of bugs then the original program must be free of bugs. However, the abstraction may contain bugs that are not in the original program. If such a false positive is

found, the abstraction must be refined to eliminate the false positive. This technique is known as Counter Example Guided Abstraction Refinement or *CEGAR*.

Another state space reduction technique is partial order reduction, which is effective when checking concurrent programs. Model checkers that use partial order reduction [25, 26] avoid checking multiple nondeterministic schedules that have provably identical runtime behavior. Thus, partial order reduction can be said to eliminate a certain kind of redundancy in the state space of a model checker. There are other techniques as well that exploit symmetries to eliminate state space redundancy [36].

Unfortunately, model checking has so far been limited in its applicability. When applied to hardware, model checkers have successfully verified fairly complex finite state control circuits with up to a few hundred bits of state information; but not circuits in general that have large data paths or memories. Similarly, for software, model checkers have primarily verified event sequences with respect to temporal properties; but not much work has been done to verify programs that manipulate rich complex data with respect to data-dependent properties. There is a combinatorially large number of possible states in such programs. For example, a binary tree with n nodes has a number of possible tree shapes exponential in n.

Thus, while there is much research on model checkers [3, 5, 9, 13, 14, 23, 26, 67, 32, 51] and on state space reduction techniques for software model checkers, none of these techniques seem to be effective at reducing the state space of model checkers in the presence of programs that manipulate complex data, such as data structures. For example, predicate abstraction relies on an alias analysis that is often too imprecise to describe heap manipulations such as those used by data structures. Partial order reduction is effective at reducing the number of nondeterministic schedules but it does little to cope with the large number of possible states of a data structure.

We present *glass box model checking*, a type of software model checking that can achieve a high degree of state space reduction in the presence of complex data.

1.1 Motivating Example

Consider checking the ordered binary search tree implementation in Figure 1.2. Suppose we would like to check that the search tree is always ordered. There are two operations to check: `get` and `insert`. (We omit the `delete` operation for simplicity.) The ordering invariant is described by the `repOk` method, such that `repOk` returns true for states that satisfy the invariant.

A model checking technique (in effect) exhaustively checks every valid state of

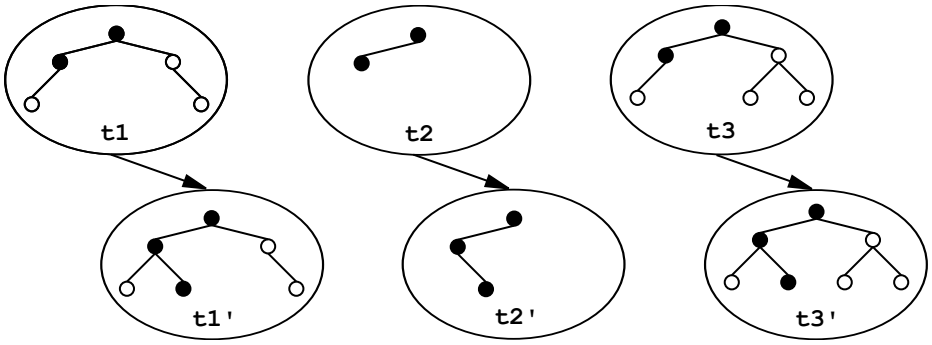


Figure 1.1: Three search trees (code in Figure 1.2), before an after an `insert` operation. The tree path touched by the operation is highlighted in each case. Note that the tree path is the same in all three cases. Once our system checks the `insert` operation on tree `t1`, it performs a static analysis to rule out the possibility of bugs in trees `t2` and `t3`.

`SearchTree` within some given finite bounds. Given a bound of 3 on the height of the tree, Figure 1.1 shows some possible states of `SearchTree`.

Consider checking an `insert` operation on state `t1` in Figure 1.1. After the operation, the resulting state is `t1'`. During execution, the `insert` operation touches only a small number of tree nodes along a tree path. These nodes are highlighted in the figure. Thus, if these nodes remain unchanged, the `insert` operation will behave *similarly* (e.g., on trees `t2` and `t3`).

At this point we would like to conclude that it is redundant to check the operation on `t1`, `t2`, and `t3`. Then we could only check `t1` and achieve a high degree of state space reduction by not checking all of the similar transitions. However, this is not sound, and can lead to bugs being missed. To see this, suppose the invariant of `SearchTree` includes a *balancing* invariant in addition to the ordering invariant, so that all trees must be full up to depth $h-1$, where h is the height of the tree. Observe that `t1'` and `t3'` are properly balanced but `t2'` is not. Therefore it would be incorrect to check the transition on `t1` and conclude that all similar transitions (such as that from `t2` to `t2'`) maintain the invariant. Simply pruning `t2` and `t3` from the search space will cause bugs to be missed.

To address this, we use a static analysis to efficiently discover if any similar transitions violate the invariant. After checking `t1`, our static analysis exploits similarities in the state space to find similar transitions that violate the invariant, or determine that none exist. With the addition of the static analysis, it becomes sound to prune all states similar to `t1`. Thus, our technique does not just eliminate redundancy in

```
1   class SearchTree implements Map {
2       static class Node {
3           int key;
4           Object value;
5           @Tree Node left;
6           @Tree Node right;
7
8           Node(int key, Object value) {
9               this.key = key;
10              this.value = value;
11          }
12      }
13
14      @Tree Node root;
15
16      Object get(int key) {
17          Node n = root;
18          while (n != null) {
19              if (n.key == key)
20                  return n.value;
21              else if (key < n.key)
22                  n = n.left;
23              else
24                  n = n.right;
25          }
26          return null;
27      }
28
29      void insert(int key, Object value) {
30          Node n = root;
31          Node parent = null;
32          while (n != null) {
33              if (n.key == key) {
34                  n.value = value;
35                  return;
36              } else if (key < n.key) {
37                  parent = n;
38                  n = n.left;
39              } else {
40                  parent = n;
41                  n = n.right;
42              }
43          }
44
45          n = new Node(key, value);
46          if (parent == null)
47              root = n;
48          else if (key < parent.key)
49              parent.left = n;
50          else
51              parent.right = n;
52      }
53
54      @Declarative
55      boolean repOk() {
56          return isOrdered(root, null, null);
57      }
58
59      @Declarative
60      static boolean isOrdered(Node n, Node low, Node high) {
61          if (n == null) return true;
62          if (low  != null && low.key  >= n.key) return false;
63          if (high != null && high.key <= n.key) return false;
64          if !(isOrdered(n.left,  low, n   ))    return false;
65          if !(isOrdered(n.right, n,   high))    return false;
66          return true;
67      }
68  }
```

Figure 1.2: A simple search tree implementation.

the state space, but also uses similarities in the state space to soundly eliminate large classes of non-redundant states as well. For this example, we only explicitly check each operation once on each unique tree path rather than each unique tree. This leads to significant reduction in the size of the state space.

1.2 Glass Box Model Checking

We call our technique *glass box model checking*. In general, the glass box technique works as follows. First, our checker initializes its search space to the set of all valid states in a program, up to some finite bounds. Next, our checker chooses an unchecked state in the search space. While executing from this state, our checker tracks information about which parts of the program state are accessed and how they are used. Using this information, our checker identifies a (usually very large) set of states that must behave similarly. Then our checker constructs a formula asserting that all similar states are bug-free, and checks it using a satisfiability solver. This leads to several orders of magnitude speedups over previous model checking approaches.

1.3 Modular Glass Box Model Checking

Programs are commonly divided into modules that each implement a self-contained part of a program. For example, many object-oriented languages such as Java provide classes and packages as a way to organize modules. It is common practice to perform *unit testing*, where each module is tested in isolation. This suggests a modular approach to model checking as well, where each module is checked by a distinct instance of a model checker. If a model checker has a state space of size n when checking a module, checking a program composed of two such modules could require a state space as large as n^2. Using a modular approach, two model checkers each explore a state space of size n for a total of $2n$. Thus, modularity has the potential to significantly reduce the state space of model checkers.

We present a system for *modular* glass box software model checking, to further improve the scalability of glass box software model checking. In a modular checking approach program modules are replaced with *abstract implementations*, which are functionally equivalent but vastly simplified versions of the modules. The problem of checking a program then reduces to two tasks: checking that each program module

behaves the same as its abstract implementation, and checking the program with its program modules replaced by their abstract implementations [12].

Extending traditional model checking to perform modular checking is trivial. For example, Java Pathfinder (JPF) [67] or CMC [51] can check that a program module and an abstract implementation behave the same on every sequence of inputs (within some finite bounds) by simply checking every reachable state (within those bounds).

However, it is nontrivial to extend glass box model checking to perform modular checking, while maintaining the significant performance advantage of glass box model checking over traditional model checking. In particular, it is nontrivial to extend glass box checking to check that a module and an abstract implementation behave the same on every sequence of inputs (within some finite bounds). This is because, unlike traditional model checkers such as Java Pathfinder or CMC, our model checker does not check every reachable state separately. Instead it checks a (usually very large) set of similar states in each single step.

1.4 Glass Box Model Checking of Type Soundness

Type systems provide significant software engineering benefits. Types can enforce a wide variety of program invariants at compile time and catch programming errors early in the software development process. Types serve as documentation that lives with the code and is checked throughout the evolution of code. Types also require little programming overhead and type checking is fast and scalable. For these reasons, type systems are the most successful and widely used formal methods for detecting programming errors. Types are written, read, and checked routinely as part of the software development process. However, the type systems in languages such as Java, C#, ML, or Haskell have limited descriptive power and only perform compliance checking of certain simple program properties. But it is clear that a lot more is possible. There is therefore plenty of research interest in developing new type systems for preventing various kinds of programming errors [8, 17, 31, 53, 54, 69].

A formal proof of type soundness lends credibility that a type system does indeed prevent the errors it claims to prevent, and is a crucial part of type system design. At present, type soundness proofs are mostly done on paper, if at all. These proofs are usually long, tedious, and consequently error prone. There is therefore a growing interest in machine checkable proofs of soundness [2]. However, both the above approaches—proofs on paper (e.g., [22]) or machine checkable proofs (e.g., [56])—

require significant manual effort.

Consider an alternate approach for checking type soundness *automatically* using a glass box software model checker. Our idea is to systematically generate every type correct intermediate program state (within some finite bounds), execute the program one small step forward if possible using its small step operational semantics, and then check that the resulting intermediate program state is also type correct—but do so efficiently by using glass box model checking to detect similarities in this search space and prune away large portions of the search space. Thus, given only a specification of type correctness and the small step operational semantics for a language, our system automatically checks type soundness by checking that the progress and preservation theorems [60, 71] hold for the language (albeit for program states of at most some finite size).

Since the initial state of a program can be any of a very large number of well-typed states, the problem of checking type soundness is difficult to even formulate in most model checkers. However, glass box model checking is well suited to handle this sort of input nondeterminism.

Note that checking the progress and preservation theorems on all program states up to a finite size does not *prove* that the type system is sound, because the theorems might not hold on larger unchecked program states. However, in practice, we expect that all type system errors will be revealed by small sized program states. Our experiments using mutation testing suggest that the small scope conjecture also holds for checking type soundness. We also examined all the type soundness errors we came across in literature and found that in each case, there is a small program state that exposes the error. Thus, exhaustively checking type soundness on all program states up to a finite size does at least generate a high degree of confidence that the type system is sound.

1.5 Contributions

This dissertation builds on previous work on glass box model checking [16], glass box model checking of type soundness [62] and modular glass box model checking [61]. We make the following contributions.

- **Efficient software model checking of data oriented programs:** We present glass box software model checking, a method for efficiently checking programs that manipulate complex data. Our key insight is that there are classes of operations that affect a program's state in similar ways. By discovering these similarities, we can dramatically reduce the state space of our model checker by checking each class of states in a single step. To achieve this state space reduction, we employ a dynamic analysis to detect similar state transitions and a static analysis to check the entire class of transitions. Our analyses employ a symbolic execution technique that increases their effectiveness.

- **Technique for modular checking:** We present a modular extension to glass box model checking, which allows us to efficiently check programs composed of many modules. We check each module for conformance to an abstract implementation of the module. Then the abstract implementation is used while checking other modules of the program. This further improves the efficiency and scalability of glass box model checking.

- **Checking type soundness:** We show how glass box checking can efficiently demonstrate the soundness of experimental type systems. Since proving type soundness can be extremely difficult, a model checking approach takes a considerable burden off the language designer.

- **Formal description:** We formalize our core technique and prove its correctness. Formalization is important for establishing the correctness of our analyses and for ensuring correct implementation of a glass box checker. Proving correctness aids evaluation of our technique and assures users of our system that bugs will not be missed.

- **Evaluation:** We give experimental evidence that glass box model checking is effective at checking properties of programs and soundness of type systems. We test our modular technique by comparing its performance to that of our non-modular technique, showing that modularity vastly improves the efficiency and scalability of our analysis. In comparisons with other model checkers, we show that glass box model checking is more efficient at checking these programs.

1.6 Organization

The rest of this dissertation is organized as follows. Chapter II describes the basic glass box model checking approach. Chapter III describes the modular extension to glass box checking. Chapter IV shows how to use glass box model checking to check the soundness of type systems. Chapter V presents a formalization of our glass box algorithm and a proof of its correctness. Chapter VI presents experimental results. Chapter VII presents related work and Chapter VIII concludes.

CHAPTER II

Glass Box Model Checking

Model checking is a formal verification technique that exhaustively tests a piece of hardware or software on all possible inputs (usually up to a given size) and on all possible nondeterministic schedules. For hardware, model checkers have successfully verified fairly complex finite state control circuits with up to a few hundred bits of state information; but not circuits in general that have large data paths or memories. Similarly, for software, model checkers have primarily verified control-oriented programs with respect to temporal properties; but not much work has been done to verify data-oriented programs with respect to complex data-dependent properties.

Thus, while there is much research on software model checkers [3, 5, 9, 13, 14, 23, 26, 67, 32, 51] and on state space reduction techniques for software model checkers such as partial order reduction [25, 26] and tools based on predicate abstraction [29] such as Slam [3], Blast [32], or Magic [9], none of these techniques seem to be effective in reducing the state space of data-oriented programs. For example, predicate abstraction relies on alias analysis that is often too imprecise. This chapter presents *glass box model checking*, a technique capable of efficiently checking data-oriented programs.

For example, consider checking that a red-black tree [15] implementation maintains the usual red-black tree invariants. Previous model checking approaches such as JPF [67, 43], CMC [51, 50], Korat [5], or Alloy [37, 41] systematically generate all red-black trees (up to a given size n) and check every red-black tree operation (such as `insert` or `delete`) on every red-black tree. Since the number of red-black trees with at most n nodes is exponential in n, these systems take time exponential in n for checking a red-black tree implementation. Our key idea is as follows. Our glass box checker detects that any red-black tree operation such as `insert` or `delete` accesses only one path in the tree from the root to a leaf (and perhaps some nearby nodes). Our checker then determines that it is sufficient to check every operation on every

unique tree path (and some nearby nodes), rather than on every unique tree. Since the number of unique red-black tree paths is polynomial in n, our checker takes time polynomial in n.

In general, the glass box technique works as follows. First, our checker initializes its search space to the set of all valid states in a program, up to some finite bounds. Next, our checker chooses an unchecked state in the search space. While executing from this state, our checker tracks information about which parts of the program state are accessed and how they are used. Using this information, our checker identifies a (usually very large) set of states that must behave similarly. Then our checker checks the entire set of states in a single step. This leads to several orders of magnitude speedups [16] over previous model checking approaches.

Note that like most model checking techniques [5, 23, 26, 67, 51], our system (in effect) exhaustively checks all states in a state space within some finite bounds. While this does not guarantee that the program is bug free because there could be bugs in larger unchecked states, in practice, almost all bugs are exposed by small program states. This conjecture, known as the *small scope hypothesis*, has been experimentally verified in several domains [38, 47, 59]. Thus, exhaustively checking all states within some finite bounds generates a high degree of confidence that the program is correct (with respect to the properties being checked).

Compared to glass box checking, formal verification techniques that use theorem provers [4, 40, 55] are fully sound. However, these techniques require significant human effort (in the form of loop invariants or guidance to interactive theorem provers). For example, an unbalanced binary search tree implemented in Java can be checked using the glass box technique with less than 20 lines of extra Java code, implementing an abstraction function and a representation invariant. In fact, it is considered a good programming practice [46] to write these functions anyway, in which case glass box checking requires no extra human effort. However, checking a similar program using a theorem prover such as Coq [4] requires more than 1000 lines of extra human effort.

Compared to glass box checking, other model checking techniques are more automatic because they do not require abstraction functions and representation invariants. However, glass box checking is significantly more efficient than other model checkers for checking certain kinds of programs and program properties.

We present glass box model checking as a middle ground between automatic model checkers and program verifiers based on theorem provers that require much more extensive human effort.

The following sections in this chapter present the glass box model checking ap-

proach in detail. First we show how program properties are specified. Next we define the search space, present the glass box algorithm, and discuss how to represent the search space efficiently. We continue with detailed descriptions of our dynamic and static analysis techniques, as well as a discussion about how we translate some methods in logical formulas.

This chapter uses the binary search tree from Chapter I (Figure 1.2) as a running example.

2.1 Specification

Our analysis guarantees coverage of every state that satisfies the program invariant. The programmer must supply this invariant, which typically appears in a method called `repOk`. The `repOk` method returns true for every state that satisfies the program invariant and returns false (or raises an exception) for every state that does not satisfy the invariant. Our analysis works by translating `repOk` into a formula, which must be done efficiently in terms of both time complexity and size of formula. Toward this end we define a *declarative* subset of Java that greatly facilitates this translation process. Note that in Figure 1.2, the methods `repOk` and `isOrdered` are annotated as `Declarative`, which indicates that they use the declarative syntax. As evidenced by these methods, the declarative syntax is expressive and capable of representing complex invariants in a way that is familiar to programmers. We present a detailed account of declarative methods and their syntax in Section 2.9. We require that the `repOk` method is always declarative.

In addition to the program invariant, the programmer can specify `precondition` and `postcondition` methods. Our analysis checks every state that satisfies the precondition and the invariant, and checks that the postcondition and the invariant hold after each operation. Furthermore, the programmer can insert assertions in the program code itself, and we provide a utility method `assume`. These mechanisms are detailed in Section 2.11

Also note the `Tree` annotations in Figure 1.2, which denote that the `Node`s form a tree rooted at the field `root`. The tree property is considered part of the program invariant. Thus, these annotations reduce our search space because we do not have to check non-tree structures. They also relieve programmers of the burden of specifying a tree structure in `repOk`. However, our analysis must check that the tree structure is maintained as the program executes. We describe how we perform this check in Section 2.10.

Field	Domain	Tree 1	Tree 2
root	{null, n0}	n0	n0
n0.left	{null, n1}	n1	n1
n0.right	{null, n2}	n2	null
n1.left	{null, n3}	n3	n3
n1.right	{null, n4}	null	null
n2.left	{null, n5}	null	
n2.right	{null, n6}	n6	
n0.key	{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}	5	6
n1.key	{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}	2	7
n2.key	{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}	7	
n3.key	{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}	0	4
n4.key	{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}		
n5.key	{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}		
n6.key	{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}	9	
n0.value	{a, b, c, d}	a	b
n1.value	{a, b, c, d}	d	d
n2.value	{a, b, c, d}	a	
n3.value	{a, b, c, d}	b	c
n4.value	{a, b, c, d}		
n5.value	{a, b, c, d}		
n6.value	{a, b, c, d}	d	
method	{get, insert}	insert	insert
get.key	{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}		
insert.key	{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}	3	3
insert.value	{a, b, c, d}	a	a

(a) (b)

Figure 2.1: (a) Search space for the binary tree in Figure 1.2 with tree height at most 3 and at most 10 keys and 4 values, and (b) two elements of that search space. Tree 1 is ordered and Tree 2 is not ordered.

The current implementation of our system checks Java programs, and we present it in that context. However, the glass box checking technique we describe in this chapter is general and can be used to check programs in other languages as well.

2.2 Search Space

We define the search space of a model checking task by specifying bounds on the program states that we will check. For example, consider checking the binary search tree implementation in Figure 1.2. Suppose we must check all trees of tree height at most 3, with at most 10 different possible keys and at most 4 different possible

values. The corresponding search space is shown in Figure 2.1(a). The tree may have any shape within its height bound because the pointers between nodes may be `null`. Every element in this search space represents a binary tree, along with an operation to run on that tree. Figure 2.1(b) shows two elements of this search space. Tree 1 represents the operation `insert(3,a)` on an ordered tree. Tree 2 represents the same operation on an unordered tree, because key 7 in node `n1` is greater than key 6 in node `n0`. The search space thus may include elements that violate the invariant.

Traditional software model checkers [3, 9, 14, 26, 67, 32, 51] explore a state space by starting from the initial state and systematically generating and checking every successor state. This approach does not work for software model checkers that use the glass box technique. Instead, we check every state within the search space that satisfies the invariant.

2.3 Search Algorithm

Figure 2.2 presents the pseudo-code for the glass box search algorithm. Given a bounded search space B, in Line 2 the glass box technique initializes the search space S to all *valid* states in B. For example, given the bounded search space B in Figure 2.1(a), the initial search space S contains all states in B on which `repOk` returns true. Lines 3-12 iterate until the search space S is exhausted. In each iteration, an unchecked state s is selected from S and the desired property is checked on it. For example, when checking the binary search tree in Figure 1.2, we check that executing the operation on s preserves its invariant. In Line 6 a set S' of states similar to s is constructed using the dynamic analysis described in Section 2.6. In Line 7 the entire set of states S' is checked using the static analysis described in Section 2.7. If any of the states fails the check, we obtain an explicit bug trace in Line 9. Finally, in Line 11 all the checked states S' are removed from S. The following sections describe the above steps in detail.

2.4 Search Space Representation

In the above algorithm, several operations are performed on the search space, including choosing an unchecked element (Line 4 in Figure 2.2), constructing a subset (Line 6), checking the subset (Line 7), and pruning the subset from the search space

```
 1: procedure GlassBoxSearch(BoundedSearchSpace B)
 2:     S ← Set of all valid elements in B
 3:     while S ≠ ∅ do
 4:         s ← Any element in S
 5:         Check the desired property on s
 6:         S′ ← Elements similar to s
 7:         Check the property on all elements in S′
 8:         if any s′ ∈ S′ fails the check then
 9:             Print bug trace s′
10:         end if
11:         S ← S − S′
12:     end while
13: end procedure
```

Figure 2.2: Pseudo-code for the glass box search algorithm.

(Line 11). Consider checking the binary search tree in Figure 1.2 on trees with at most n nodes. The size of the search space is exponential in n. However, our model checking algorithm described below completes the search in time polynomial in n. Thus, if we are not careful and choose an explicit representation of the search space, then search space management itself would take exponential time and negate the benefits of our search space reduction techniques. We avoid this by choosing a compact representation. We represent the search space as a finite boolean formula. We use the incremental SAT solver MiniSat [24] to perform the various search space operations.

For example, consider the search space in Figure 2.1(a). We encode the value of each field using $\lceil \log_2 n \rceil$ boolean variables, where n is the size of the domain of the field. So we encode `n0.key` with four boolean variables and `n1.right` with one boolean variable. A formula over these bits represents a set of states. For example, the following formula represents the set of all trees of height one: `root` $=$ `n0` $\wedge$ `n0.left` $=$ `null` $\wedge$ `n0.right` $=$ `null`. We invoke the SAT solver to provide a satisfying assignment to the variables of the formula and then decode it into a concrete state. Thus there may be expensive operations at Line 3 in Figure 2.2, checking if a set is empty, and Line 4, choosing an element of a non-empty set, because they invoke the SAT solver. Line 11 in Figure 2.2, subtracting one set (S') from another (the search space S), takes linear time (w.r.t. size of S') because it only injects clauses (in S') into the incremental SAT solver.

2.5 Search Space Initialization

In Line 2 of Figure 2.2, given a bounded search space B, we first initialize the search space S to the set of all valid states in B. For example, given the bounded search space B in Figure 2.1(a), we first initialize the search space S to all states in B on which `repOk` returns true. This requires constructing a boolean formula that represents all states that satisfy `repOk`. We accomplish this by translating the `repOk` method and all the methods that `repOk` transitively invokes into such a boolean formula, given the finite bounds. For example, translating the `repOk` method of the binary search tree in Figure 1.2 with a tree height of at most of two produces the following boolean formula: `root` $=$ `null` $\vee$ `((n0.left` $=$ `null` $\vee$ `(n1.key` $<$ `n0.key))` $\wedge$ `(n0.right` $=$ `null` $\vee$ `(n2.key` $>$ `n0.key)))`. Section 2.9 describes how we translate declarative methods such as `repOk` into formulas.

2.6 Dynamic Analysis

Given an element of the search space, the purpose of the dynamic analysis (Line 6 in Figure 2.2) is to identify a set of similar states that can all be checked efficiently in a single step by the static analysis described in Section 2.7.

Consider checking the binary search tree implementation in Figure 1.2. Suppose that we choose the unchecked state shown in Tree 1 in Figure 2.1(b). Call this state $s1$. We run the corresponding `insert(3,a)` operation on the state $s1$ to obtain the state $s2$.

As the method `insert` is concretely executed in the above example, it is also symbolically executed [44] to build a *path constraint*. The symbolic execution tracks formulas representing the values of variables and fields. The path constraint is a formula that describes the states in the search space that follow the same path through the program as the current concrete execution. For example, in the above concrete execution, the first branch point is on Line 32 (in Figure 1.2), with branch condition `n != null`. At this program point, `n` has the concrete value of `n0` and the symbolic value of `root`. The symbolic value of the branch condition is thus `root` $\neq$ `null`. This symbolic value is saved. The concrete value of the branch condition is `true`, so the control flow proceeds into the while loop. The next branch in the concrete execution is on Line 33, testing `n.key == key`. This symbolically evaluates to `n0.key` $=$ `key`, concretely to `false`. Execution continues in this way. Figure 2.3 summarizes all branch

Line	Symbolic Value of Branch Condition	Concrete Value of Branch Condition
32	root≠null	true
33	n0.key=key	false
36	key<n0.key	true
32	n0.left≠null	true
33	n1.key=key	false
36	key<n1.key	false
32	n1.right≠null	false
46	n1=null	false
48	key<n1.key	false

Figure 2.3: Symbolic and concrete values of the branch conditions encountered during the execution of the `insert(3,a)` operation on Tree 1 in Figure 2.1(b). The symbolic values are used to generate the path constraint.

conditions encountered during execution of the `insert` method.

We generate the path constraint by taking the conjunction of the symbolic branch conditions, with the false conditions negated. All states satisfying the path constraint are considered *similar* to each other (Line 6 in Figure 2.2). In the binary tree example, the `insert` method does not find `key` in the tree, so it inserts a new node. The path constraint asserts that `root` and `n0.left` are not null, but `n1.right` is null. The parameter `key` must be less than `n0.key` and greater than `n1.key`. (The path constraint also asserts that the method being checked is `insert`).

In general, path constraints are not just branch conditions but also include values used in instructions that cannot efficiently execute symbolically. This includes parameters to external code and receiver objects of field assignments. In addition, instructions that may result in runtime exceptions also generate path constraints. Figure 2.4 summarizes Java constructs that execute symbolically without generating path constraints (except for a possible exception condition).

Note from Figure 2.4 that calls to declarative methods execute symbolically. Declarative methods do not contain side effects. Given a declarative method and the current symbolic program state, we generate a symbolic return value of the declarative method by translating the declarative method (and the methods it transitively invokes) into a formula. (We present a detailed explanation of this process in Section 2.9). Thus we can use declarative methods during the dynamic analysis without increasing the size of the path constraint. This allows us to identify a larger set of similar states.

Construct	Restrictions	Exception Conditions

`i<j  i>j  i<=j  i>=j`
`i+j  i-j`
`+i  -i  ~i`
`i&j  i|j  i^j`
`x==y  x!=y`
`!a`
`a||b  a&&b`
`if c return x else return y`
`x=e`
`x.m(..)`
`x.f`
`x[i]`
`(C)x`
`x instanceof C`
	`b` is effect free
	`x` and `y` are effect free
	
	`m` is declarative
		
		`x`≠`null`
		`x`≠`null`
		`x`≠`null` $\wedge$ `i` ≥ 0 $\wedge$ `i` $<$ `x.length`
		`x`=`null` $\vee$ (`x` is an instance of `C`)

Figure 2.4: Java constructs that execute symbolically without generating path constraints (except for exception condition). The restrictions indicate the conditions under which these constructs execute symbolically without generating path constraints. The exception conditions are constraints that are added to the path constraint when no exception is thrown during the concrete execution.

2.7 Static Analysis

The dynamic analysis above identifies a set S' of similar states (Line 6 in Figure 2.2) that follow the same program path during execution. But the fact that the code works correctly on one of the states does not necessarily imply that it works correctly on *all* of them. For example, a buggy `get` method of a binary tree might correctly traverse the tree, but return the value of the node's parent instead of the value of the node itself. By chance we might have chosen a state where the two values are the same. That particular state would not expose the bug, but most of the similar states would. The purpose of the static analysis is to check that the code works correctly on all of the similar states in S' (Line 7 in Figure 2.2).

Consider checking the binary search tree implementation in Figure 1.2. To check that the code works correctly on all the states S' that follow the same program path, we construct the formula: $S' \rightarrow R$. The proposition S' asserts that a state $s1$ is among the ones we are checking. R asserts that a state $s1$ transitions to a state $s2$ such that $s2$`.repOk()` returns `true`. We use a SAT solver to find a counterexample to this formula, or establish that none exists. If we find a counterexample, we decode it into a concrete state and present it as an explicit bug trace.

Field	Symbolic Value
root	pre.root
n0.left	pre.n0.left
n0.right	pre.n0.right
n1.left	pre.n1.left
n1.right	n'
n2.left	pre.n2.left
n2.right	pre.n2.right
n0.key	pre.n0.key
n1.key	pre.n1.key
n2.key	pre.n2.key
n3.key	pre.n3.key
n4.key	
n5.key	pre.n5.key
n6.key	pre.n6.key
n0.value	pre.n0.value
n1.value	pre.n1.value
n2.value	pre.n2.value
n3.value	pre.n3.value
n4.value	
n5.value	pre.n5.value
n6.value	pre.n6.value
n'.key	pre.insert.key
n'.value	pre.insert.value
method	pre.*method*
get.key	pre.get.key
insert.key	pre.insert.key
insert.value	pre.insert.value

Figure 2.5: Symbolic state of the search tree in Figure 1.2 generated by symbolically executing the `insert` operation on Tree 1 in Figure 2.1(b). Node `n'` is a fresh node created during the operation.

To generate the formula R, we symbolically execute the operation using the dynamic analysis described above in Section 2.6. After symbolic execution, every field and variable contains a symbolic value that is represented by a formula. For example, Figure 2.5 shows the symbolic state of the binary tree in Figure 1.2 generated by symbolically executing the `insert` method on Tree 1 in Figure 2.1(b).

2.8 Isomorphism Analysis

Consider checking a method `foo` with three formal parameters `p1`, `p2`, and `p3`. Figure 2.6(a) presents an example of such a search space, where each method parameter can be one of three objects `o1`, `o2`, and `o3`. Consider the two elements of the above search space in Figure 2.6(b). These two elements are isomorphic because

Field	Domain
o1.value	{null, o1, o2, o3}
o2.value	{null, o1, o2, o3}
o3.value	{null, o1, o2, o3}
method	{foo}
foo.p1	{o1, o2, o3}
foo.p2	{o1, o2, o3}
foo.p3	{o1, o2, o3}

(a)

Element 1	Element 2
null	null
null	null
null	null
foo	foo
o1	o2
o2	o1
o3	o3

(b)

Figure 2.6: (a) Search space for checking a method `foo` with three formal parameters `p1`, `p2`, and `p3` that can each be one of three objects `o1`, `o2`, and `o3`. (b) Two isomorphic elements of this search space. Element 1 and Element 2 are isomorphic because `o1` and `o2` are equivalent memory locations.

`o1` and `o2` are equivalent memory locations. Therefore, once we check Element 1, it is redundant to check Element 2. We avoid checking isomorphic elements as follows. Consider Element 1 in Figure 2.6(b). Suppose that the execution of the method `foo` depends only on the values of `p1` and `p2`, and the analyses in the previous sections conclude that all states where (`p1=o1` $\wedge$ `p2=o2`) can be pruned. The isomorphism analysis then determines that all states that satisfy the following formula can also be safely pruned: (`p1` $\in$ {`o2`,`o3`} $\vee$ (`p1=o1` $\wedge$ `p2` $\in$ {`o3`})).

In general, given a program state `s`, we construct such a formula $\mathtt{I_s}$ denoting the set of states isomorphic to `s` as follows. Recall from Section 2.6 that the symbolic execution on `s` builds a path constraint formula, say $\mathtt{P_s}$. Suppose during symbolic execution we encounter a fresh object `o` by following a field `f` that points to `o`. Suppose the path constraint built so far is $\mathtt{P'_s}$. The isomorphism analysis includes in $\mathtt{I_s}$ all states that satisfy ($\mathtt{P'_s} \wedge \mathtt{f{=}o'}$), for every $\mathtt{o'}$ in the domain of the field `f` that is another fresh object. We then prune all the states denoted by $\mathtt{I_s}$ from the search space.

Note that some software model checkers also prune isomorphic program states using heap canonicalization [35, 49]. The difference is that in heap canonicalization, once a checker *visits* a state, it canonicalizes the state and checks if the state has been previously visited. In contrast, once we check a state `s`, we compute a compact formula $\mathtt{I_s}$ denoting a (often exponentially large) set of states isomorphic to s, and prune $\mathtt{I_s}$ from the search space. We *never visit* the (often exponentially many) states in the set $\mathtt{I_s}$.

2.9 Declarative Methods and Translation

The above search algorithm relies on efficiently translating declarative methods into formulas. The efficiency must not only be in the speed of translation, but also in the compactness of the final formula so that it can be efficiently solved by a SAT solver. To achieve this, we restrict declarative methods to use a subset of Java and be free of side effects.

Declarative methods have the `Declarative` annotation. A declarative method may not contain exception handlers, and may only call declarative methods. Declarative methods allow assignment only to local variables, and permit a limited form of temporary object creation and iterative loop structures, described below. Declarative methods may be overridden only by other declarative methods. Note that declarative methods can contain recursion, so our declarative subset of Java is Turing complete. Our experience indicates that declarative methods are sufficiently expressive to write program specifications (such as invariants and assertions).

The translation process is somewhat similar to that of AAL [42]. However, because our declarative methods do not contain side effects, the formulas for declarative methods that we generate are considerably simpler than the formulas for regular Java methods that AAL generates. We translate our declarative variant of Java directly into propositional logic, unlike AAL which first translates Java into Alloy [37] and then translates Alloy into propositional logic.

Non-declarative methods may call declarative methods. If a declarative method is encountered during symbolic execution, we symbolically execute the declarative method by translating it into a formula on the current symbolic state. Branches in declarative methods thus do not generate path constraints. Therefore, making methods declarative enables the checking and pruning of a larger set of states in each iteration of the loop in Figure 2.2. This is particularly useful for methods that depend on a large part of the state, such as a method that returns the number of nodes in a tree. We interpret some core Java library calls declaratively, including the identity hash function and the `getClass` method. This allows us to use such calls in specifications and to prune larger sets of similar states.

The glass box algorithm translates the same declarative method several times. For example, consider checking the code in Figure 1.2. During search space initialization (see Section 2.5), the above algorithm translates the invocation of the `isOrdered` method on each tree node. Subsequently, during each iteration of the loop in Figure 2.2, it again translates the invocation of the `isOrdered` method on each tree

node. However, note that each operation on the tree, such as an `insert` operation, changes only a small part of the tree. Thus, most invocations of the `isOrdered` method on the same tree node return the same formula. To speed up translation, we cache the formulas generated by translating the declarative methods during search space initialization. The cache stores a different formula per combination of parameters passed to a declarative method. We also maintain a list of fields that each cache entry depends on. If any of these fields changes during an operation then the cache entry is temporarily disabled, requiring the declarative method to be translated again on the changed state. Sometimes a declarative method is called with the same parameters multiple times per iteration (of the loop in Figure 2.2). If the cache entry is disabled or does not exist, the method must be translated every time. To avoid that, we use a *temporary* cache. When a declarative method is called, we look up the method and its parameters in the main cache. If the cache misses (because the entry is disabled or was never created) then we try the temporary cache. If that misses then we translate the method and store the result in the temporary cache. After every iteration, the temporary cache is cleared and all main cache entries are enabled. This caching system improves the performance of glass box checking considerably.

2.9.1 Core Translation

Consider a core syntax of declarative methods that includes only `if` statements, calls to declarative methods, and return statements, as well as expressions with no side effects. For example, the declarative methods in Figure 1.2 have such a syntax. We translate such methods to formulas in a straightforward way. Each return statement is changed into a guarded command, where the guards are the particular conditions from the `if` statements that cause control flow to reach the return statement. Given the list of guarded return statements, we construct a formula asserting that under each guard, the return variable has the correct corresponding value.

Calls to other declarative methods are resolved recursively. If the condition guarding a method call is trivially unsatisfiable, we short circuit the recursion. Furthermore, we can detect when the same method is recursively called with the same parameters and short circuit the recursion. Nevertheless, it is possible that recursion will continue indefinitely. To address this, we assume that a stack overflow has occurred after a very large depth of recursion has been reached, and that no further recursion is needed. Any guard that leads to a stack overflow does not satisfy the invariant.

The core syntax is fully expressive for writing specifications. In the following sec-

tions we present enhancements to the core syntax that add convenience for programmers.

2.9.2 Assignment to Local Variables

The declarative syntax allows assignments to local (non-state) variables. When such assignments are present, we translate the method to Static Single Assignment (SSA) form, so each variable is assigned exactly once. Some of these assignments are at merge points in the code, and combine variables from different branches. We translate each assignment to a guarded command, as above, and use the guards to construct formulas defining the values of each such merge assignment. Then we replace each reference to a variable with the corresponding value, restoring the core syntax.

2.9.3 Iterative Structures

We also permit iterative structures such as `while` loops in declarative methods. When a `while` loop is present in declarative code, we effectively create a new method m_v for each variable v that is modified in the loop. The parameters to the new methods include every local variable that is accessed in the loop. The return value is the formula describing the variable v after the loop terminates. The method implements the body of the loop, returning if the loop condition does not hold and calling itself recursively if it does. Finally, we replace the loop with assignments for the form v `=` $m_v()$, such that each variable v is updated with its new value from the method m_v. This restores the core syntax, with assignments to variables processed as above.

2.9.4 Object Creation

Declarative methods may not alter any state variables, but they may create additional objects on the heap. Once initialized, the objects may not be modified from within a declarative method. This is a convenient way to aggregate multiple values into a single parameter or return value. Furthermore, this mechanism allows a declarative `clone` method to be created. Large parts of the program state can be cloned without adding to the path constraint, improving the efficiency of our analysis.

To allow object creation, we introduce declarative constructors. Declarative constructors have the same syntax as declarative methods, with the following exceptions.

```
1: procedure WARSHALL(Set T, Matrix m)
2:     for k ∈ T do
3:         for i ∈ T do
4:             for j ∈ T do
5:                 m[i][j] ← m[i][j] ∨ (m[i][k] ∧ m[k][j])
6:             end for
7:         end for
8:     end for
9: end procedure
```

Figure 2.7: Pseudo-code for the symbolic Warshall's algorithm that computes reachability. The input T is the set of tree field indices and the input m is the initial matrix. After the call, $m[i][j]$ contains a formula asserting that field f_j is reachable through tree fields from field f_i.

Like all constructors, declarative constructors may not return a value. Declarative constructors may call other declarative constructors as well as declarative methods. If the superclass default constructor is not declarative then another constructor must be specified. (We take the default `Object` constructor to be implicitly declarative.) Finally, declarative constructors may assign to their own fields, in addition to local variables. Since the new object is not shared at the time the constructor is invoked, its fields are effectively local.

We translate declarative constructors as though each field is being returned when the constructor exits. Thus we find formulas for each field of the new object. Furthermore, we repeat this process at each call site in the constructor. Since the new object could be passed as a parameter to a method or constructor, the fields of the object must accurately reflect its state at the call site.

2.10 Checking the Tree Structure

Recall that we specify the tree structure of programs by using `Tree` annotations. Each field with this annotation is considered a *tree field*. The object graph induced by the tree fields must have a tree rooted at the main program object. For example, the tree fields in Figure 1.2 require that the main `SearchTree` object is the root of a tree that includes the nodes reachable through the `root` field.

This property is considered to be part of the program invariant, so it must be translated into a formula and checked along with the rest of the invariant. We construct this formula as follows. First, we create a special tree field f_0 that always points

```
1: procedure TreeFormula(Set T)
2:     Initialize matrix m with tree field connections.
3:     Warshall(T, m)
4:     result ← true
5:     for i ∈ T do
6:         for j ∈ T where i < j do
7:             result ← result ∧ (m[0][i] ∧ m[0][j] → ¬Eq(f_i, f_j))
8:         end for
9:     end for
10: end procedure
```

Figure 2.8: Pseudo-code for building the formula asserting that the tree structure is valid. The input T is the set of tree field indices. After the call, *result* is a formula asserting that the tree fields form a tree rooted at f_0. The formula $Eq(f_i, f_j)$ asserts that fields f_i and f_j point to the same object.

to the main program object. For each tree field f, let $R(f)$ be a formula that is true exactly when the field f is reachable from f_0 through a path containing only tree fields. When $R(f)$ is true, we say that f is tree-reachable. For a pair of tree fields f_1 and f_2, let $Eq(f_1, f_2)$ be a formula that is true exactly when the fields f_1 and f_2 point to the same object. The formula $(R(f_1) \wedge R(f_2)) \rightarrow \neg Eq(f_1, f_2)$ asserts that when f_1 and f_2 are both tree-reachable, they point to different objects. The conjunction of this formula applied to all pairs of fields asserts that tree fields define a tree rooted at f_0. The $Eq(f_1, f_2)$ formulas are easily constructed by comparing the symbolic values stored in f_1 and f_2 for equality, and ensuring that they are not `null`. We construct the $R(f)$ formulas as described below.

To compute tree-reachability, we first construct a matrix m of formulas, where $m[i][j]$ is true when field f_j is reachable from field f_i through tree fields. Then $R(f_i)$ is equal to $m[0][i]$. We use a symbolic version of Warshall's algorithm to compute the entries of m. Figure 2.7 presents this algorithm. Next, we construct a formula asserting that if any two tree fields i and j are both reachable from the root object through tree fields, then i and j must not point to the same object. This algorithm appears in Figure 2.8.

The above technique will construct an formula asserting that the tree structure holds. We construct this formula once to initialize the search space and again at every iteration of the algorithm in Figure 2.2 to check that the tree structure is maintained. However, most aspects of the tree structure are not likely to have changed across transitions.

In Section 2.9 we presented a caching system for declarative methods that takes

```
1: procedure WarshallIncremental(Set T, Set M, Matrix m, Set L)
2:     I ← {i | ∃j ∈ M m[i][j] ≠ false}
3:     for k ∈ T − M do
4:         for i ∈ I do
5:             for j ∈ T do
6:                 m[i][j] ← m[i][j] ∨ (m[i][k] ∧ m[k][j])
7:             end for
8:         end for
9:     end for
10:    for k ∈ M do
11:        for i ∈ I do
12:            for j ∈ T do
13:                if i = 0 ∧ m[i][k] ≠ false ∧ m[k][j] ≠ false then
14:                    L ← L ∪ {j}
15:                end if
16:                m[i][j] ← m[i][j] ∨ (m[i][k] ∧ m[k][j])
17:            end for
18:        end for
19:    end for
20: end procedure
```

Figure 2.9: Pseudo-code for the incremental Warshall's algorithm. T is the set of tree fields indices, M is the set of modified field indices, and m is the previous matrix. This procedure adds to L the indices of all fields that are tree-reachable through a modified field. The only rows of m that are updated are rows i where field i can reach a modified field.

advantage of the fact that usually only a small part of the state is modified during an operation. Likewise, the efficiency of our tree checking is greatly improved by using an *incremental* approach. Given that the reachability matrix m has been created for the initial state, we would like to make minimal changes to update it to the final state. Consider an entry $m[i][j]$, which gives a condition for f_i reaching f_j in the initial state. If f_i was not modified and can't reach any modified fields (i.e. $m[i][k] = false$ for all modified f_k) then the formula $m[i][j]$ will be unchanged in the final state. Thus, only entries $m[i][j]$ where f_i can reach a modified field need to be considered. It is simple to calculate which f_i satisfy this criterion because we monitor changes in the fields. Since the number of modified fields is expected to be small, this significantly increases the efficiency of building the new matrix m.

We introduce one further optimization. After we update the matrix m as described above, we need to construct a formula $(R(f_i) \wedge R(f_j)) \rightarrow \neg Eq(f_i, f_j)$ for each pair of tree fields f_i and f_j. However, many of these formulas are likely to be unchanged,

```
1: procedure TreeFormulaIncremental(Set T, Set M, Matrix m)
2:     L ← M
3:     WarshallIncremental(T, M, m, L)
4:     result ← true
5:     for i ∈ L do
6:         for j ∈ T where i < j do
7:             result ← result ∧ (m[0][i] ∧ m[0][j] → ¬Eq(f_i, f_j))
8:         end for
9:     end for
10: end procedure
```

Figure 2.10: Pseudo-code for incrementally building the formula asserting that the tree structure is valid after a transition. T is the set of tree field indices, M is the set of all modified tree field indices, and m is the matrix computed by Warshall's algorithm before the transition. After the call, *result* contains a formula with the *additional* constraints for ensuring that the tree structure holds, given the modified fields in M.

and therefore trivially hold. We omit the construction of such formulas and avoid the computational complexity of considering every pair of fields. We can omit considering f_i and f_j when the following conditions hold:

- Neither field has been modified, so $Eq(f_i, f_j)$ is unchanged.
- Neither field is tree-reachable through any modified fields, so $R(f_i)$ and $R(f_j)$ are unchanged (or strictly weaker).

We generate a set L of tree fields indices that are tree-reachable through modified fields, or are modified themselves. Then, instead of considering every pair of tree fields, we only need consider pairs that include at least one field index from L. Identifying the modified fields is simple, since we track changes to fields. We identify the fields that are tree-reachable through modified fields by splitting the incremental Warshall's algorithm above into two phases. First, we only use values of k such that f_k is not modified. Thus, we compute reachability through only fields that have not been modified. Next, we complete the algorithm by considering values of k such that f_k is modified. During this phase, if the entry $m[0][j]$ is changed then f_j may be tree-reachable through f_k, and so we add j to L. We present the pseudo-code of the incremental version of Warshall's algorithm in Figure 2.9 and the pseudo-code of the incremental tree checking algorithm in Figure 2.10.

2.11 Advanced Specifications

In addition to the invariant defined by `repOk` and the tree property defined by the `Tree` annotation, we provide other ways for the programmer to create specifications. First, the programmer can define preconditions and postconditions using the declarative methods `precondition` and `postcondition` as follows.

```
@Declarative boolean precondition() {
    // specify precondition
}

@Declarative boolean postcondition(Object prestate) {
    // specify postcondition using prestate
}
```

In addition to the usual restrictions on the search space, the model checker will only check states where `precondition` returns true. Likewise, in addition to the checks that are performed after each transition, the model checker will ensure that `postcondition` returns true.

It is common for postconditions to check correctness relative to the state of the program before the transition, so we provide a parameter `prestate` to the postcondition. This parameter is effectively a copy of the program before the transition, and it can be accessed directly through its fields or indirectly through declarative method calls.

One subtle point is the behavior of the Java operator `==` when comparing objects from the prestate to objects from the current state. For example, consider the comparison `x == prestate.x` where the field `x` has not been modified. If the comparison evaluates to true then it's natural to assume that `x.f == prestate.x.f` will also evaluate to true (given that `x` is not `null`). However, that may not be the case, since `x.f` may have been modified. To avoid this confusion, we take `x == prestate.x` to be false. In general, an equality comparison between an object from the current state and an object from the prestate evaluates to false. Nevertheless, it is often useful to know if `x` is the same (but perhaps modified) object as `prestate.x`. We provide a method `isSameObject` for this purpose. `isSameObject(x, y)` returns true when `x` and `y` refer to the same object in memory, regardless of whether or not one or the other is in the prestate.

One common mechanism for program specification is to use `assert` statements to check important properties at certain places in the code. We support this practice by

interpreting `assert` statements as additional checks that must be made. We append the symbolic values of all assert statements to the postcondition and confirm after each transition that no asserts could have been violated. Additionally, we provide a method `assume` that restricts the search space to only those states that satisfy every call to `assume` encountered during execution. The `assert` and `assume` statements allow programmers to specify pre- and postconditions based on intermediate program states during a transition.

2.12 Conclusions

This chapter presents a system for glass box software model checking. A glass box software model checker does not check every state separately but instead checks a large set of states together in each step. A dynamic analysis discovers a set of similar states, and a static analysis checks all of them efficiently in a single step using a SAT solver. Our analysis achieves a high degree of state space reduction in this way.

Our technique includes features such as isomorph pruning and tree structure checking, and supports a number of ways for a programmer to specify program behavior. We present a convenient syntax for specifying *declarative* code, which we use for specifications. Declarative code can be efficiently translated into compact formulas.

CHAPTER III

Modular Glass Box Model Checking

This chapter extends the glass box software model checking technique presented in Chapter II by introducing *modularity*. Our modular approach further improves the scalability of the glass box technique when checking large programs composed of multiple modules. In a modular checking approach program modules are replaced with *abstract implementations*, which are functionally equivalent but vastly simplified versions of the modules. The problem of checking a program then reduces to two tasks: checking that each program module behaves the same as its abstract implementation, and checking the program with its program modules replaced by their abstract implementations [12].

Extending traditional model checking to perform modular checking is trivial. For example, Java Pathfinder (JPF) [67] or CMC [51] can check that a program module and an abstract implementation behave the same on every sequence of inputs (within some finite bounds) by simply checking every reachable state (within those bounds).

However, it is nontrivial to extend glass box model checking to perform modular checking, while maintaining the significant performance advantage of glass box model checking over traditional model checking. In particular, it is nontrivial to extend glass box checking to check that a module and an abstract implementation behave the same on every sequence of inputs (within some finite bounds). This is because, unlike traditional model checkers such as Java Pathfinder or CMC, our model checker does not check every reachable state separately. Instead it checks a (usually very large) set of similar states in each single step. This chapter presents a technique to solve this problem.

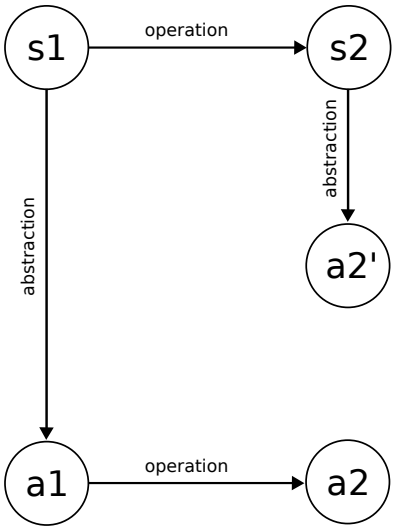


Figure 3.1: Glass box checking against an abstraction. Our modular technique checks that the outputs of executing the same operation on `s1` and `a1` are the same and the states `a2` and `a2′` are equal.

3.1 Example

Consider checking the Java program in Figure 3.2. This program tracks the frequency of integers received by its `count` method, storing the most frequent in its `most_frequent_i` field. It internally uses a map data structure, implemented as a binary search tree shown in Figure 3.4. Thus the program has two modules: `IntCounter` and `SearchTree`. Our modular approach checks each of these independently.

3.1.1 Abstraction

Our system first checks `SearchTree` against an abstract map implementation, and then uses the abstract map to check `IntCounter`. The abstract map must implement the `Map` interface, which includes the operations `insert` and `get`. (For brevity, this example omits other `Map` operations such as `delete`.) Figure 3.5 shows an `AbstractMap` implementation. It stores map entries in an unsorted list and uses a simple linear search algorithm to implement the map operations. `AbstractMap` is not an optimized implementation, but its simplicity makes it ideal as an abstraction for efficient software model checking. Using `AbstractMap` in place of `SearchTree` significantly improves the performance of our system. In fact, `AbstractMap` can be used in place of any data structure that implements the `Map` interface, including complex data

```
1  class IntCounter {
2      Map map = new SearchTree();
3      int max_frequency   = 0;
4      int most_frequent_i = 0;
5
6      public void count(int i) {
7          Integer frequency = (Integer)map.get(i);
8          if (frequency == null) frequency = new Integer(0);
9          map.insert(i, new Integer(frequency+1));
10
11         if (frequency >= max_frequency) {
12           max_frequency   = frequency;
13           most_frequent_i = i;
14         }
15     }
16
17     public int get_most_frequent_i() {
18           return most_frequent_i;
19     }
20
21     public int get_max_frequency() {
22           return max_frequency;
23     }
24 }
```

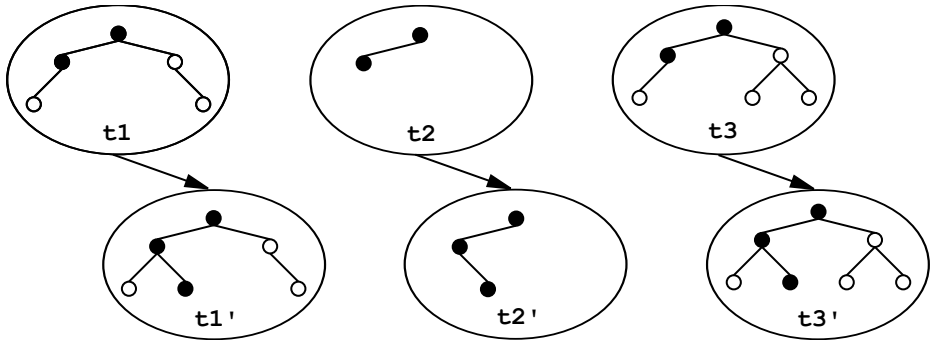
Figure 3.2: `IntCounter` internally using a `SearchTree`.

structures such as hash tables and red-black trees.

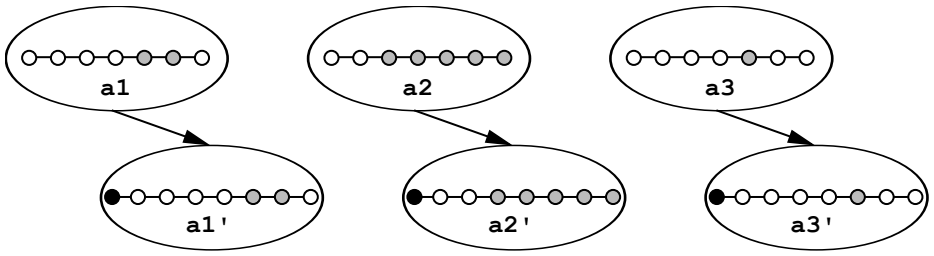
Note that `AbstractMap` uses a construct called `AbstractionList`. This is a linked list provided by our system that is useful in many abstract implementations. Using `AbstractionList` enables our system to arrange the list internally to achieve optimal performance during model checking. From the programmer's perspective, it is just a linked list data structure.

3.1.2 Checking the Abstraction

Our system checks that `SearchTree` behaves the same as `AbstractMap`. To do this it uses glass box checking to (in effect) exhaustively check every valid state of `SearchTree` within some given finite bounds against an equivalent state of `AbstractMap`. Figure 3.1 illustrates how we check that `SearchTree` and `AbstractMap` have the same behavior. We run the same operation on a `SearchTree` state `s1` and its abstraction `a1` to obtain states `s2` and `a2` respectively. We then check that (1) the abstraction of `s2` is equal to `a2`, and (2) the return values are same. Our system invokes the abstraction function to generate the abstractions of states `s1` and `s2`. The abstraction function



(a)



(b)

Figure 3.3: (a) Three search trees (code in Figure 3.4), before and after an `insert` operation, and (b) the corresponding abstract maps (code in Figure 3.5). The tree path touched by the operation is highlighted in each case. Note that the tree path is the same in all three cases. Once our system checks the `insert` operation on tree `t1`, it determines that it is redundant to check the same `insert` operation on trees `t2` and `t3`. The list nodes in gray correspond to tree nodes that are not reachable.

for `SearchTree` is in Figure 3.4. The method for testing equality of two `AbstractMap`s is shown in Figure 3.5.

Given a bound of 3 on the height of the tree, Figure 3.3(a) shows some possible states of `SearchTree`. Our system generates states of `AbstractMap` by calling an abstraction function. It creates a `AbstractionList` and passes it as an argument to the constructor of `AbstractMap`. Our system provides methods for generating `AbstractionList`s from several data structures, to make it convenient to implement abstraction functions. Behind the scenes, our system constructs a list long enough to hold the largest possible tree within the given bounds. Figure 3.3(b) shows the result of generating a few lists from trees. The list nodes in gray correspond to tree nodes that are not reachable. This arrangement facilitates the performance of the glass box model checking algorithm described in Section 2.3.

Consider checking an `insert` operation on state `t1` in Figure 3.3(a). After the operation, the resulting state is `t1'`. As described in Chapter II, our glass box checking identifies similar states (such as `t2` and `t3`) and prunes them all from the search space. The glass box analyses described in Sections 2.6 and 2.7 ensure that the presence of the abstract map does not increase the number of states that are explicitly checked.

3.1.3 Checking Using the Abstraction

Once our system establishes that `AbstractMap` and `SearchTree` have the same behavior, it uses `AbstractMap` instead of `SearchTree` to simplify the checking of `IntCounter`. For example, consider checking the invariant of `IntCounter`, that the fields `most_frequent_i` and `max_frequency` correspond to the most frequent integer in the map and its frequency, respectively. When checking `IntCounter`, our system substitutes `SearchTree` with `AbstractMap`. Otherwise, the checking proceeds as above. Our system repeatedly generates valid states of `IntCounter` (including its `AbstractMap`), identifies similar states, checks the similar states in a single step, and prunes them from its search space.

Using `AbstractMap` instead of `SearchTree` has several advantages. First, our state space reduction techniques are more effective on `AbstractMap`. In Figure 3.3, `a1`, `a2`, and `a3` are part of a larger set of similar states than `t1`, `t2`, and `t3` (w.r.t. the `insert` operation). Second, `AbstractMap` has a smaller state space to begin with. `SearchTree` encodes the shape of the tree in addition to key value pairs. More complex data structures such as red-black trees have even larger state spaces. Third, `AbstractMap` has a simpler invariant which translates to smaller formulas (as described in Section 2.9).

3.2 Specification

Given a program module `M`, programmers must first define an abstraction `A` which is functionally equivalent to `M` but is presumably simpler than `M`. However, note that an abstraction needs to be defined only once per interface and can be shared by all program modules that implement the same interface. For example, the `AbstractMap` defined in Figure 3.5 can be shared by all implementations of the `Map` interface including those that implement the map using an unbalanced binary tree (as in Figure 3.4), using a balanced binary tree such as a red-black tree, using a hash table, or using

```
1   class SearchTree implements Map {
2       static class Node implements AbstractionList.ListNodeSource {
3           int key;
4           Object value;
5           @Tree Node left;
6           @Tree Node right;
7
8           @Declarative
9           Node(int key, Object value) {
10              this.key = key;
11              this.value = value;
12          }
13
14          @Declarative
15          AbstractMap.Node abstraction() { return new AbstractMap.Node(key, value); }
16      }
17
18      @Tree Node root;
19
20      Object get(int key) {
21          Node n = root;
22          while (n != null) {
23              if (n.key == key)
24                  return n.value;
25              else if (key < n.key)
26                  n = n.left;
27              else
28                  n = n.right;
29          }
30          return null;
31      }
32
33      void insert(int key, Object value) {
34          Node n = root;
35          Node parent = null;
36          while (n != null) {
37              if (n.key == key) {
38                  n.value = value;
39                  return;
40              } else if (key < n.key) {
41                  parent = n;
42                  n = n.left;
43              } else {
44                  parent = n;
45                  n = n.right;
46              }
47          }
48
49          n = new Node(key, value);
50          if (parent == null)
51              root = n;
52          else if (key < parent.key)
53              parent.left = n;
54          else
55              parent.right = n;
56      }
57
58      @Declarative
59      boolean repOk() { return isOrdered(root, null, null); }
60
61      @Declarative
62      static boolean isOrdered(Node n, Node low, Node high) {
63          if (n == null) return true;
64          if (low  != null && low.key  >= n.key) return false;
65          if (high != null && high.key <= n.key) return false;
66          if !(isOrdered(n.left,  low, n   ))    return false;
67          if !(isOrdered(n.right, n,   high))    return false;
68          return true;
69      }
70
71      @Declarative
72      AbstractMap abstraction() {
73          return new AbstractMap(GlassBox.ListFromTree_BF(root));
74
75          // ListFromTree_BF returns an AbstractionList corresponding
76          // to a breadth first traversal of the tree.
77      }
78  }
```

Figure 3.4: A simple search tree implementation.

```
1   class AbstractMap implements Map {
2       static class Node {
3           Object key;
4           Object value;
5
6           Node(Object key, Object value) {
7               this.key = key;
8               this.value = value;
9           }
10
11          @Declarative
12          boolean equalTo(Node n) {
13              return n.key.equals(key) && n.value == value;
14          }
15      }
16
17      AbstractionList list;
18
19      @Declarative
20      AbstractMap(AbstractionList l) {
21          list = l;
22      }
23
24      Object get(Object key) {
25          AbstractionList.Node pnode = list.head();
26
27          while (pnode != null) {
28              Node n = (Node)pnode.data();
29              if (n.key.equals(key)) {
30                  return n.value;
31              } else {
32                  pnode = pnode.next();
33              }
34          }
35      }
36
37      void insert(Object key, Object value) {
38          AbstractionList.Node pnode = list.head();
39
40          while (pnode != null) {
41              Node n = (Node)pnode.data();
42              if (n.key.equals(key)) {
43                  n.value = value;
44                  return;
45              } else {
46                  pnode = pnode.next;
47              }
48          }
49
50          list.add(new Node(key, value));
51      }
52
53      @Declarative
54      public boolean equalTo(AbstractMap m) {
55          return list.equalTo(m.list);
56      }
57  }
```

Figure 3.5: An abstract map implementation.

a linked list. Every abstraction must also define an `equalTo` method to check if two instances of the abstraction are equivalent.

To check a program module `M` against an abstraction `A`, programmers must specify the invariant of `M`, an abstraction function that given an instance of `M` returns an equivalent instance of `A`, and finite bounds on the size of instances of `M`. For example, to check the binary search tree implementation in Figure 3.4 against the abstract map in Figure 3.5, programmers only need to specify the representation invariant of the search tree (`repOk` and `Tree` annotations), the abstraction function (`abstraction` in Figure 3.4), and finite bounds on the size of the search trees. Our system then checks that within the given bounded domain, the behavior of `M` is functionally equivalent to that of `A` on every sequence of inputs. Functional equivalence is defined in Section 3.4.

3.3 Modular Analysis

In our modular approach, the glass box search algorithm of Figure 2.2 is applied to the task of checking a module for equivalence to an abstraction. At each iteration of the algorithm, we apply the process depicted in Figure 3.1. For example, consider checking the binary search tree implementation in Figure 3.4 against the abstract map in Figure 3.5. We proceed as usual by choosing an unchecked `SearchTree` state `s1`. However, before running a transition on this state we use the abstraction function to generate a corresponding `AbstractMap` state `a1`. The abstraction function is defined as the declarative method `abstraction` of `SearchMap`. This method calls a declarative constructor of `AbstractMap`. We describe declarative constructors in Section 2.9.4.

Next we run the same operation on both `s1` and `a1` to yield states `s2` and `a2`, applying the dynamic analysis of Section 2.6. After generating `s2`, we apply the abstraction function to produce `a2'`, which corresponds to `s2`.

Finally, we use the static analysis described in Section 2.7 to check the invariant, along with the additional postcondition that `a2` is equal to `a2'`.

We express this process using the powerful specification mechanisms described in Chapter II. We present a driver for checking modules against abstractions in Figure 3.6. This driver checks that a class `Module` is functionally equivalent to a class `Abstraction` with respect to a method `transition`. The assertions in the method `transition` are checked after the transition, along with the invariant specified in `repOk`.

After we have checked functional equivalence of a module with its abstraction, we

```
1  class ModularDriver {
2      @Tree Module s;
3      @Tree Abstraction a;
4
5      void transition() {
6          a = s.abstraction();
7
8          Object result_s = s.transition();
9          Object result_a = a.transition();
10
11         assert(result_s == result_a);
12         assert(a.equalTo(s.abstraction()));
13     }
14
15     @Declarative
16     boolean repOk() { return s.repOk(); }
17 }
```

Figure 3.6: A driver for checking a module against an abstraction. We assume that a method `transition` calls corresponding methods of `Module` and `Abstraction`. The assertions contain declarative expressions that are checked along with `repOk`.

replace the module with the abstraction and check the rest of the program. Instead of specifying finite bounds on the module we specify finite bounds on the abstraction. The checking proceeds as described in Chapter II. Because the module has been replaced with a simple abstraction, the process takes significantly fewer iterations and less time per iteration than it would with the original module. (See Chapter VI.)

3.4 Checking Functional Equivalence

A module `M` is said to be functionally equivalent to an abstraction `A` if starting from an initial state of the module and the corresponding state of the abstraction, every sequence of operations on `M` and `A` produce the same outputs.

To check the functional equivalence between a module `M` and an abstraction `A` within some given finite bounds, we check the following two properties in those finite bounds.

The first property that we check is as follows. See Figure 3.7 for notation. We check that for every valid state `s1` of the module, that is, on every state `s1` on which `repOk` returns true: `s2.repOk()`, the outputs of executing the operation on `s1` and `a1` are the same, and `a2.equalTo(a2′)`.

The second property that we check is as follows. See Figure 3.7 for notation. We

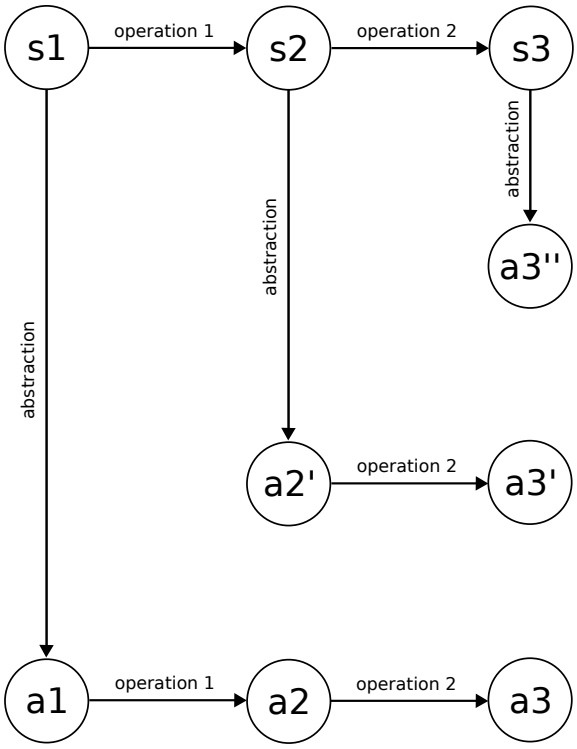


Figure 3.7: Operations on a module and its abstraction.

check that for every pair of states `a2` and `a2′` of the abstraction that are equal, that is, for every pair of states `a2` and `a2′` such that `a2.equalTo(a2′)`: the outputs of executing the same operation on `a2` and `a2′` are the same and the resulting states `a3` and `a3′` are equal, that is, `a3.equalTo(a3′)`. Checking this property checks that the `equalTo` method of the abstraction is implemented correctly with respect to the other methods in the abstraction.

The above two properties together imply functional equivalence, assuming that every initial state of the module satisfies `repOk`. Consider a sequence of two operations on a module state `s1`. See Figure 3.7 again for notation. Property 1 asserts that the outputs of executing the first operation on `s1` and `a1` are the same and that the outputs of executing the second operation on `s2` and `a2′` are the same. Property 1 also asserts that `a2.equalTo(a2′)`. Property 2 then asserts that the outputs of executing the second operation on `a2` and `a2′` are the same. Thus, together these properties assert that the outputs of executing the sequence of two operations on `s1` and `a1` are the same. Extending this argument to a sequence of operations of arbitrary length proves that the above two properties together imply functional equivalence.

We check the above two properties efficiently using the search algorithm described

in the above sections. To check the first property using the glass box search algorithm (see Figure 2.2), our system creates a bounded search space `B` consisting of all instances of the module within the given finite bounds. An element `s` of this search space is valid if `s.repOk()` returns true. To check the second property, our system creates a bounded search space `B` consisting of all pairs instances of the abstraction within the given finite bounds. An element `(a, a′)` of this search space is valid if `a.equalTo(a′)`. This is why we require `repOk` and `equalTo` to be declarative methods, so that they can be efficiently translated into boolean formulas during search space initialization.

3.5 Conclusions

This chapter presents a modular extension to glass box software model checking. Our system first checks a program module against an abstract implementation, establishing functional equivalence. It then replaces the program module with the abstract implementation when checking other program modules. We explain how we leverage the strengths of glass box software model checking to establish functional equivalence. Modular checking further improves the scalability of glass box software model checking when checking large programs composed of multiple modules.

CHAPTER IV

Glass Box Model Checking of Type Soundness

Type systems provide significant software engineering benefits. Types can enforce a wide variety of program invariants at compile time and catch programming errors early in the software development process. Types serve as documentation that lives with the code and is checked throughout the evolution of code. Types also require little programming overhead and type checking is fast and scalable. For these reasons, type systems are the most successful and widely used formal methods for detecting programming errors. Types are written, read, and checked routinely as part of the software development process. However, the type systems in languages such as Java, C#, ML, or Haskell have limited descriptive power and only perform compliance checking of certain simple program properties. But it is clear that a lot more is possible. There is therefore plenty of research interest in developing new type systems for preventing various kinds of programming errors [8, 17, 31, 53, 54, 69].

A formal proof of type soundness lends credibility that a type system does indeed prevent the errors it claims to prevent, and is a crucial part of type system design. At present, type soundness proofs are mostly done on paper, if at all. These proofs are usually long, tedious, and consequently error prone. There is therefore a growing interest in machine checkable proofs of soundness [2]. However, both the above approaches—proofs on paper (e.g., [22]) or machine checkable proofs (e.g., [56])—require significant manual effort.

This chapter presents an alternate approach for checking type soundness *automatically* using a glass box software model checker. Our idea is to systematically generate every type correct intermediate program state (within some finite bounds), execute the program one small step forward if possible using its small step operational semantics, and then check that the resulting intermediate program state is also type correct—but do so efficiently by using glass box model checking to detect similar

t	::=	`true`	*constant true*
	|	`false`	*constant false*
	|	`0`	*constant zero*
	|	`succ t`	*successor*
	|	`pred t`	*predecessor*
	|	`iszero t`	*zero test*
	|	`if t then t else t`	*conditional*

Figure 4.1: Abstract syntax of the language of integer and boolean expressions from [60, Chapters 3 & 8].

states and prune away large portions of the search space. Thus, given only a specification of type correctness and the small step operational semantics for a language, our system automatically checks type soundness by checking that the progress and preservation theorems [60, 71] hold for the language (albeit for program states of at most some finite size).

Note that checking the progress and preservation theorems on all programs states up to a finite size does not *prove* that the type system is sound, because the theorems might not hold on larger unchecked program states. However, in practice, we expect that all type system errors will be revealed by small sized program states. This conjecture, known as the *small scope hypothesis* [38], has been experimentally verified in several domains. Our experiments using mutation testing [59, 47] suggest that the conjecture also holds for checking type soundness. We also examined all the type soundness errors we came across in literature and found that in each case, there is a small program state that exposes the error. Thus, exhaustively checking type soundness on all programs states up to a finite size does at least generate a high degree of confidence that the type system is sound.

4.1 Example

This section illustrates our approach with an example. Consider the language of integer and boolean expressions in the book *Types and Programming Languages* [60, Chapters 3 & 8]. The syntax of the language is shown in Figure 4.1. The small step operational semantics and the type checking rules for this language are in [60]. To check type soundness, our system systematically generates and checks the progress and preservation theorems on every type correct program state within some finite bounds.

Figure 4.2 shows three abstract syntax trees (ASTs) `t1`, `t2`, and `t3`. AST `t1` rep-

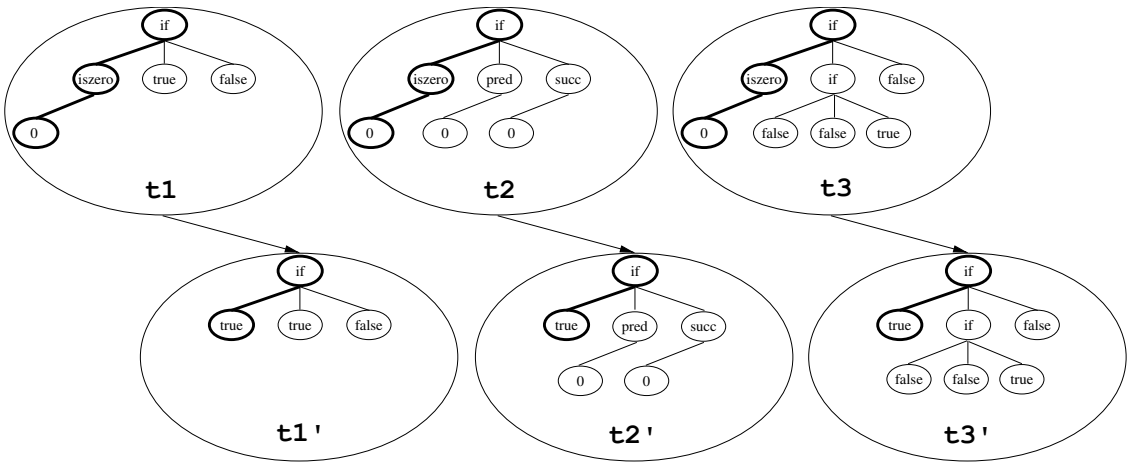


Figure 4.2: Three abstract syntax trees (ASTs) for the language in Figure 4.1, before and after a small step evaluation. The tree path touched by each evaluation is highlighted. Note that the tree path is the same in all three cases. Once our system checks the progress and preservation theorems on AST `t1`, it determines that it is redundant to check them on ASTs `t2` and `t3`.

resents the term '`if (iszero 0) then true else false`'. AST `t2` represents the term '`if (iszero 0) then (pred 0) else (succ 0)`'. AST `t3` represents the term '`if (iszero 0) then (if false then false else true) else false`'. Each of the ASTs is presented before and after a small step evaluation according to the small step operational semantics of the language.

Our state space reduction technique works as follows. As our system checks the progress and preservation theorems on `t1`, it detects that the small step evaluation of `t1` touches only a small number of AST nodes along a tree path in the AST. These nodes are highlighted in the figure. If these nodes remain unchanged, the small step evaluation will behave similarly (e.g., on ASTs such as `t2` and `t3`). Our system determines that it is redundant to check the progress and preservation theorems on ASTs such as `t2` and `t3` once it checks the theorems on `t1`. Our system safely prunes those program states from its search space, while still achieving complete test coverage within the bounded domain. Our system thus checks the progress and preservation theorems on every unique tree path (and some nearby nodes) rather than on every unique AST. Note that the number of unique ASTs of a given maximum height h is exponential in n, where $n = 3^h$, but the number of unique tree paths is only polynomial in n. This leads to significant reduction in the size of the search space and makes our approach feasible.

Our system performs even better if the operational semantics of the above language is implemented efficiently. For the example in Figure 4.2, our system detects

that only the nodes in the redex '`iszero 0`' matter, as long as that is the next redex to be reduced. It therefore prunes all program states where those nodes remain the same and that is the next redex to be reduced. This leads to even greater speedups. Our system then only checks $O(n)$ number of program states.

4.2 Specifying Language Semantics

To check the soundness of a type system, language designers only need to specify the small step operational semantics of the language, rules for checking type correctness of intermediate program states, and finite bounds on the size of intermediate program states. The operational semantics must be specified in an executable language to facilitate our dynamic analysis (see Section 2.6). The type system must be specified in a declarative language to facilitate our static analysis (see Section 2.7). The operational semantics may also be specified in a declarative language if 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 [45].

Figure 4.3 shows an example implementation of the expression language in Figure 4.1. An object of class `ExpressionLanguage` represents an intermediate program state of the expression language. Every such class that implements `Language` must have three methods: (i) a Java method `smallStep` that either performs a small step of evaluation and terminates normally, or throws an exception if the evaluation gets stuck; (ii) a declarative method `wellTyped` that returns true if and only if the corresponding intermediate program state is well typed; and (iii) another declarative method `isFinalState` that returns true if and only if the corresponding program state is fully evaluated. Declarative methods are annotated as `Declarative`, and are described in detail in Section 2.9.

We note that our model checking techniques are not tied to our above choice of specification language and can also be made to work with other languages (e.g., Ott [64]).

```
1   class ExpressionLanguage implements Language {
2       static final int TRUE   = 0;
3       static final int FALSE  = 1;
4       static final int ZERO   = 2;
5       static final int SUCC   = 3;
6       static final int PRED   = 4;
7       static final int ISZERO = 5;
8       static final int IF     = 6;
9       static final int BOOL   = 0;
10      static final int INT    = 1;
11
12      static class Expression {
13          int             kind;       /* TRUE / FALSE / ZERO / SUCC / PRED / ISZERO / IF */
14          @Tree Expression e1, e2, e3; /* Subexpressions                                    */
15
16          @Declarative
17          boolean wellTyped() {
18              if ( !syntaxOk() ) return false;
19              if ( kind == TRUE || kind == FALSE || kind == PRED   ) return true;
20              if ( kind == SUCC || kind == PRED  || kind == ISZERO )
21                      return e1.wellTyped() && e1.type() == INT;
22              if ( kind == IF )
23                      return e1.wellTyped() && e1.type() == BOOL
24                      && e2.wellTyped() && e3.wellTyped() && e2.type() == e3.type();
25              return false;
26          }
27
28          Expression smallStep() throws StuckException {
29              if ( e1 == null    ) {                     throw new StuckException(); }
30              if ( !e1.isValue() ) { e1 = e1.smallStep(); return this;                }
31
32              if ( kind == PRED   && e1.kind == ZERO  ) return e1;
33              if ( kind == PRED   && e1.kind == SUCC  ) return e1.e1;
34              if ( kind == ISZERO && e1.kind == ZERO  ) return True();
35              if ( kind == ISZERO && e1.kind == SUCC  ) return False();
36              if ( kind == IF     && e1.kind == TRUE  ) return e2;
37              if ( kind == IF     && e1.kind == FALSE ) return e3;
38
39              throw new StuckException();
40          }
41
42          // Helper functions
43
44          @Declarative
45          boolean syntaxOk() {
46              if ( kind == TRUE || kind == FALSE || kind == ZERO   )
47                      return e1 == null && e2 == null && e3 == null;
48              if ( kind == SUCC || kind == PRED  || kind == ISZERO )
49                      return e1 != null && e2 == null && e3 == null;
50              if ( kind == IF                                      )
51                      return e1 != null && e2 != null && e3 != null;
52              return false;
53          }
54
55          @Declarative
56          int type() {
57              if        ( kind == TRUE || kind == FALSE || kind == ISZERO )   return BOOL;
58              else if   ( kind == ZERO || kind == SUCC  || kind == PRED   )   return INT;
59              else    /*( kind == IF                                      )*/ return e2.type();
60          }
61
62          @Declarative
63          boolean isValue() {
64              return kind == TRUE || kind == FALSE || kind == ZERO || kind == SUCC && e1.isValue();
65          }
66
67          static Expression True () {Expression e = new Expression(); e.kind = TRUE;  return e;}
68          static Expression False() {Expression e = new Expression(); e.kind = FALSE; return e;}
69      }
70
71      @Tree Expression root;
72
73      @Declarative public boolean wellTyped()                         { return root.wellTyped(); }
74      @Declarative public boolean isFinalState()                      { return root.isValue();   }
75                    public void    smallStep()    throws StuckException { root = root.smallStep(); }
76  }
```

Figure 4.3: An implementation of the language in Figure 4.1.

```
1  class LanguageDriver {
2      @Tree Language language;
3
4      void transition() {
5          try {
6              language.smallStep();
7          } catch (StuckException e) {
8              assert(false);
9          }
10     }
11
12     @Declarative
13     boolean precondition() { return !language.isFinalState(); }
14
15     @Declarative
16     boolean repOk() { return language.wellTyped(); }
17 }
```

Figure 4.4: A driver for checking a language for type soundness. Checking the method `transition` with specification defined in methods `repOk` and `precondition` effectively checks the type soundness of `language` by checking that the progress and preservation theorems hold.

4.3 Glass Box Analysis

Given the specification of the language, we apply the glass box technique described in Chapter II. In order to check that the progress and preservation theorems hold for all well-typed states, we enumerate all well-typed program states, evaluate them over one small step, and ensure that no stuck exception was thrown and that the resulting program state is also well-typed. We need not check the final states of the program, since these implicitly satisfy the progress and preservation theorems.

We express this process using the powerful specification mechanisms described in Chapter II. We present a driver for checking languages for type soundness in Figure 4.4. This driver checks that a class implementing `Language` satisfies the progress and preservation theorems. Here we use the construct `assert(false)`, which is interpreted as an unconditional error. We also use a `precondition` method to exclude the final states of the language from consideration. See Section 2.11 for more information on these constructs.

```
1  class Node {
2      int kind;
3      int value;
4
5      @Tree Node left;
6      @Tree Node right;
7
8      @Declarative
9      Node copy() {
10         return new Node(this);
11     }
12
13     @Declarative
14     Node(Node n) {
15         kind = n.kind;
16         value = n.value;
17         if (n.left != null) left = n.left.copy();
18         if (n.right != null) right = n.right.copy();
19     }
20 }
```

Figure 4.5: A class that implements a declarative clone operation. A call to `copy` produces a deep copy of the node, such that the `left` and `right` fields are copied as well. This operation is declarative, so the shape of the subtree rooted at this node is not part of the path constraint.

4.4 Handling Special Cases

In addition to the standard analyses of glass box checking, our checker handles the following special cases.

Handling Term Cloning

Consider the following semantics for the `while` statement of the imperative language IMP from [70, Chapter 2], which clones the entire loop body.

$$\texttt{while c do b} \longrightarrow \texttt{if c then (b; while c do b)}$$

The cloning of different loop bodies could make `smallStep` follow different control flow paths. However, in one iteration of the glass box algorithm (See Figure 2.2), the symbolic execution described above only prunes states on which `smallStep` follows the same control flow path. To enable the pruning of program states with different loop bodies in the same iteration of the glass box algorithm, we implement term cloning using declarative methods and constructors. (See Section 2.9.4.) Figure 4.5

presents an example of such a declarative cloning operation.

Other examples of cloning include method calls that have a method inlining semantics (e.g., in Featherweight Java [34]).

Handling Substitution

Consider a language where method calls have a method inlining semantics. Suppose one small step of evaluation substitutes all the formals with actuals in the method body. Our model checker works best when each small step of evaluation reads only a small part of the program state. However, the above substitution reads the entire method body. Language designers can avoid the problem by defining the semantics of method calls using incremental substitution, where each small step of evaluation performs substitution on at most one AST node, and by ensuring that the type checking rules handle partially substituted program states.

Handling Nondeterministic Languages

The discussion so far assumes deterministic languages. Consider a language `L` with nondeterministic operational semantics. Its implementation in our system must include a *deterministic* method `smallStep` that takes an integer `x` as an argument, as shown below. If there are n transitions enabled on a given state, then `smallStep` must execute a different transition for each different value of `x` from 1 to n. Our system then checks that the progress and preservation theorems hold on every program state (within the finite bounds), with respect to every transition that is enabled on the state. For example, the following language nondeterministically chooses which of two threads to execute.

```
1  class L extends NondeterministicLanguage {
2      LThread thread1;
3      LThread thread2;
4      @Declarative public boolean wellTyped() {...}
5      @Declarative public boolean isFinalState() {...}
6      public void smallStep(int x) throws StuckException {
7          if (x == 0) thread1.smallStep();
8          else        thread2.smallStep();
9      }
10 }
```

4.5 Conclusions

This chapter presents a technique that *automatically* checks the soundness of a type system, given only the specification of type correctness of intermediate program states and the small step operational semantics. We adapt our glass box software model checking technique to perform this check. Currently, proofs of type soundness are either done on paper or are machine checked, but require significant manual assistance in both cases. Consequently proofs of type soundness are usually done *after* language design, if at all. Our system can be used *during* language design with little extra cost.

CHAPTER V

Formal Description

This chapter formalizes the core features of the glass box dynamic and static analyses for the simple Java-like language in Figure 5.1. This language resembles Featherweight Java [34] but it also includes imperative constructs such as assignments to a mutable heap. We assume that programs in this language have been type checked so that all field accesses and method calls are valid, except when accessed through a `null` pointer. Null pointer dereferencing is fatal in this language.

We define the values of the language as `true`, `false`, `null`, and all heap objects. Let a heap H be a mapping of objects and fields to values. We denote the value mapped by object obj and field f by $H(f, obj)$. We use the notation $H[(f, obj) \leftarrow \alpha]$ to refer to the heap H with the additional mapping of (f, obj) to the value α.

We define the small-step operational semantics of this language. For a heap H and expression e, each reduction of the form $\langle H, e\rangle \longrightarrow \langle H', e'\rangle$ modifies the heap to H' and reduces the expression to e'. For the purposes of the semantics, we extend the syntax to allow all objects on the heap to be expressions.

$$e ::= \ldots \mid obj$$

Figure 5.3 presents congruence rules for reducing subexpressions and Figure 5.4 presents rules for reducing the various syntactic forms. The rule R-Call uses a helper function *mbody*. For a method m and a heap object α, $mbody(m, \alpha)$ is equal to $\overline{x}.e$, where $\overline{x}$ is the sequence of formal parameters and e is the main expression of method m of object α.

We define a stuck state as a state $\langle H, e\rangle$ that can not be reduced by any of the rules, and where e is not a value. Stuck states represent runtime errors.

			e	::=	e ; e
				|	$e.f$
cn	:	class name		|	$e.f$ = e
m	:	method name		|	if e then e else e
f	:	field name		|	while e do e
x	:	variable name		|	new C
				|	$e.m(\overline{e})$
P	::=	$\overline{cd}$		|	x
cd	::=	class cn extends C $\{\overline{fd}\ \overline{md}\}$		|	this
C	::=	cn | Object		|	null
T	::=	C | boolean		|	true
fd	::=	$T\ f$;		|	false
md	::=	$T\ m(\overline{vd})\ \{e\}$		|	e && e
vd	::=	$T\ x$		|	e || e
				|	!e
				|	e == e

Figure 5.1: Syntax of a simple Java-like language. We write $\overline{cd}$ as shorthand for cd_1 cd_2 ... cd_n (without commas), write $\overline{vd}$ as shorthand for vd_1, vd_2, ..., vd_n (with commas), etc., similar to the notation in [34].

5.1 Symbolic Values and Symbolic State

We now define the notion of a symbolic state. We define a *symbolic value* as a set of elements of the form $c \to \alpha$, where c is a boolean formula and α is a concrete value as defined above. Informally, the formula c is a condition that must hold for the symbolic value to correspond to the concrete value α. For a given symbolic value $\{c_1 \to \alpha_1, c_2 \to \alpha_2, \ldots, c_n \to \alpha_n\}$, we require that all conditions c_i are pairwise unsatisfiable and mutually exhaustive, and that all concrete values α_i are distinct. We refer to this symbolic value as $\{\overline{c} \to \overline{\alpha}\}$, where $\overline{c} = c_1, c_2, \ldots, c_n$ and $\overline{\alpha} = \alpha_1, \alpha_2, \ldots, \alpha_n$. As a notational convenience, we use $\{\alpha\}$ as an abbreviation for the singleton set $\{\mathit{true} \to \alpha\}$, which is a symbolic value that unconditionally corresponds to a single concrete value α. Furthermore, we use $\{c \to \texttt{true}\}$ as an abbreviation for the set $\{c \to \texttt{true}, \neg c \to \texttt{false}\}$, which is a boolean symbolic value whose boolean concrete value depends to the formula c. For a symbolic value v and a concrete value α, let $v = \alpha$ denote c if $(c \to \alpha) \in v$ and *false* otherwise.

We define a *symbolic state* as a triple $\langle H, P, e \rangle$, where H is the symbolic heap, P is the current path constraint, and e is the current expression to evaluate. The symbolic heap H maps objects and fields to symbolic values. We use the notations $H(f, obj)$

$$
\begin{array}{rcl}
md & ::= & \texttt{@Declarative}\ T\ m(\overline{vd})\ \{de\} \\
\\
de & ::= & \texttt{if}\ de\ \texttt{then}\ de\ \texttt{else}\ de \\
& | & de.f \\
& | & de.m(\overline{de}) \\
& | & x \\
& | & \texttt{this} \\
& | & \texttt{null} \\
& | & \texttt{true} \\
& | & \texttt{false} \\
& | & de\ \texttt{\&\&}\ de \\
& | & de\ \texttt{||}\ de \\
& | & \texttt{!}de \\
& | & de\ \texttt{==}\ de
\end{array}
$$

Figure 5.2: Syntax of a declarative subset of the language in Figure 5.1, showing the syntax of declarative methods.

RC-SEQ

$$\frac{\langle H, e_0\rangle \longrightarrow \langle H', e_0'\rangle}{\langle H, e_0\ \texttt{;}\ e_1\rangle \longrightarrow \langle H', e_0'\ \texttt{;}\ e_1\rangle}$$

RC-FIELD-READ

$$\frac{\langle H, e_0\rangle \longrightarrow \langle H', e_0'\rangle}{\langle H, e_0.f\rangle \longrightarrow \langle H', e_0'.f\rangle}$$

RC-FIELD-WRITE

$$\frac{\langle H, e_0\rangle \longrightarrow \langle H', e_0'\rangle}{\langle H, e_0.f\ \texttt{=}\ e_1\rangle \longrightarrow \langle H', e_0'.f\ \texttt{=}\ e_1\rangle}$$

RC-FIELD-WRITE-2

$$\frac{\langle H, e_1\rangle \longrightarrow \langle H', e_1'\rangle}{\langle H, v_0.f\ \texttt{=}\ e_1\rangle \longrightarrow \langle H', v_0.f\ \texttt{=}\ e_1'\rangle}$$

RC-IF

$$\frac{\langle H, e_0\rangle \longrightarrow \langle H', e_0'\rangle}{\langle H, \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2\rangle \longrightarrow \langle H', \texttt{if}\ e_0'\ \texttt{then}\ e_1\ \texttt{else}\ e_2\rangle}$$

RC-CALL

$$\frac{\langle H, e_0\rangle \longrightarrow \langle H', e_0'\rangle}{\langle H, e_0.m(\overline{e})\rangle \longrightarrow \langle H', e_0'.m(\overline{e})\rangle}$$

RC-CALL-2

$$\frac{\langle H, e_i\rangle \longrightarrow \langle H', e_i'\rangle}{\langle H, v.m(v_0, \ldots, v_{i-1}, e_i, \ldots)\rangle \longrightarrow \langle H', v.m(v_0, \ldots, v_{i-1}, e_i', \ldots)\rangle}$$

RC-OP

$$\frac{\langle H, e_0\rangle \longrightarrow \langle H', e_0'\rangle}{\begin{array}{c}\langle H, e_0\ \texttt{\&\&}\ e_1\rangle \longrightarrow \langle H', e_0'\ \texttt{\&\&}\ e_1\rangle \\ \langle H, e_0\ \texttt{||}\ e_1\rangle \longrightarrow \langle H', e_0'\ \texttt{||}\ e_1\rangle \\ \langle H, \texttt{!}e_0\rangle \longrightarrow \langle H', \texttt{!}e_0'\rangle \\ \langle H, e_0\ \texttt{==}\ e_1\rangle \longrightarrow \langle H', e_0'\ \texttt{==}\ e_1\rangle \\ \langle H, v\ \texttt{==}\ e_0\rangle \longrightarrow \langle H', v\ \texttt{==}\ e_0'\rangle\end{array}}$$

Figure 5.3: Congruence reduction rules for the simple Java-like language in Figure 5.1.

R-SEQ

$$\frac{}{\langle H, v\ \texttt{;}\ e\rangle \longrightarrow \langle H, e\rangle}$$

R-FIELD-READ

$$\frac{v' = H(f, v) \quad v \neq \texttt{null}}{\langle H, v\texttt{.}f\rangle \longrightarrow \langle H, v'\rangle}$$

R-FIELD-WRITE

$$\frac{v_0 \neq \texttt{null}}{\langle H, v_0\texttt{.}f\ \texttt{=}\ v_1\rangle \longrightarrow \langle H[(f, v_0) \leftarrow v_1], v_1\rangle}$$

R-IF-T

$$\frac{}{\langle H, \texttt{if true then}\ e_1\ \texttt{else}\ e_2\rangle \longrightarrow \langle H, e_1\rangle}$$

R-IF-F

$$\frac{}{\langle H, \texttt{if false then}\ e_1\ \texttt{else}\ e_2\rangle \longrightarrow \langle H, e_2\rangle}$$

R-WHILE

$$\frac{}{\langle H, \texttt{while}\ e_0\ \texttt{do}\ e_1\rangle \longrightarrow \langle H, \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{;}\ \texttt{while}\ e_0\ \texttt{do}\ e_1\ \texttt{else false}\rangle}$$

R-NEW

$$\frac{\alpha \text{ is a fresh object of class } C \text{ with fields } \overline{f}.}{\langle H, \texttt{new}\ C\rangle \longrightarrow \langle H[(\overline{f}, \alpha) \leftarrow \{\texttt{null}\}], \alpha\rangle}$$

R-CALL

$$\frac{v \neq \texttt{null} \quad mbody(m, v) = \overline{x}.e}{\langle H, v\texttt{.}m(\overline{v})\rangle \longrightarrow \langle H, e[\overline{v}/\overline{x}, v/\texttt{this}]\rangle}$$

R-EQUALS-SAME

$$\frac{}{\langle H, v_0\ \texttt{==}\ v_0\rangle \longrightarrow \langle H, \texttt{true}\rangle}$$

R-EQUALS-DIFF

$$\frac{v_0 \neq v_1}{\langle H, v_0\ \texttt{==}\ v_1\rangle \longrightarrow \langle H, \texttt{false}\rangle}$$

R-AND-T

$$\frac{}{\langle H, \texttt{true \&\&}\ e\rangle \longrightarrow \langle H, e\rangle}$$

R-AND-F

$$\frac{}{\langle H, \texttt{false \&\&}\ e\rangle \longrightarrow \langle H, \texttt{false}\rangle}$$

R-OR-T

$$\frac{}{\langle H, \texttt{true ||}\ e\rangle \longrightarrow \langle H, \texttt{true}\rangle}$$

R-OR-F

$$\frac{}{\langle H, \texttt{false ||}\ e\rangle \longrightarrow \langle H, e\rangle}$$

R-NOT-T

$$\frac{}{\langle H, \texttt{!true}\rangle \longrightarrow \langle H, false\rangle}$$

R-NOT-F

$$\frac{}{\langle H, \texttt{!false}\rangle \longrightarrow \langle H, true\rangle}$$

Figure 5.4: Small-step operational semantics for the simple Java-like language in Figure 5.1.

and $H[(f, object) \leftarrow v]$ as usual for reading and modifying symbolic heaps.

We sometimes need to construct a symbolic value from a number of other symbolic values. Let $\overline{c} = c_1, c_2, \ldots, c_n$ be a sequence of n pairwise unsatisfiable and mutually exhaustive boolean formulas, and let $\overline{v} = v_1, v_2, \ldots, v_n$ be a sequence of n symbolic values. Then we define the symbolic value $\overline{c} \rightarrow \overline{v}$ as follows.

$$\overline{c} \rightarrow \overline{v} = \left\{ d \rightarrow \alpha \;\middle|\; d = \bigvee_{1 \leq i \leq n} c_i \wedge (v_i = \alpha), \text{ for } d \neq \mathit{false} \right\}$$

Informally, $\overline{c} \rightarrow \overline{v}$ selects the symbolic value v_i when the condition c_i holds. In addition, we define the following notation for building symbolic values from values on the heap.

$$\overline{c} \rightarrow H(f, \overline{\alpha}) = \overline{c} \rightarrow (H(f, \alpha_1), H(f, \alpha_2), \ldots, H(f, \alpha_n))$$

To facilitate dynamic semantics, we extend the expression syntax of Figure 5.1 to include symbolic values:

$$e ::= \ldots \mid v$$

Initially, the symbolic state is $\langle H_0, \mathit{true}, \{\alpha_0\}.m(\overline{v}) \rangle$, where H_0 is the initial symbolic heap of the finite search space, α_0 is the main object, m is a method to be run, and $\overline{v}$ are symbolic values of arguments to m. The initial heap H_0 contains symbolic values for each field of each object. Each symbolic value $v = \{\overline{c} \rightarrow \overline{\beta}\}$ defines a domain of n concrete values $\overline{\beta}$ (see Figure 2.1). The formulas $\overline{c}$ are each in terms of $\lceil \log n \rceil$ fresh boolean variables that mutually define a binary index into $\overline{\beta}$.

The informal correspondence between symbolic and concrete values is made explicit by assignments. An assignment is a map from the fresh boolean variables in H_0 to truth values. Consider an assignment Φ. For a boolean formula c, let $\Phi(c)$ denote the truth value of c when each boolean variable A is given the truth value $\Phi(A)$. For a symbolic value v, let $\Phi(v)$ denote the unique concrete value α such that $\Phi(v = \alpha)$.

5.2 Symbolic Execution

Figures 5.5 and 5.6 define the small-step operational semantics of symbolic execution. The reductions in Figure 5.5 are congruence rules for evaluating subexpressions. The reductions in Figure 5.6 define the rules for symbolic execution. Each conclusion of the form $\langle H, P, e \rangle \longrightarrow_\Phi \langle H', P', e' \rangle$ describes a transition in the presence of an

RCS-SEQ

$$\frac{\langle H, P, e_0\rangle \longrightarrow_\Phi \langle H', P', e_0'\rangle}{\langle H, P, e_0 \texttt{ ; } e_1\rangle \longrightarrow_\Phi \langle H', P', e_0' \texttt{ ; } e_1\rangle}$$

RCS-FIELD-READ

$$\frac{\langle H, P, e_0\rangle \longrightarrow_\Phi \langle H', P', e_0'\rangle}{\langle H, P, e_0.f\rangle \longrightarrow_\Phi \langle H', P', e_0'.f\rangle}$$

RCS-FIELD-WRITE

$$\frac{\langle H, P, e_0\rangle \longrightarrow_\Phi \langle H', P', e_0'\rangle}{\langle H, P, e_0.f \texttt{ = } e_1\rangle \longrightarrow_\Phi \langle H', P', e_0'.f \texttt{ = } e_1\rangle}$$

RCS-FIELD-WRITE-2

$$\frac{\langle H, P, e_1\rangle \longrightarrow_\Phi \langle H', P', e_1'\rangle}{\langle H, P, v_0.f \texttt{ = } e_1\rangle \longrightarrow_\Phi \langle H', P', v_0.f \texttt{ = } e_1'\rangle}$$

RCS-IF

$$\frac{\langle H, P, e_0\rangle \longrightarrow_\Phi \langle H', P', e_0'\rangle}{\langle H, P, \texttt{if } e_0 \texttt{ then } e_1 \texttt{ else } e_2\rangle \longrightarrow_\Phi \langle H', P', \texttt{if } e_0' \texttt{ then } e_1 \texttt{ else } e_2\rangle}$$

RCS-CALL

$$\frac{\langle H, P, e_0\rangle \longrightarrow_\Phi \langle H', P', e_0'\rangle}{\langle H, P, e_0.m(\overline{e})\rangle \longrightarrow_\Phi \langle H', P', e_0'.m(\overline{e})\rangle}$$

RCS-CALL-2

$$\frac{\langle H, P, e_i\rangle \longrightarrow_\Phi \langle H', P', e_i'\rangle}{\langle H, P, v.m(v_0, \ldots, v_{i-1}, e_i, \ldots)\rangle \longrightarrow_\Phi \langle H', P', v.m(v_0, \ldots, v_{i-1}, e_i', \ldots)\rangle}$$

RCS-OP

$$\frac{\langle H, P, e_0\rangle \longrightarrow_\Phi \langle H', P', e_0'\rangle}{\begin{array}{c}\langle H, P, e_0 \texttt{ \&\& } e_1\rangle \longrightarrow_\Phi \langle H', P', e_0' \texttt{ \&\& } e_1\rangle \\ \langle H, P, e_0 \texttt{ || } e_1\rangle \longrightarrow_\Phi \langle H', P', e_0' \texttt{ || } e_1\rangle \\ \langle H, P, \texttt{!}e_0\rangle \longrightarrow_\Phi \langle H', P', \texttt{!}e_0'\rangle \\ \langle H, P, e_0 \texttt{ == } e_1\rangle \longrightarrow_\Phi \langle H', P', e_0' \texttt{ == } e_1\rangle \\ \langle H, P, v \texttt{ == } e_0\rangle \longrightarrow_\Phi \langle H', P', v \texttt{ == } e_0'\rangle\end{array}}$$

Figure 5.5: Congruence reduction rules of symbolic execution for the simple Java-like language in Figure 5.1.

RS-SEQ

$$\frac{}{\langle H, P, v\ \texttt{;}\ e\rangle \longrightarrow_\Phi \langle H, P, e\rangle}$$

RS-FIELD-READ

$$\frac{\begin{array}{c} v = \{\overline{c} \to \overline{\alpha}\} \\ v' = \overline{c} \to H(f, \overline{\alpha}) \\ \Phi(v) \neq \texttt{null} \end{array}}{\langle H, P, v\,\texttt{.}\,f\rangle \longrightarrow_\Phi \langle H, P \wedge \neg(v = \texttt{null}), v'\rangle}$$

RS-FIELD-WRITE

$$\frac{\Phi(v_0) = \alpha \quad \alpha \neq \texttt{null}}{\langle H, P, v_0\,\texttt{.}\,f\ \texttt{=}\ v_1\rangle \longrightarrow_\Phi \langle H[(f, \alpha) \leftarrow v_1], P \wedge (v_0 = \alpha), v_1\rangle}$$

RS-IF-T

$$\frac{\Phi(v) = \texttt{true}}{\langle H, P, \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2\rangle \longrightarrow_\Phi \langle H, P \wedge (v = \texttt{true}), e_1\rangle}$$

RS-IF-F

$$\frac{\Phi(v) = \texttt{false}}{\langle H, P, \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2\rangle \longrightarrow_\Phi \langle H, P \wedge (v = \texttt{false}), e_2\rangle}$$

RS-WHILE

$$\frac{}{\langle H, P, \texttt{while}\ e_0\ \texttt{do}\ e_1\rangle \longrightarrow_\Phi \langle H, P, \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{;}\ \texttt{while}\ e_0\ \texttt{do}\ e_1\ \texttt{else}\ \{\texttt{false}\}\rangle}$$

RS-NEW

$$\frac{\alpha \text{ is a fresh object of class } C \text{ with fields } \overline{f}.}{\langle H, P, \texttt{new}\ C\rangle \longrightarrow_\Phi \langle H[(\overline{f}, \alpha) \leftarrow \{\texttt{null}\}], P, \{\alpha\}\rangle}$$

RS-CALL

$$\frac{\Phi(v) = \alpha \quad \alpha \neq \texttt{null} \quad mbody(m, \alpha) = \overline{x}.e}{\langle H, P, v\,\texttt{.}\,m(\overline{v})\rangle \longrightarrow_\Phi \langle H, P \wedge (v = \alpha), e[\overline{v}/\overline{x}, v/\texttt{this}]\rangle}$$

RS-EQUALS

$$\frac{\begin{array}{c} v_0 = \{\overline{c} \to \overline{\alpha}\} \quad v_1 = \{\overline{d} \to \overline{\beta}\} \\ v' = \{\bigvee\{c_i \wedge d_i \mid \alpha_i = \beta_i\} \to \texttt{true}\} \end{array}}{\langle H, P, v_0\ \texttt{==}\ v_1\rangle \longrightarrow_\Phi \langle H, P, v'\rangle}$$

RS-AND-T

$$\frac{\Phi(v) = \texttt{true}}{\langle H, P, v\ \texttt{\&\&}\ e\rangle \longrightarrow_\Phi \langle H, P \wedge (v = \texttt{true}), e\rangle}$$

RS-AND-F

$$\frac{\Phi(v) = \texttt{false}}{\langle H, P, v\ \texttt{\&\&}\ e\rangle \longrightarrow_\Phi \langle H, P \wedge (v = \texttt{false}), \{\texttt{false}\}\rangle}$$

RS-OR-T

$$\frac{\Phi(v) = \texttt{true}}{\langle H, P, v\ \texttt{||}\ e\rangle \longrightarrow_\Phi \langle H, P \wedge (v = \texttt{true}), \{\texttt{true}\}\rangle}$$

RS-OR-F

$$\frac{\Phi(v) = \texttt{false}}{\langle H, P, v\ \texttt{||}\ e\rangle \longrightarrow_\Phi \langle H, P \wedge (v = \texttt{false}), e\rangle}$$

RS-NOT

$$\frac{\begin{array}{c} v = \{c \to \texttt{true}\} \\ v' = \{\neg c \to \texttt{true}\} \end{array}}{\langle H, P, \texttt{!}v\rangle \longrightarrow_\Phi \langle H, P, v'\rangle}$$

RS-TRUE

$$\frac{}{\langle H, P, \texttt{true}\rangle \longrightarrow_\Phi \langle H, P, \{true \to \texttt{true}\}\rangle}$$

RS-FALSE

$$\frac{}{\langle H, P, \texttt{false}\rangle \longrightarrow_\Phi \langle H, P, \{false \to \texttt{true}\}\rangle}$$

RS-NULL

$$\frac{}{\langle H, P, \texttt{null}\rangle \longrightarrow_\Phi \langle H, P, \{\texttt{null}\}\rangle}$$

Figure 5.6: Small-step operational semantics of symbolic execution. The symbolic state includes a heap H, a path constraint P, and an expression e. An assignment Φ is required for converting symbolic values v into concrete values α.

RD-FIELD-READ

$$\frac{H \vdash de \Downarrow (v, E) \quad v = \{\overline{c} \to \overline{\alpha}\} \quad v' = \overline{c} \to H(f, \overline{\alpha})}{H \vdash de.f \Downarrow (v', E \vee v = \texttt{null})}$$

RD-IF

$$\frac{H \vdash de_0 \Downarrow (\{b \to \texttt{true}\}, E_0) \quad H \vdash de_1 \Downarrow (v_1, E_1) \quad H \vdash de_2 \Downarrow (v_2, E_2) \quad v = (b, \neg b) \to (v_1, v_2) \quad E = E_0 \vee (b \wedge E_1) \vee (\neg b \wedge E_2)}{H \vdash \texttt{if}\ de_0\ \texttt{then}\ de_1\ \texttt{else}\ de_2 \Downarrow (v, E)}$$

RD-CALL

$$\frac{H \vdash de_0 \Downarrow (v_0, E_0) \quad v_0 = \{\overline{c} \to \overline{\alpha}\} \quad mbody(m, \alpha_i) = \overline{x}^i.de_i' \quad \forall i : H \vdash de_i \Downarrow (v_i, E_i) \quad H \vdash de_i'[\overline{v}/\overline{x}^i, \{\alpha_i\}/\texttt{this}] \Downarrow (v_i', E_i') \quad E = E_0 \vee \bigvee \overline{E} \vee (v_0 = \texttt{null}) \vee \bigvee_i (c_i \wedge E_i')}{H \vdash de_0.m(\overline{de}) \Downarrow (\overline{c} \to \overline{v'}, E)}$$

RD-EQUALS

$$\frac{H \vdash de_0 \Downarrow (\{\overline{c} \to \overline{\alpha}\}, E_0) \quad H \vdash de_1 \Downarrow (\{\overline{d} \to \overline{\beta}\}, E_1) \quad v = \{\bigvee_{i,j} \{c_i \wedge d_j \mid \alpha_i = \beta_j\} \to \texttt{true}\}}{H \vdash de_0\ \texttt{==}\ de_1 \Downarrow (v, E_0 \vee E_1)}$$

RD-AND

$$\frac{H \vdash de_0 \Downarrow (\{c \to \texttt{true}\}, E_0) \quad H \vdash de_1 \Downarrow (\{d \to \texttt{true}\}, E_1) \quad v = \{c \wedge d \to \texttt{true}\}}{H \vdash de_0\ \texttt{\&\&}\ de_1 \Downarrow (v, E_0 \vee (c \wedge E_1))}$$

RD-OR

$$\frac{H \vdash de_0 \Downarrow (\{c \to \texttt{true}\}, E_0) \quad H \vdash de_1 \Downarrow (\{d \to \texttt{true}\}, E_1) \quad v = \{c \vee d \to \texttt{true}\}}{H \vdash de_0\ \texttt{||}\ de_1 \Downarrow (v, E_0 \vee (\neg c \wedge E_1))}$$

RD-NOT

$$\frac{H \vdash de \Downarrow (\{c \to \texttt{true}\}, E) \quad v = \{\neg c \to \texttt{true}\}}{H \vdash \texttt{!}de \Downarrow (v, E)}$$

RD-VALUE

$$\frac{}{H \vdash v \Downarrow (v, false)}$$

RD-TRUE

$$\frac{}{H \vdash \texttt{true} \Downarrow (\{true \to \texttt{true}\}, false)}$$

RD-FALSE

$$\frac{}{H \vdash \texttt{false} \Downarrow (\{false \to \texttt{true}\}, false)}$$

RD-NULL

$$\frac{}{H \vdash \texttt{null} \Downarrow (\{\texttt{null}\}, false)}$$

Figure 5.7: Big-step operational semantics of declarative methods, used in their translation to formulas. Given a heap H an expression e evaluates to a value v with an error condition E. E holds true for concrete states that encounter an error.

assignment Φ. We say that a symbolic state $\langle H, P, e\rangle$ is a symbolic stuck state in Φ when e is not a symbolic value and none of the reductions apply to this state using Φ.

The rules describe how expressions evaluate to symbolic values, and how expressions change the heap and the path constraint. For example, the rule RS-Field-Read evaluates expressions of the form $v.f$, where $v = \{\overline{c} \to \overline{\alpha}\}$ is a symbolic value and f is a field. For each non-null α_i in the sequence $\overline{\alpha}$, the symbolic value $H(f, \alpha_i)$ is the result of accessing field f through the object α_i. The result v' combines all such symbolic values into one result. The rule requires that $\Phi(v)$, the concrete evaluation of v, is not `null`. If $\Phi(v)$ is `null`, then no evaluation rule applies and this is a symbolic stuck state in Φ. Otherwise, if $\Phi(v)$ is not `null`, then this fact is added to the path constraint.

The rule RS-Call performs method calls by inlining a method's body at the call site and substituting formal parameters with their actual symbolic values. Recall that the function $mbody(m, \alpha)$ denotes $\overline{x}\,.\,e$, where e is the body of the method m of object α and $\overline{x}$ is the list of formal method parameters. We use $e[v/x]$ to denote the expression e with all instances of variable x replaced with symbolic value v.

The rules RS-If-T and RS-If-F for the `if` expressions depend on the concrete value of the branch condition v. If the concrete value is `true`, then $v =$ `true` is added to the path constraint. If the concrete value is `false`, then $v =$ `false` is added to the path constraint. Similarly, the rules for the operators `&&` and `||` generate path constraints.

The reason the rules for the `if` expressions and the `&&` and `||` operators generate path constraints is that these expressions and operators short circuit and their operands might have side effects. For example, `(true &&` e`)` evaluates e but `(false &&` e`)` does not and e might have side effects.

However, if the operands of `if` expressions and the `&&` and `||` operators do not have any side effects then they can be executed symbolically without generating path constraints. We describe this process in Section 5.4.

5.3 Translation of Declarative Methods

Figure 5.2 presents a declarative subset of the language in Figure 5.1, showing the syntax of declarative methods. Declarative methods may not contain object creations, assignments, or loops and may only call declarative methods. This corresponds

to the core declarative syntax presented in Section 2.9. Accordingly, additional syntax features such as assignment to local variables and iteration structures may be implemented by first translating to the core syntax.

Figure 5.7 presents the big-step operational semantics of declarative methods that are used to translate them into formulas. Declarative methods do not have assignments or object creations, so they do not modify the heap H. Unlike non-declarative methods, the semantics of declarative methods do not depend on concrete values, so they do not need an assignment Φ. Furthermore, branches in declarative methods do not generate path constraints, so the semantics of declarative methods do not use a path constraint. As with the non-declarative expressions above, we extend the syntax of declarative expressions to include symbolic values:

$$de ::= \ldots \mid v$$

Judgments are of the form $H \vdash e \Downarrow (v, E)$, indicating that under heap H, an expression e translates to a value v with an error condition E. The error condition E is a formula that holds true for concrete states that encounter an error.

The result of calling a declarative method is a symbolic value and error condition (v, E). In the case of boolean methods (such as `repOk`), v is of the form $\{c \rightarrow \texttt{true}\}$ where $c \wedge \neg E$ holds for states where the method successfully returns `true`. Thus, this process translates a boolean declarative method into a formula that describes the conditions under which the method returns `true`.

5.4 Symbolic Execution of Declarative Expressions

Recall from Figure 5.6 that the symbolic execution of the short circuiting operators `&&` and `||` generates path constraints because of the possibility of side effects. However, the operands of these operators often do not have side effects. In such cases, these operators execute symbolically without generating path constraints. The same applies to `if` statements. In general, strictly declarative expressions symbolically execute without generating path constraints (except for the exception condition) according to the following rule.

RS-DECL

$$\frac{\begin{array}{c}e \text{ has declarative syntax} \\ H \vdash e \Downarrow (v, E) \\ \Phi(E) = \mathit{false}\end{array}}{\langle H, P, e\rangle \longrightarrow \langle H, P \wedge \neg E, v\rangle}$$

Require: H is a symbolic heap and α is a checkable object in H.
1: **procedure** GlassBoxSearch(H, α)
2: $S \leftarrow s \wedge \neg E$, where $H \vdash \{\alpha\}$`.repOk()` $\Downarrow (\{s \rightarrow$ `true`$\}, E)$
3: **while** S is satisfiable **do**
4: $\Phi \leftarrow$ any satisfying assignment of S
5: $\langle H', P, e\rangle \leftarrow$ symbolic execution of $\langle H,$ `true`$, \{\alpha\}$`.transition()`$\rangle$ with Φ
6: **if** $\langle H', P, e\rangle$ is a stuck state **then**
7: **return** false
8: **end if**
9: $R \leftarrow r \wedge \neg E$, where $H' \vdash \{\alpha\}$`.repOk()` $\Downarrow (\{r \rightarrow$ `true`$\}, E)$
10: **if** $S \wedge P \wedge \neg R$ is satisfiable **then**
11: **return** false
12: **end if**
13: $S \leftarrow S \wedge \neg P$
14: **end while**
15: **return** true
16: **end procedure**

Figure 5.8: The glass box search algorithm as applied to the formalism of a simple Java-like language.

A simple static analysis determines if an expression has declarative syntax. By requiring $\Phi(E)$ to be false, the above rule only applies when no errors are encountered. The final path constraint reflects this requirement.

5.5 The Glass Box Algorithm

Figure 5.8 presents the glass box algorithm in terms of the formalism above. For simplicity, we assume that we are checking that a method called `transition` maintains the invariant defined in a declarative method called `repOk`. We consider an object *checkable* when it provides these methods. The symbolic heap H defines the search bounds and the checkable object α is exhaustively tested up to these bounds.

We now prove the correctness of the algorithm. In preparation, we introduce the following notation. Let $\longrightarrow^*$ be the transitive and reflexive closure of the concrete transition operator $\longrightarrow$. Given an assignment Φ, let $\longrightarrow^*_\Phi$ be the transitive and reflexive closure of the symbolic transition operator $\longrightarrow_\Phi$. For a symbolic heap H, let H_Φ be the concrete heap defined by the composition $\Phi \circ H$. Thus, H_Φ is the heap H with symbolic values replaced with concrete values. For a symbolic expression e, let e_Φ be the concrete expression generated by replacing every symbolic value v that

appears as a subexpression of e with $\Phi(v)$. Note that according to this definition, $v_\Phi = \Phi(v)$.

The correctness of the algorithm depends on the soundness of declarative translation and symbolic execution with respect to the concrete semantics of the language. We formalize this soundness in the following theorems.

Theorem 1 (Soundness of Declarative Translation)**.** *If $H \vdash e \Downarrow (v, E)$ then for all assignments Φ,*

(i) if $\Phi(E)$ then there exists no value v' such that $\langle H_\Phi, e_\Phi \rangle \longrightarrow^* \langle H_\Phi, v' \rangle$, and

(ii) if $\neg\Phi(E)$ then $\langle H_\Phi, e_\Phi \rangle \longrightarrow^* \langle H_\Phi, v_\Phi \rangle$.

Theorem 2 (Soundness of Symbolic Execution)**.** *For all assignments Φ, if there exists a chain of symbolic transitions such that $\langle H, true, e \rangle \longrightarrow^*_\Phi \langle H', P, e' \rangle$ then*

(i) if $\langle H', P, e' \rangle$ is a symbolic stuck state in Φ then $\langle H'_\Phi, e'_\Phi \rangle$ is a stuck state,

(ii) $\Phi(P)$ holds, and

(iii) for all assignments Ψ, if $\Psi(P)$ holds then $\langle H_\Psi, e_\Psi \rangle \longrightarrow^* \langle H'_\Psi, e'_\Psi \rangle$.

Proving these theorems requires a straightforward enumeration of the declarative and symbolic semantic rules defined above. The proofs are included at the end of this chapter for completeness.

We prove the correctness of the GlassBoxSearch algorithm with the following theorem, which states that the algorithm returns true exactly when the invariant is maintained for all states in the finite bounds.

Theorem 3 (Correctness of the Glass Box Algorithm)**.** *When it terminates, the* GlassBoxSearch *algorithm exits with a return value of true if and only if the following property holds for every assignment Ψ: If $\langle H_\Psi, \alpha\texttt{.repOk()} \rangle$ evaluates to $\langle H_\Psi, \texttt{true} \rangle$ then $\langle H_\Psi, \alpha\texttt{.transition()} \rangle$ evaluates to $\langle h, v \rangle$ for some concrete heap h and some concrete value v, where $\langle h, \alpha\texttt{.repOk()} \rangle$ evaluates to $\langle h, \texttt{true} \rangle$.*

Proof. For an assignment Ψ and concrete heap h, we define the following propositions.

$OK(h)$: $\langle h, \alpha\texttt{.repOk()}\rangle \longrightarrow^* \langle h, \texttt{true}\rangle$

Pre(Ψ): OK(H_Ψ)

Trans(Ψ, h): $\langle H_\Psi, \alpha\texttt{.transition()}\rangle \longrightarrow^* \langle h, v\rangle$ for some value v

Post(Ψ): Trans(Ψ, h) and OK(h) for some heap h

Then the theorem states that when it terminates, GlassBoxSearch returns true if and only if $\forall\Psi(\text{Pre}(\Psi) \rightarrow \text{Post}(\Psi))$. Note that because the language semantics are determinate, for a given Ψ there will be at most one h such that Trans(Ψ, h).

After Line 2, we establish that for every assignment Ψ, $\Psi(S)$ holds if and only if OK(H_Ψ). The forward implication follows immediately from Theorem 1(ii), and the inverse implication follows from Theorem 1(i) when $\Psi(E)$ holds and from Theorem 1(ii) when $\neg\Psi(s)$ holds, along with the knowledge that the concrete semantics are determinate.

Throughout each iteration of the main loop in Lines 3-14, we show that the following invariant holds:

Invariant: For every assignment Ψ,

(i) if $\Psi(S)$ then Pre(Ψ), and

(ii) if $\neg\Psi(S)$ then Pre(Ψ) $\rightarrow$ Post(Ψ).

The invariant holds at the start of the first iteration, by the established property of S above. At the end of the last iteration, S is not satisfiable, which means that $\neg\Psi(S)$ holds for all Ψ. Thus the invariant establishes the forward implication of the theorem statement, since the only place where the algorithm returns true is immediately after the loop exits.

It remains to show that the invariant is maintained across iterations, and that the inverse implication holds.

Suppose the invariant holds at the start of an iteration. At Line 4 we find Φ such that $\Phi(S)$ holds. Next, at Line 5, we use symbolic execution to find $\langle H', P, e\rangle$ such that $\langle H, \texttt{true}, \{\alpha\}\texttt{.transition()}\rangle \longrightarrow^*_\Phi \langle H', P, e\rangle$. By Theorem 2(iii) we conclude that for all Ψ where $\Psi(P)$ holds (including Φ), $\langle H_\Psi, \{\alpha\}\texttt{.transition()}\rangle \longrightarrow^* \langle H'_\Psi, e_\Psi\rangle$.

If we reached a stuck state during symbolic execution then by Theorem 2(i), the state $\langle H_\Phi, \{\alpha\}\texttt{.transition()}\rangle$ does not reduce to a value, so Post(Φ) does not hold. However, Pre(Φ) holds by the invariant(i). Therefore, Pre(Φ) $\rightarrow$ Post(Φ) does not hold. If this is the case, the algorithm detects it on Line 6 and returns false.

Otherwise, no stuck state was reached during symbolic execution, and e is a value. Thus for all Ψ, $\text{Trans}(\Psi, H'_\Psi)$ holds when $\Psi(P)$ holds. At Line 9 we define R as $r \wedge \neg E$, where $H' \vdash \{\alpha\}\texttt{.repOk()} \Downarrow (\{r \to \texttt{true}\}, E)$. As above, we observe using Theorem 1 that for every assignment Ψ, $\Psi(R)$ holds if and only if $\text{OK}(H'_\Psi)$.

Then at Line 10 we check to see if there exists Ψ such that $\Psi(S) \wedge \Psi(P) \wedge \neg\Psi(R)$. If such a Ψ exists, we establish $\text{Pre}(\Psi)$ from invariant(i). From $\Psi(P)$ we know that $\text{Trans}(\Psi, H'_\Psi)$ holds. However, we know from $\neg\Psi(R)$ that $\text{OK}(H'_\Psi)$ does not hold. There can be no other h such that $\text{Trans}(\Psi, h)$ holds, so $\text{Post}(\Psi)$ does not hold. Therefore, $\text{Pre}(\Psi) \to \text{Post}(\Psi)$ does not hold. If this is the case, the algorithm returns false.

Otherwise, no such Ψ exists, and we conclude that for all Ψ, if $\Psi(S)$ and $\Psi(P)$ hold then $\Psi(R)$ holds as well. Thus, for all Ψ such that $\Psi(S)$ and $\Psi(P)$, it is true that $\text{Pre}(\Psi) \to \text{Post}(\Psi)$. Therefore, when the algorithm updates S to $S \wedge \neg P$, the invariant is maintained.

Finally, note that if the algorithm returns false, either at Line 7 or at Line 11, we found Ψ such that $\text{Pre}(\Psi) \to \text{Post}(\Psi)$ does not hold. Therefore the inverse implication in the theorem statement holds. This concludes the proof. □

Due to the inherent undecidability of the problem of analyzing program behavior, the algorithm may not always terminate. However, the following theorem shows that the algorithm always terminates under the assumption that the symbolic execution and declarative translation steps terminate.

Theorem 4 (Conditional Termination of the Glass Box Algorithm)**.** *If all symbolic execution and declarative translation operations terminate then* GlassBoxSearch *terminates.*

Proof. If symbolic execution and declarative translation terminate then the algorithm must terminate as long as there is a bounded number of loop iterations. We show that the number of satisfying assignments of S strictly decreases across iterations, so the number of iterations is bounded by the finite number of assignments.

As established in the proof of Theorem 3, Φ is a satisfying assignment of both S and P. Thus, Φ is not a satisfying assignment of $S \wedge \neg P$. Therefore, the number of satisfying assignments of S decreases by at least one in every iteration. This concludes the proof. □

5.6 Proofs of Theorem 1 and Theorem 2

Theorem 1 (Soundness of Declarative Translation)**.** *If* $H \vdash e \Downarrow (v, E)$ *then for all assignments* Φ*,*

(i) if $\Phi(E)$ *then there exists no value* v' *such that* $\langle H_\Phi, e_\Phi \rangle \longrightarrow^* \langle H_\Phi, v' \rangle$*, and*

(ii) if $\neg\Phi(E)$ *then* $\langle H_\Phi, e_\Phi \rangle \longrightarrow^* \langle H_\Phi, v_\Phi \rangle$*.*

Proof. Let Φ be an arbitrary assignment. We proceed by induction on the derivation of the judgment $H \vdash e \Downarrow (v, E)$. For each rule in Figure 5.7, we assume that the theorem holds for all judgments in the premises and show that the theorem also holds for the judgment in the conclusion. We use the notation from Figure 5.7.

Consider the rule RD-Value, where $H \vdash v \Downarrow (v, false)$. Then (i) vacuously holds and (ii) is immediate from the fact that $\longrightarrow^*$ is reflexive.

Consider the rules RD-True, RD-False, and RD-Null. Then (i) vacuously holds and (ii) is immediate from the definition of Φ.

Consider the rule RD-Field-Read, where $H \vdash de.f \Downarrow (\overline{c} \to H(f, \overline{\alpha}), E \vee v = \texttt{null})$. If $\Phi(E \vee v = \texttt{null})$ then either $\Phi(E)$ or $\Phi(v = \texttt{null})$. Suppose that $\Phi(E)$. Then by the induction hypothesis $\langle H_\Phi, de_\Phi \rangle$ does not reduce to a value. So $\langle H_\Phi, (de.f)_\Phi \rangle$ does not reduce to a value either, and the theorem holds. Next, suppose that $\neg\Phi(E)$. By the induction hypothesis $\langle H_\Phi, de_\Phi \rangle \longrightarrow^* \langle H_\Phi, v_\Phi \rangle$ and thus $\langle H_\Phi, de.f_\Phi \rangle \longrightarrow^* \langle H_\Phi, v_\Phi.f \rangle$. If $\Phi(v = \texttt{null})$ then $v_\Phi = \texttt{null}$ and the evaluation above leads to the stuck state $\langle H_\Phi, \texttt{null}.f \rangle$, so the theorem holds. If $\neg\Phi(v = \texttt{null})$ then $\langle H_\Phi, v_\Phi.f \rangle \longrightarrow \langle H_\Phi, H_\Phi(f, v_\Phi) \rangle$. Note that $\Phi(\overline{c} \to H(f, \overline{\alpha}))$ is equal to $H_\Phi(f, v_\Phi)$, so the theorem holds.

Consider the rule RD-If, where $H \vdash \texttt{if}\ de_0\ \texttt{then}\ de_1\ \texttt{else}\ de_2 \Downarrow (v, E)$. If $\Phi(E_0)$ then by the induction hypothesis $\langle H_\Phi, de_{0\Phi} \rangle$ does not reduce to a value, so the theorem holds. Suppose then that $\neg\Phi(E_0)$, so $\langle H_\Phi, de_{0\Phi} \rangle \longrightarrow^* \langle H_\Phi, \{b \to \texttt{true}\}_\Phi \rangle$. If either $\Phi(b \wedge E_1)$ or $\Phi(\neg b \wedge E_2)$, then the corresponding expression $de_{1\Phi}$ or $de_{2\Phi}$ does not reduce to a value, and the `if` expression reduces to one of these stuck states. Suppose that $\neg\Phi(b \wedge E_1)$ and $\neg\Phi(\neg b \wedge E_2)$. Then, depending on $\Phi(b)$, the original `if` expression reduces to $v_{1\Phi}$ or $v_{2\Phi}$. Observe that this is equal to $v_\Phi = \Phi((b, \neg b) \to (v_1, v_2))$, so the theorem holds.

Consider the rule RD-Call, where $H \vdash de_0.m(\overline{de}) \Downarrow (\overline{c} \to \overline{v'}, E)$. The expression does not reduce to a value in the following cases.

- $\Phi(E_0)$: The method target de_0 does not reduce to a value.
- $\Phi(E_i)$ for some i : The method parameter de_i does not reduce to a value.
- $\Phi(v_0 = \texttt{null})$: The expression reduces to $\texttt{null}.m(\overline{v_\Phi})$, which can not be reduced.
- $\Phi(c_i \wedge E_i')$ for some i : The body of the called method de_i' does not reduce to a value.

If none of the above cases hold then, using the induction hypothesis, the expression reduces to $v_{0\Phi}.m(\overline{v_\Phi})$ for $v_{0\Phi} \neq \texttt{null}$. Thus, it reduces to $de_i'[\overline{v_\Phi}/\overline{x}^i, \{\alpha_i\}/\texttt{this}]$ for some i, which by induction hypothesis reduces to $v_{i\Phi}'$. Observe that this is equal to $(\overline{c} \to \overline{v'})_\Phi$ because $\Phi(c_i)$ holds.

Consider the rule RD-Equals. If either $\Phi(E_0)$ or $\Phi(E_1)$ then by the induction hypothesis either $de_{0\Phi}$ or $de_{1\Phi}$ does not reduce to a value, so the expression does not reduce to a value. If neither $\Phi(E_0)$ nor $\Phi(E_1)$ then by the induction hypothesis, $de_{0\Phi} \longrightarrow^* \{\overline{c} \to \overline{\alpha}\}_\Phi$ and $de_{1\Phi} \longrightarrow^* \{\overline{d} \to \overline{\beta}\}_\Phi$. The equality expression then reduces to `true` if and only if these two reduced values are equal. Thus the result is `true` exactly when $\Phi(c_i)$ and $\Phi(d_j)$ for $\alpha_i = \beta_j$. This is the same as v_Φ.

Consider the rule RD-And. If $\Phi(E_0)$ then $de_{0\Phi}$ does not reduce to a value, so $(de_0 \&\& de_1)_\Phi$ does not reduce to a value. Otherwise, $(de_0 \&\& de_1)_\Phi \longrightarrow^* \{c \to \texttt{true}\}_\Phi \&\& de_1$. Suppose that $\neg\Phi(c)$. Then $(de_0 \&\& de_1)_\Phi \longrightarrow^* \texttt{false}$, which is equal to v_Φ under the assumption that $\neg\Phi(c)$. Next, suppose that $\Phi(c)$, so $de_0 \&\& de_1 \longrightarrow^* de_1$. If $\Phi(E_1)$ then this expression does not reduce to a value. Otherwise, it reduces to $\{d \to \texttt{true}\}_\Phi$, which is equal to v_Φ under the assumption that $\Phi(c)$.

The rule RD-Or is similar to the above.

Consider the rule RD-Not. If $\Phi(E)$ then de_Φ does not evaluate to a value, so neither does $!de_\Phi$. Otherwise, $!de_\Phi \longrightarrow^* !\{c \to \texttt{true}\}_\Phi$, which reduces to `true` if and only if $\neg\Phi(c)$. This is equal to $\{\neg c \to \texttt{true}\}_\Phi$, so the theorem holds. □

Toward a proof of Theorem 2, we first establish the following lemma.

Lemma 1. *For all assignments Φ, if $\langle H, P, e\rangle \longrightarrow_\Phi \langle H', P', e'\rangle$ then*

(i) if $\Phi(P)$ then $\Phi(P')$,

(ii) for all assignments Ψ, if $\Psi(P')$ then $\Psi(P)$, and

(ii) for all assignments Ψ, if $\Psi(P')$ then $\langle H_\Psi, e_\Psi\rangle \longrightarrow^* \langle H'_\Psi, e'_\Psi\rangle$.

Proof. We prove by structural induction on the expression e. That is, we assume that the lemma holds for all subexpressions of e and show that it holds for e as well. We do this by considering each possible application of the symbolic semantic rules.

First, we show that property (i) holds. This is trivial for rules RS-Seq, RS-While, RS-New, RS-Equals, RS-True, RS-False, and RS-Null. Furthermore, the property is immediate for the remaining rules from Figure 5.6, $P' = P \wedge b$, where $\Phi(b)$ holds by premise. It is also immediate for rule RS-Decl. Finally, we note that property (i) follows immediately from the induction hypothesis for the congruence rules of Figure 5.5.

Next we show property (ii) for an arbitrary assignment Ψ such that $\Psi(P')$ holds. Note that $\Psi(P)$ holds by inspection for the rules in Figure 5.6, along with RS-Decl. For the congruence rules, (ii) follows immediately from the induction hypothesis.

Next we show property (iii) for an arbitrary assignment Ψ such that $\Psi(P')$ holds.

The rules RS-True, RS-False, and RS-Null trivially correspond to a reflexive application of $\longrightarrow^*$. The rules RS-Seq, RS-While, and RS-New correspond to single applications of the rules R-Seq, R-While, and R-New by inspection. Also by inspection, the rules RS-And-T and RS-Or-T correspond to single applications of the rules R-And-T and R-Or-T given that $\Psi(v) = \texttt{false}$. Likewise for RS-And-F and RS-Or-F for $\Psi(v) = \texttt{false}$.

Consider the rule RS-Field-Read. Given that $\Psi(\neg(v = \texttt{null})$ holds, apply rule R-Field-Read to show that $\langle H_\Psi, v_\Psi.f\rangle \longrightarrow \langle H_\Psi, H_\Psi(f, v_\Psi)\rangle$. The final expression is the same as $(\overline{c} \rightarrow H(f, \overline{\alpha}))_\Psi$.

Consider the rule RS-Field-Write. Given that $\Psi(v_0) = \alpha$ and $\alpha \neq \texttt{null}$, apply rule R-Field-Write to show that $\langle H_\Psi, v_{0\Psi}.f \texttt{ = } v_{1\Psi}\rangle \longrightarrow \langle H_\Psi[(f, v_{0\Psi}) \leftarrow v_{1\Psi}], v_{1\Psi}\rangle$. Note that the final heap state is the same as $H[(f, \alpha) \leftarrow v_1]_\Psi$.

Consider the rule RS-If-T. Given that $\Psi(v) = \texttt{true}$, apply rule R-If-T to show that $\langle H_\Psi, \texttt{if true then } e_{1\Psi} \texttt{ else } e_{2\Psi}\rangle \longrightarrow \langle H_\Psi, e_{1\Psi}\rangle$.

Consider the rule RS-If-F. Given that $\Psi(v) = \texttt{false}$, apply rule R-If-F to show that $\langle H_\Psi, \texttt{if false then } e_{1\Psi} \texttt{ else } e_{2\Psi}\rangle \longrightarrow \langle H_\Psi, e_{2\Psi}\rangle$.

Consider the rule RS-Call. Then we have $\Psi(v) = \alpha$ and $\alpha \neq \texttt{null}$, as well as the fact that $mbody(m, \alpha) = \overline{x}.e$. Apply rule R-Call to show that $\langle H_\Psi, v_\Psi.m(\overline{v_\Psi})\rangle \longrightarrow \langle H_\Psi, e[\overline{v_\Psi}/\overline{x}, v_\Psi/\texttt{this}]\rangle$. Note that the final expression is the same as $e[\overline{v}/\overline{x}, v/\texttt{this}]_\Psi$.

Consider the rule RS-Equals. Suppose that $\Psi(v_0) = \Psi(v_1) = \alpha_i = \beta_j$ for some i and j. Then $v'_\Psi = \texttt{true}$ by the definition of v'. Apply rule R-Equals-Same to show that $\langle H_\Psi, v_{0\Psi} \texttt{ == } v_{1\Psi}\rangle \longrightarrow \langle H, \texttt{true}\rangle$. Now suppose that $\Psi(v_0) = \alpha_i \neq Phi(v_1) = \beta_j$

for some i and j. Then $v'_\Psi = \texttt{false}$ by the definition of v'. Apply rule R-EQUALS-DIFF.

Next consider the rule RS-DECL. Then we have $H \vdash e \Downarrow (v, E)$ and $\Psi(E) = false$. Applying Theorem 1, we find that $\langle H_\Psi, e_\Psi \rangle \longrightarrow^* \langle H_\Psi, v_\Psi \rangle$.

Finally, consider the congruence rules. Each of these rules has a premise of the form $\langle H, P, e \rangle \longrightarrow_\Phi \langle H', P', e' \rangle$ for some subexpression e. By induction hypothesis, we have $\langle H_\Psi, e_\Psi \rangle \longrightarrow^* \langle H'_\Psi, e'_\Psi \rangle$. Then an application of the corresponding concrete congruence rule from Figure 5.3 yields the desired result.

This concludes the proof. □

Theorem 2 (Soundness of Symbolic Execution)**.** *For all assignments Φ, if there exists a chain of symbolic transitions such that $\langle H, true, e \rangle \longrightarrow^*_\Phi \langle H', P, e' \rangle$ then*

(i) if $\langle H', P, e' \rangle$ is a symbolic stuck state in Φ then $\langle H'_\Phi, e'_\Phi \rangle$ is a stuck state,

(ii) $\Phi(P)$ holds, and

(iii) for all assignments Ψ, if $\Psi(P)$ holds then $\langle H_\Psi, e_\Psi \rangle \longrightarrow^* \langle H'_\Psi, e'_\Psi \rangle$.

Proof. Let Φ be an arbitrary assignment.

We establish (i) by proving the stronger property that if any symbolic state $\langle H, P, e \rangle$ is a symbolic stuck state then $\langle H_\Phi, e_\Phi \rangle$ is a stuck state. Assuming that all the usual type errors are eliminated by the type system, we have the following possible symbolic stuck states:

- $\langle H, P, \texttt{null}.f \rangle$
- $\langle H, P, \texttt{null}.f = v \rangle$
- $\langle H, P, \texttt{null}.m(\overline{v}) \rangle$

If $\langle H, P, e \rangle$ is one of the above states then $\langle H_\Phi, e_\Phi \rangle$ is clearly a stuck state.

We prove (ii) and (iii) by induction on the length n of the chain of symbolic reductions from $\langle H, \text{true}, e \rangle$ to $\langle H'_n, P_n, e'_n \rangle$. As the basis of the inductive argument, consider that (ii) and (iii) hold trivially for a chain consisting of a single state, $\langle H, \text{true}, e \rangle$. We next assume that (ii) and (iii) hold for chains of length n and show that they must also hold for chains of length $n + 1$, for $n \geq 1$. For property (ii), this follows immediately from Lemma 1(i) and the induction hypothesis.

Next, let Ψ be an assignment such that $\Psi(P_{n+1})$ holds. By Lemma 1(ii), it must also be the case that $\Psi(P_n)$ holds. Thus by the induction hypothesis, $\langle H_\Psi, e_\Psi \rangle \longrightarrow^* \langle H'_{n\Psi}, e'_{n\Psi} \rangle$. By Lemma 1(iii), $\langle H'_{n\Psi}, e'_{n\Psi} \rangle \longrightarrow^* \langle H'_{(n+1)\Psi}, e'_{(n+1)\Psi} \rangle$. Thus, property (iii) holds. This concludes the proof. □

CHAPTER VI

Experimental Results

This chapter presents experimental results evaluating our glass box model checking technique. We implemented the glass box checking system described in Chapter II as well as the modular extension described in Chapter III and the extension for checking type soundness described in Chapter IV. Our implementation is written in Java and checks Java programs, but other programming languages would work as well. We extended the Polyglot [58] compiler framework to automatically instrument program modules to perform our dynamic analysis. We used MiniSat [24] as our incremental SAT solver to perform our static analysis. We ran all our experiments on a Linux Fedora Core 8 machine with a Pentium 4 3.4 GHz processor and 1 GB memory using IcedTea Java 1.7.0.

We evaluate our system in three phases. In Section 6.1 we use the basic glass box checking implementation to check a variety of data structure invariants. In Section 6.2 we use the modular extension to glass box checking to check some programs composed of modules. In Section 6.3 we use the extension for checking type soundness to check the soundness of several type systems.

6.1 Checking Data Structures

To evaluate the effectiveness of glass box model checking, we used our system to check the invariants of several data structures. We implemented each invariant as a declarative `repOk` method. We checked the following data structures:

- `Stack`, a stack implemented with a linked list. The invariant asserts that the list is acyclic. We check the `push` and `pop` operations.
- `Queue`, a queue implemented with two stacks, a front stack and a back stack. Items are enqueued onto the back stack and dequeued from the front stack.

The invariant asserts that the stacks are non-null, and that their invariants also hold. We check the `enqueue` and `dequeue` operations.

- `TreeMap`, a red-black tree from the Java Collections Framework. The invariants are the usual red-black tree invariants that guarantee ordering and balancing properties, plus the additional requirement that the `size` field accurately reflects the size of the tree. We check the `put`, `remove`, `get`, `isEmpty`, and `clear` methods.

- `HashMap`, a hash table from the Java Collections Framework. The invariant asserts that each key in the hash table is unique and is in the correct bin and that the `size` field accurately reflects the size of the tree. We check the `put`, `remove`, `get`, `isEmpty`, and `clear` methods.

`TreeMap` and `HashMap` are from the source code of the Java SE 6 JDK with Java generics removed. Although generics do not pose any difficulties to our technique, the Polyglot compiler framework does not fully support them.

We compare our glass box checker with the following state-of-the-art model checkers:

- **Java Pathfinder**

  Java Pathfinder (JPF) is a stateful model checker for Java [67]. We use JPF to exhaustively check all possible operations starting with an initially empty data structure.

- **Korat**

  Korat is a system for efficient test case generation [5]. We supply Korat with our class invariant and have it generate all test cases up to our finite bounds.

- **Blast**

  Blast is a model checker that uses predicate abstraction based on Counter Example Guided Abstraction Refinement (CEGAR) [32]. Due to its imprecise alias analysis, Blast is unable to complete any of the checks we tested. We therefore do not show results for our tests using Blast. This illustrates the challenge of checking complex data-oriented properties.

We used our system and the above model checkers to check all operations on all states of these data structures up to a maximum of n nodes with at most n different possible keys and eight different possible values. In `Stack` and `Queue`, we exclude the

Data Structure	Max Number of Nodes	JPF		Korat		Glass Box	
		Transitions	Time (s)	Transitions	Time (s)	Transitions	**Time (s)**
	1	28	0.539	4	0.275	2	**0.050**
	2	36	0.551	6	0.270	2	**0.049**
	3	44	0.564	8	0.273	2	**0.051**
	4	52	0.571	10	0.271	2	**0.053**
	...	...	...	...	...	...	...
`Stack`	8	84	0.622	18	0.276	2	**0.074**
	16	148	0.733	34	0.295	2	**0.088**
	32	276	0.952	66	0.312	2	**0.105**
	64	532	1.690	130	0.362	2	**0.154**
	128	1044	4.604	258	0.528	2	**0.266**
	256	2068	17.138	Timeout		2	**0.730**
	512	4116	743.233			2	**2.846**
	1024	Timeout				2	**15.977**
	1	180	0.714	8	0.395	3	**0.066**
	2	792	0.997	18	0.391	4	**0.075**
	3	6144	2.704	32	0.397	5	**0.083**
	4	71768	22.993	50	0.400	6	**0.087**
	5	Memory Out		72	0.410	7	**0.109**
	6			98	0.409	8	**0.116**
	7			128	0.415	9	**0.125**
`Queue`	8			162	0.424	10	**0.137**
	...			...	...	...	...
	16			578	0.474	18	**0.230**
	32			2178	0.666	34	**0.419**
	64			8450	3.089	66	**0.915**
	128			33282	38.728	130	**3.420**
	256			Timeout		258	**23.406**
	512					514	**199.202**
	1024					1026	**2489.931**
	1	210	0.526	10	0.668	14	**0.171**
	2	4880	1.182	35	0.625	24	**0.238**
	3	194130	24.020	120	0.643	46	**0.430**
	4	Memory Out		465	0.719	58	**0.505**
	5			2070	0.885	86	**0.689**
	6			8655	1.638	98	**0.775**
	7			31460	4.445	132	**0.950**
	8			241885	123.615	144	**1.028**
	9			1137890	698.132	184	**1.351**
`TreeMap`	10			5149115	3699.716	198	**1.478**
	11			Timeout		256	**1.967**
	12					270	**2.099**
	13					313	**2.500**
	14					326	**2.661**
	15					396	**3.372**
	...					...	...
	31					1052	**18.956**
	63					2748	**240.928**
	127					6644	**5726.031**
	1	245	1.025	15	0.506	10	**0.171**
	2	3000	1.799	50	0.431	15	**0.210**
	3	94095	38.028	165	0.503	20	**0.241**
	4	Memory Out		530	0.526	27	**0.279**
	5			1665	0.556	36	**0.332**
	6			5150	0.626	45	**0.387**
	7			15765	0.687	54	**0.430**
	8			68520	1.446	60	**0.512**
	9			297765	3.566	66	**0.551**
`HashMap`	10			1293720	11.427	72	**0.633**
	11			5619765	40.493	78	**0.721**
	12			24406920	149.523	84	**0.857**
	13			105981765	571.363	90	**1.068**
	14			460129120	2236.581	96	**1.361**
	15			Timeout		102	**1.651**
	16					108	**2.271**
	...					...	...
	32					204	**55.115**
	64					396	**1748.281**

Table 6.1: Results of checking data structure invariants.

`pop` and `dequeue` operations when the corresponding data structure is empty, since in this case the methods' preconditions are not met. We aborted all experiments after two hours, or after running out of memory.

The results of these experiments are tabulated in Table 6.1. For each given maximum number of nodes, we present results for checking exhaustively up to those bounds. For each model checker, we show the number of transitions explicitly checked, as well as the time it takes to perform the check.

Our system checks `Stack` very quickly because all calls to `push` and `pop` behave similarly, regardless of the size of the `Stack`. Thus we check the `push` and `pop` operations only once each, and prune the rest of the transitions. The analysis to safely prune these transitions becomes more complex for larger stacks, even though the number of transitions does not increase. We check `Queue` efficiently as well, only explicitly checking a number of transitions linear in the maximum number of nodes. Java Pathfinder manages to check the simple stack up to 512 nodes, but the extra complexity of the Queue causes it to quickly run out of memory. Korat checks both the Stack and the Queue up to t node bound of 128 before suddenly encountering a state space explosion and running out of time.

Our system achieves good performance when checking the more complex data structures as well. We exhaustively checked all `TreeMap`s with up to 63 nodes in under five minutes and all `HashMap`s with up to 64 nodes in under 30 minutes. In contrast, Java Pathfinder and Korat quickly encounter a state space explosion and are forced to abort. These results indicate that the state space reduction of glass box model checking is effective.

6.2 Modular Checking

Using our modular extension to glass box checking, we tested modules that implement the `Map` and `Set` interfaces from the Java Collections Framework. We created two abstract implementations, `AbsMap` and `AbsSet` and tested several modules for conformance to these implementations. Our `AbsMap` implementation is similar to `AbstractMap` from Figure 3.5. We tested the `Map` interface on the methods `put`, `remove`, `get`, `isEmpty`, and `clear` and the `Set` interface on the methods `add`, `remove`, `contains`, `isEmpty`, and `clear`. All the Java Collections Framework modules are from the source code of the Java SE 6 JDK. Although Java generics do not pose any difficulties to our technique, the Polyglot compiler framework does not fully support

Module	Max Number of Nodes	Time (s)
TreeMap	1	0.188
	2	0.244
	3	0.392
	4	0.485
	5	0.670
	6	0.751
	7	0.985
	8	1.124
	9	1.491
	10	1.670
	11	2.303
	12	2.555
	13	3.142
	14	3.435
	15	4.571
	...	...
	31	40.405
	63	787.411
HashMap	1	0.176
	2	0.227
	3	0.258
	4	0.305
	5	0.373
	6	0.422
	7	0.494
	8	0.599
	9	0.675
	10	0.780
	11	0.953
	12	1.162
	13	1.356
	14	1.708
	15	2.143
	16	2.879
	...	...
	32	75.139
	64	2004.723

Module	Max Number of Nodes	Time (s)
TreeSet	1	0.195
	2	0.246
	3	0.393
	4	0.489
	5	0.651
	6	0.752
	7	0.961
	8	1.090
	9	1.473
	10	1.665
	11	2.238
	12	2.493
	13	3.065
	14	3.399
	15	4.481
	...	...
	31	39.789
	63	796.955
HashSet	1	0.171
	2	0.221
	3	0.248
	4	0.299
	5	0.363
	6	0.414
	7	0.478
	8	0.579
	9	0.644
	10	0.773
	11	0.948
	12	1.068
	13	1.344
	14	1.608
	15	2.029
	16	2.816
	...	...
	32	68.011
	64	2543.034

Table 6.2: Results of checking modules against abstractions. We check `TreeMap`s of up to 63 nodes in under 15 minutes.

them. So we removed them from the source. We tested the following modules:

- `TreeMap`, which implements the `Map` interface using a red-black tree, which is a balanced binary tree.
- `TreeSet`, which implements the `Set` interface using an underlying `TreeMap`
- `HashMap`, which implements the `Map` interface using a hash table.
- `HashSet`, which implements the `Set` interface using an underlying `HashMap`

We used our system to exhaustively check module states up to a maximum of n nodes, with at most n different possible keys and eight different possible values. We checked the functional equivalence (see Section 3.4) between the above modules and their abstractions.

The results of these experiments are presented in Table 6.2. We exhaustively checked all `TreeMap`s with up to 63 nodes in under 15 minutes and all `HashMap`s with up to 64 nodes in under 40 minutes.

Next, we tested the effectiveness of replacing the maps with the abstract map. We checked the following programs:

- `TreeSet` and `HashSet`. Since they use a `Map` internally, they can be checked modularly. We again checked functional equivalence between these modules and `AbsSet`.
- `IntCounter` from Figure 3.2, implemented with a `TreeMap`. We checked that the fields `most_frequent_i` and `max_frequency` are consistent with the state of the map.
- A two-layer cache, `DualCache`, similar to the one described in Section 2.9, implemented using two `TreeMap`s. One map is the *permanent* map and the other is the *temporary* map. `DualCache` has some internal consistency constraints, such as the property that no key can be in both maps at once. We checked the following operations:
  - `lookup` : Looks up a value in the cache. If the value is not present, it computes it and adds it to the cache.
  - `remove` : Removes a value from the cache, if it exists.
  - `enableTemporary` : Causes future cache additions to go to the temporary map.
  - `disableTemporary` : Clears the temporary map and causes future cache additions to go to permanent map.
  - `promote` : Removes a value from the temporary map and adds it to the permanent map.
  - `demote` : Removes a value from the permanent map and adds it to the temporary map.

We used our system to check the given implementations of these programs. We then replaced the maps with abstract maps and checked the programs again. We checked maps with at most n nodes. We checked `IntCounter` with at most n integers

Module	Max Number of Nodes	Time (s)	
		Original Program	Modules Replaced with Abstract Implementations
`TreeSet`	1	0.195	0.122
	2	0.246	0.153
	3	0.393	0.167
	4	0.489	0.185
	5	0.651	0.207
	6	0.752	0.246
	7	0.961	0.251
	...	...	...
	15	4.481	0.429
	31	39.789	0.991
	63	796.955	3.388
	127	Timeout	16.690
	255		184.827
	511		425.328
`HashSet`	1	0.171	0.106
	2	0.221	0.141
	3	0.248	0.153
	4	0.299	0.169
	5	0.363	0.193
	6	0.414	0.218
	7	0.478	0.238
	...	...	...
	15	2.029	0.412
	16	2.816	0.451
	32	68.011	0.989
	64	2543.034	3.464
	128	Timeout	17.071
	256		91.629
	512		754.426
`IntCounter`	1	0.198	0.113
	2	0.279	0.154
	3	0.469	0.164
	4	0.579	0.184
	5	0.815	0.214
	6	0.918	0.251
	7	1.182	0.267
	...	...	...
	15	5.591	0.539
	31	41.500	1.867
	63	632.488	10.794
	127	Timeout	93.276
	255		946.091
`DualCache`	1	0.203	0.222
	2	0.283	0.323
	3	0.503	0.327
	4	0.589	0.511
	5	0.828	0.654
	6	0.950	0.548
	7	1.207	0.529
	...	...	...
	15	5.765	1.015
	31	53.434	4.057
	63	723.267	26.192
	127	Timeout	215.521
	255		2180.506

Table 6.3: Results of checking programs that use a map internally. Replacing the map with an abstract implementation speeds up the checking considerably.

and frequencies ranging from 0 to 7. We checked `DualCache` with at most n keys and at most eight values.

The results of these experiments are presented in Table 6.3. Checking these programs with `AbsMap` is significantly faster than checking them with a `TreeMap` or a `HashMap`.

6.3 Checking Type Soundness

This section presents results of the extension to glass box model checking for checking type soundness. We present results for the following languages, each with increasing complexity:

1. The language of integer and boolean expressions from [60, Chapters 3 & 8], as implemented in Figure 4.3.

2. A typed version of the imperative language IMP from [70, Chapter 2].

   This language contains integer and boolean variables, so its type checking rules include an environment context. This language also contains `while` statements.

3. An object-oriented language Featherweight Java [34].

   This language has classes, objects, and methods. The semantics of method calls require term level substitution (of the formal method parameters with their actual values).

4. An extension to Featherweight Java we call Mini Java.

   This language models the heap explicitly, supports mutations to objects in the heap, and includes a `null` value. This language also contains integers and booleans, and operations on integers and booleans.

5. An extension to Mini Java to support ownership types [1, 7, 11], that we call Ownership Java.

   This language has classes parameterized by owner parameters. Therefore the semantics of a method call require both term level and type level substitution.

For each benchmark, we checked the progress and preservation theorems *exhaustively* on all program states up to a maximum size n. In all languages, we limited

Benchmark	Max Expression Size	**States Checked**	Time (s)
Expression Language	1	**1**	0.068
	2	**3**	0.093
	3	**3**	0.105
	4	**5**	0.122
	...	...	...
	13	**11**	0.246
	40	**17**	0.551
	121	**23**	1.376
	364	**29**	3.633
	1093	**35**	10.833
	3280	**41**	38.543
IMP	1	**1**	0.102
	2	**7**	0.185
	3	**11**	0.256
	4	**19**	0.408
	5	**34**	0.710
	6	**34**	0.739
	7	**34**	0.816
	...	...	...
	15	**61**	2.158
	31	**96**	5.107
	63	**147**	10.066
	127	**230**	21.013
	255	**377**	52.208
	511	**652**	331.138
Featherweight Java	1	**3**	1.148
	2	**7**	1.594
	3	**9**	1.650
	4	**9**	1.899
	5	**13**	2.151
	...	...	...
	21	**70**	6.905
	85	**298**	43.756
	341	**1210**	475.022
Mini Java	1	**5**	2.721
	2	**21**	3.117
	3	**40**	3.897
	4	**53**	5.750
	5	**59**	6.191
	...	...	...
	21	**275**	37.354
	85	**1133**	342.435
	341	**4565**	5981.114
Ownership Java	1	**13**	50.818
	2	**73**	77.135
	3	**110**	103.230
	4	**135**	231.328
	5	**157**	247.954
	...	...	...
	21	**733**	2760.734
	25	**877**	3963.836
	29	**1021**	5271.509
	33	**1165**	6255.260

Table 6.4: Results of checking soundness of type systems. Our system achieves significant state space reduction. For example, there are over 2^{786} well typed IMP programs of expression size up to 511, but our system checks only 652 states to exhaustively cover this space.

Max Expression Size	**Percentage of Errors Caught**
1	**0**
2	**8**
3	**40**
4	**68**
5	**76**
6	**80**
7	**84**
8	**100**

Table 6.5: Evaluating the small scope hypothesis. A maximum expression size of 8 is sufficient to catch all the type system errors that we introduced into Ownership Java.

the maximum expression size to be bound by a balanced AST with n nodes. In the imperative language IMP, we limited program states to have at most n variables and n integer literals. In Featherweight Java, Mini Java, and Ownership Java, we limited program states to have at most four classes, where each class can have at most two fields and two methods (in addition to inherited fields and methods). In Mini Java and Ownership Java, we limited program states to have at most four heap objects and n integer literals. In Ownership Java, we limited classes to have at most two owner parameters.

We report both the number of states explicitly checked by our checker and 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. The results indicate that our approach is feasible and that our model checker achieves significant state space reduction. For example, the number of well typed IMP programs of maximum size 511 is over 2^{786}, but our checker explicitly checks only 652 states to exhaustively cover this space.

Finally, Table 6.5 presents our results that suggest that exhaustive testing within a small finite domain does indeed catch all type system errors in practice, a conjecture also known as the *small scope hypothesis* [38, 47, 59]. We introduced twenty different errors into the type system of Ownership Java (one at a time) and five different errors into the operational semantics. Some are simple mistakes such as forgetting to include a type checking clause. Some are more subtle errors as the following examples illustrate.

The Java compiler rejects as ill typed a term containing a type cast of a value of declared type `T1` into a type `T2` if `T1` is neither a subtype nor supertype of `T2`. The Ownership Java (as also the Featherweight Java) compiler, however, accepts such a term as well typed. We changed Ownership Java to reject such casts as ill typed. Our model checker then correctly detected that the preservation theorem does not hold

for the changed language. The term `(T2) (Object<world>) new T1()` provides a counter example. It is well typed initially. But after the upcast, the term in effect simplifies to `(T2) new T1()` which is ill typed in the changed language. The preservation theorem therefore does not hold.

We also introduced a subtle bug (see [6, Figure 24]) into Ownership Java such that the *owners as dominators* property does not hold. Our checker correctly detected the bug.

The results in Table 6.5 indicate that exhaustive testing within a small finite domain is an effective approach for checking soundness of type systems.

CHAPTER VII

Related Work

Model checking is a formal verification technique that exhaustively tests a circuit/program on all possible inputs (sometimes up to a given size) and on all possible nondeterministic schedules. There has been much research on model checking of software. Verisoft [26] is a stateless model checker for C programs. Java Pathfinder (JPF) [67, 43] is a stateful model checker for Java programs. XRT [30] checks Microsoft CIL programs. Bandera [14] and JCAT [18] translate Java programs into the input language of model checkers like SPIN [33] and SMV [48]. Bogor [23] is an extensible framework for building software model checkers. CMC [51] is a stateful model checker for C programs that has been used to test large software including the Linux implementation of TCP/IP and the ext3 file system. Chess [52] and CalFuzzer [39] help find and reproduce concurrency bugs.

For hardware, model checkers have been successfully used to verify fairly complex finite state control circuits with up to a few hundred bits of state information; but not circuits that have large data paths or memories. Similarly, for software, model checkers have been primarily used to verify control-oriented programs with respect to temporal properties; but not much work has been done to verify data-oriented programs with respect to complex data-dependent properties.

Thus, most of the research on reducing the state space of a software model checker has focused on checking temporal properties of programs. Tools such as Slam [3], Blast [32], and Magic [9] use heuristics to construct and check an abstraction of a program (usually predicate abstraction [29]). Abstractions that are too coarse generate false positives, which are then used to refine the abstraction and redo the checking. This technique is known as Counter Example Guided Abstraction Refinement or *CEGAR*. There are also many static [26] and dynamic [25] partial order reduction systems for concurrent programs. There are many other symmetry-based reduction techniques

as well (e.g., [36]). However, none of the above techniques seem to be effective in reducing the state space of a model checker when checking complex data-dependent properties of programs. Our experiments comparing the performance of our system to other model checkers support this observation.

Tools such as Alloy [37, 41] and Korat [5] systematically generate all test inputs that satisfy a given precondition. A version of JPF [43] uses lazy initialization of fields to essentially simulate the Korat algorithm. Kiasan [19] uses a lazier initialization. However, these tools generate and test every valid state within the given finite bounds (or portion of state that is used, in case of Kiasan) and so do not achieve as much state space reduction as glass box checking. In particular, unlike the above systems, our static analysis allows us to prune a very large number of states in a single step using a SAT solver.

Tools such as CUTE [63, 27], Whispec [65], and a version of JPF [68] use constraint solvers to obtain good branch coverage (or good path coverage on paths up to a given length) for testing data structures. However, this approach could miss bugs even on small sized data structures. For example, a buggy tree insertion method that does not rebalance the tree might work correctly on a test case that exercises a certain program path, but fail on a different *unchecked* test case that exercises the same program path because the second test case makes the tree unbalanced. Therefore, it seems to us that this approach is more suitable for checking control-dependent properties rather than data-dependent properties.

Jalloy [66], Miniatur [21], and Forge [20] translate a Java program and its specifications into a boolean formula and check it with a SAT solver. In our experience with a similar approach, translating general Java code usually leads to large formulas that take a lot of time to solve with a SAT solver. Our technique of translating declarative methods and using symbolic execution for general Java code is more efficient.

Because of input nondeterminism, it is difficult to even formulate the problem of checking type soundness automatically in the context of most software model checkers. A technique exists for checking properties of programming languages specified in αProlog, using a bounded backtracking search in an αProlog interpreter [10]. However, that work does not use our search space reduction techniques and does not scale as well as our model checker.

This dissertation builds on previous work on glass box software model checking. We present the culmination of this work to date. Darga and Boyapati introduced the idea that some data-oriented programs can be efficiently checked by pruning transitions that touch the same part of the program state [16]. We then extended this

idea by using a precise symbolic execution to identify similar transitions and a static analysis to guarantee the correctness of the pruning. We also created a declarative method syntax and a process for translating declarative methods into formulas, and we used a boolean satisfiability solver to efficiently work with these formulas. Our system for glass box software model checking is presented in our work on checking type soundness [62] as well as our work on modularity [61]. This dissertation completely describes our technique, including numerous extensions, enhancements, and optimizations.

CHAPTER VIII

Conclusions

This dissertation presents *glass box model checking*, a technique for efficiently checking data-dependent properties of programs. A glass box software model checker does not check every state separately but instead checks a large set of states together in each step. A dynamic analysis discovers a set of similar states, and a static analysis checks all of them efficiently in a single step using a SAT solver. Our analysis achieves a high degree of state space reduction in this way. We present a formal description of our symbolic execution and declarative method translation.

We extended glass box model checking to enable modular checking. Our system first checks a program module against an abstract implementation, establishing functional equivalence. It then replaces the program module with the abstract implementation when checking other program modules. Adding modularity to glass box checking presents unique challenges due to the nature of glass box pruning. We overcome these challenges by using an abstraction function and a notion of equality between abstractions.

We also extended our checker to automatically check the soundness of a type system, given only the specification of type correctness of intermediate program states and the small step operational semantics. Currently, proofs of type soundness are either done on paper or are machine checked, but require significant manual assistance in both cases. Consequently proofs of type soundness are usually done *after* language design, if at all. Our system can be used *during* language design with little extra cost.

Our experimental results indicate that our glass box checking technique is effective at checking data-dependent properties. We compare our system to several state-of-the-art software model checkers and find that glass box checking is significantly more efficient in this problem domain. We find that our modular approach is effective at checking programs composed of multiple modules. Our modular approach significantly outperforms the non-modular technique for these programs. Finally, we tested our

system for checking type soundness on several small to medium sized languages that include several features such as term and type level substitution, explicit heap, and objects. Our results indicate that our approach is feasible. We expect this system to be particularly useful to researchers who design novel type systems but formalize only a core subset of their type systems, as is the standard practice in the research community.

8.1 Future Work

We present the following directions for future work.

Checking Arbitrary Code Blocks

The glass box software model checking approach excels at checking operations that depend on only a small part of the program state. Sometimes these operations appear as code blocks within a larger context, rather than as methods of a class. Using the `assert` and `assume` constructs described in Section 2.11, programmers can specify the behavior of these code blocks. Then, instead of checking a class method our system could check the code block in isolation of the rest of the method. One challenge to this approach is setting appropriate bounds on the search space, including bounds on all local variables that are live in the code block. Using a combination of intelligent defaults and programmer input, these bounds could be set appropriately for efficient and effective model checking.

Checking Atomic Method Commutativity

The property of atomicity guarantees that every concurrent interleaving is equivalent to a sequential execution. However, two atomic methods might not *commute*, which means that the order in which they are sequentially invoked affects the outcome of the program. This can lead to errors due to unexpected interleavings of atomic methods. If we check that two atomic methods commute then we can ensure that no such errors exist.

Consider checking that methods `m1` and `m2` are atomic and commute. We assume that the atomicity of the methods has already been established by one of the existing techniques. Now consider using glass box software model checking to show that these methods commute. We can exhaustively check two successive of calls to `m1` and `m2` and compare the result to the same calls but in reverse order. If we establish that

calling `m1` followed by `m2` is equivalent to calling `m2` followed by `m1` then the methods commute up to the bounds of our check.

The above analysis requires a notion of two program states being equivalent. For example, the order in which items are entered into a red-black tree affects the shape of the tree, but does not affect the resulting set of items, so these trees are all equivalent with respect to the sets they represent. Thus we would need a method analogous to the `equalTo` method for comparing abstractions in Chapter III. We would also need to check that this notion of equivalence is maintained across operations. For every two equivalent states, we would check that the states are still equivalent after the same operation is run on them.

Using First-Order Logic

We currently convert all of our constraints into boolean formulas and use a boolean satisfiability solver to check them. In this system, modeling arithmetic operators on floating point numbers is very complex. This affects our ability to effectively prune in the presence of these operators as well as our ability to handle specifications that use them. As future work, we could represent such operators in first-order logic and use a Satisfiability Modulo Theories (SMT) solver in place of a boolean satisfiability solver. This would come at an increased runtime cost, but it may be essential to properly check programs that make extensive use of floating point computations.

Parallelizing the Glass Box Algorithm

Our experiments show the total time for running our algorithm sequentially. However, the glass box algorithm can be parallelized. The critical path through the algorithm includes exhaustively choosing an unchecked state and pruning a set of similar states. Thus the dynamic analysis, which generates the set of similar states, is on the critical path as well. The static analysis is not on the critical path. As long as all static analyses are eventually run, they can be deferred or done in parallel. As future work, we could parallelize the algorithm by offloading the static analysis tasks onto other processors or cores. In our experience, that this would improve performance considerably. The static analysis comprises half of the invocations of the satisfiability solver, and these problems tend to have larger formulas and take much more time than the problem of incrementally exploring the state space.

REFERENCES

REFERENCES

[1] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.

[2] B. E. Aydemir et al. Mechanized metatheory for the masses: The POPLMARK challenge, May 2005. http://www.cis.upenn.edu/ plclub/wiki-static/poplmark.-pdf.

[3] 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.

[4] Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development*. Springer Verlag, 2004.

[5] 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.

[6] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.

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

[8] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2001.

[9] 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.

[10] J. Cheney and A. Momigliano. Mechanized metatheory model-checking. In *Principle and Practice of Declarative Programming (PPDP)*, July 2007.

[11] 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.

[12] E. M. Clarke, E. A. Emerson, and J. Sifakis. Model checking: Algorithmic verification and debugging. *Communications of the ACM (CACM) 52(11)*, 2009.

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

[14] 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.

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

[16] P. Darga and C. Boyapati. Efficient software model checking of data structure properties. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2006.

[17] R. DeLine and M. Fahndrich. Enforcing high-level protocols in low-level software. In *Programming Language Design and Implementation (PLDI)*, June 2001.

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

[19] X. Deng, J. Lee, and Robby. Bogor/Kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In *Automated Software Engineering (ASE)*, September 2006.

[20] G. Dennis, F. Chang, and D. Jackson. Modular verification of code with SAT. *International Symposium on Software Testing and Analysis*, 2006.

[21] J. Dolby, M. Vaziri, and F. Tip. Finding bugs efficiently with a SAT solver. In *European Software Engineering Conference and Foundations of Software Engineering (ESEC/FSE)*, September 2007.

[22] S. Drossopoulou and S. Eisenbach. Java is type safe—probably. In *European Conference for Object-Oriented Programming (ECOOP)*, June 1997.

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

[24] N. Een and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *Theory and Applications of Satisfiability Testing (SAT)*, June 2005.

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

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

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

[28] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley, 2005.

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

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

[31] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Programming Language Design and Implementation (PLDI)*, June 2002.

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

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

[34] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1999.

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

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

[37] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.

[38] D. Jackson and C. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Transactions on Software Engineering (TSE) 22(7)*, July 1996.

[39] P. Joshi, M. Naik, C.-S. Park, and K. Sen. An extensible active testing framework for concurrent programs. *Computer Aided Verification (CAV)*, 2009.

[40] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

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

[42] S. Khurshid, D. Marinov, and D. Jackson. An analyzable annotation language. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.

[43] 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.

[44] J. C. King. Symbolic execution and program testing. In *Communications of the ACM (CACM) 19(7)*, August 1976.

[45] 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.

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

[47] D. Marinov, A. Andoni, D. Daniliuc, S. Khurshid, and M. Rinard. An evaluation of exhaustive testing for data structures. Technical Report TR-921, MIT Laboratory for Computer Science, September 2003.

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

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

[50] 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.

[51] 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.

[52] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. *Operating System Design and Implementation (OSDI)*, 2008.

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

[54] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Principles of Programming Languages (POPL)*, January 2002.

[55] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer Verlag, 2002.

[56] T. Nipkow and D. von Oheimb. Java light is type-safe—definitely. In *Principles of Programming Languages (POPL)*, January 1998.

[57] The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and Technology, Planning Report 02-3, May 2002.

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

[59] J. Offutt and R. Untch. Mutation 2000: Uniting the orthogonal. In *Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, October 2000.

[60] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[61] M. Roberson and C. Boyapati. Efficient modular glass box software model checking. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2010.

[62] M. Roberson, M. Harries, P. T. Darga, and C. Boyapati. Efficient software model checking of soundness of type systems. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2008.

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

[64] P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strnisa. Ott: Effective tool support for the working semanticist. In *International Conference on Functional Programming (ICFP)*, October 2007.

[65] D. Shao, S. Khurshid, and D. Perry. Whispec: White-box testing of libraries using declarative specifications. *Library-Centric Software Design (LCSD)*, 2007.

[66] 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.

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

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

[69] D. Walker. A type system for expressive security policies. In *Principles of Programming Languages (POPL)*, January 2000.

[70] G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.

[71] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. In *Information and Computation 115(1)*, November 1994.