# Efficient Modular Glass Box Software Model Checking

Michael Roberson Chandrasekhar Boyapati

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

## Abstract

Glass box software model checking incorporates novel techniques to identify similarities in the state space of a model checker and safely prune large numbers of redundant states without explicitly checking them. It is significantly more efficient than other software model checking approaches for checking certain kinds of programs and program properties.

This paper presents PIPAL, a system for *modular* glass box software model checking. Extending glass box software model checking to perform modular checking is important to further improve its scalability. It is nontrivial because unlike traditional software model checkers such as Java PathFinder (JPF) and CMC, a glass box software model checker does not check every state separately—instead, it checks a large set of states together in each step. We present a solution and demonstrate PIPAL's effectiveness on a variety of programs.

***Categories and Subject Descriptors*** D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.5 [*Software Engineering*]: Testing and Debugging; D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.3.1 [*Logics and Meanings of Programs*]: Specifying, Verifying, and Reasoning about Programs

***General Terms*** Algorithms, Reliability, Verification

***Keywords*** Pipal, Software Model Checking

## 1. Introduction

Model checking is a formal verification technique that exhaustively tests a circuit/program 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 [1, 3–5, 7, 13, 16, 21, 39, 48] and on state space reduction techniques for software model checkers such as partial order reduction [15, 16] and tools based on predicate abstraction [19] such as Slam [1], Blast [21], or Magic [4], none of these techniques seem to be effective in reducing the state space of data-oriented programs. For example, predicate abstraction relies on alias analysis that is often too imprecise.

In recent previous work [8, 43], we introduced *glass box* software model checking to address this problem. Our checker incorporates novel techniques to identify similarities in the state space of a model checker and safely prune large numbers of redundant states without explicitly checking them. Thus, while traditional software model checkers such as Java PathFinder (JPF) [48] and CMC [39] separately check every reachable state within a state space, our glass box checker checks a (usually very large) set of similar states in each step. This leads to several orders of magnitude speedups [8] over previous model checking approaches.

This paper presents PIPAL, 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 [6].

Extending traditional model checking to perform modular checking is trivial. For example, Java PathFinder (JPF) [48] or CMC [39] 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 signif-



*OOPSLA/SPLASH'10,* October 17–21, 2010, Reno/Tahoe, Nevada, USA.

icant performance advantage of glass box model checking over traditional model checking. In particular, it is nontrivial to extend the previous work on glass box checking [8, 43] 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 paper presents a technique to solve this problem.

Note that like most model checking techniques [3, 13, 16, 39, 48], our system PIPAL (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 [27, 35, 42]. 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 PIPAL, formal verification techniques that use theorem provers [2, 29, 40] 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 PIPAL 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 [34] to write these functions anyway, in which case PIPAL requires no extra human effort. However, checking a similar program using a theorem prover such as Coq [2] requires more than 1000 lines of extra human effort.

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

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

We tested PIPAL on a variety of programs. Our experiments indicate that the modular model checking technique is far more efficient than checking programs as a unit. We also compared PIPAL to Blast [21], JPF [48], and Korat [3] and found that PIPAL is significantly more efficient when checking data-oriented programs and data-dependent properties.

The rest of this paper is organized as follows. Section 2 illustrates our approach with an example. Sections 3 describes our modular model checking approach. Section 4 presents a formal description. Section 5 contains experimental results. Section 6 discusses related work and Section 7 concludes.

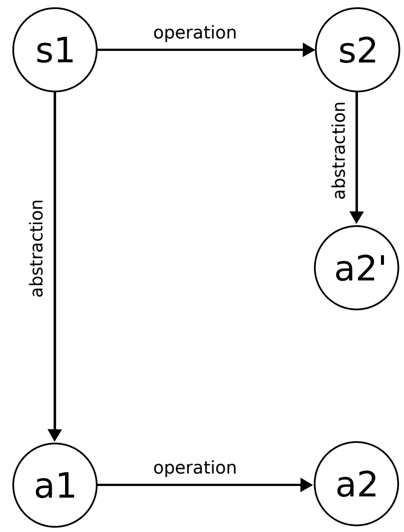


**Figure 1.** Glass box checking against an abstraction. PIPAL checks that the outputs of executing the same operation on `s1` and `a1` are the same and the states `a2` and `a2′` are equal.

## 2. Example

Consider checking the Java program in Figure 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 4. Thus the program has two modules: `IntCounter` and `SearchTree`. PIPAL's modular approach checks each of these independently.

### 2.1 Abstraction

PIPAL 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 simplicity, this example omits other `Map` operations such as `delete`.) Figure 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 PIPAL's performance. In fact, `AbstractMap` can be used in place of any data structure that implements the `Map` interface, including complex data structures such as hash tables and red-black trees.

Note that `AbstractMap` uses a construct called `PipalList`. This is simply a linked list provided by PIPAL that is useful in many abstract implementations. Using `PipalList` enables PIPAL 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.

```
class IntCounter {
  Map map = new SearchTree();
  int max_frequency   = 0;
  int most_frequent_i = 0;

  public void count(int i) {
    Integer frequency = (Integer)map.get(i);
    if (frequency == null) frequency = new Integer(0);
    map.insert(i, new Integer(frequency+1));

    if (frequency >= max_frequency) {
      max_frequency   = frequency;
      most_frequent_i = i;
    }
  }

  public int get_most_frequent_i() {
      return most_frequent_i;
  }

  public int get_max_frequency() {
      return max_frequency;
  }
}
```

**Figure 2.** `IntCounter` internally using a `SearchTree`.

### 2.2 Checking the Abstraction

PIPAL checks that `SearchTree` behaves the same as `AbstractMap`. To do this it (in effect) exhaustively checks every valid state of `SearchTree` within some given finite bounds against an equivalent state of `AbstractMap`. Figure 1 illustrates how PIPAL checks that `SearchTree` and `AbstractMap` have the same behavior. PIPAL runs the same operation on a `SearchTree` state `s1` and its abstraction `a1` to obtain states `s2` and `a2` respectively. PIPAL then checks that (1) the abstraction of `s2` is equal to `a2`, and (2) the return values are same. PIPAL invokes the abstraction function to generate the abstractions of states `s1` and `s2`. The abstraction function for `SearchTree` is in Figure 4. The method for testing equality of two `AbstractMaps` is shown in Figure 5.

Given a bound of 3 on the height of the tree, Figure 3(a) shows some possible states of `SearchTree`. PIPAL generates states of `AbstractMap` by calling an abstraction function. It creates a `PipalList` and passes it as an argument to the constructor of `AbstractMap`. PIPAL provides methods for generating `PipalLists` from several data structures, to make it convenient to implement abstraction functions. Behind the scenes, PIPAL constructs a list long enough to hold the largest possible tree within the given bounds. Figure 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 model checking algorithm described in Section 3.

Consider checking an `insert` operation on state `t1` in Figure 3(a). After the operation, the resulting state is `t1'`. PIPAL detects that 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`). PIPAL then determines that once it checks the `insert` operation on tree `t1`, it is redundant to check the same `insert` operation on trees `t2`, `t3`, and the exponentially many trees where the highlighted nodes remain the same. PIPAL safely prunes those trees from its search space, while still achieving complete coverage within the bounded domain. Thus, for this example, PIPAL only explicitly checks each operation once on each unique tree path rather than each unique tree. This leads to significant reduction in the size of the search space. PIPAL's symbolic analysis (c.f. Section 3.6) and static analysis (c.f. Section 3.7) techniques ensure that the presence of the abstract map does not increase the number of states that are explicitly checked.

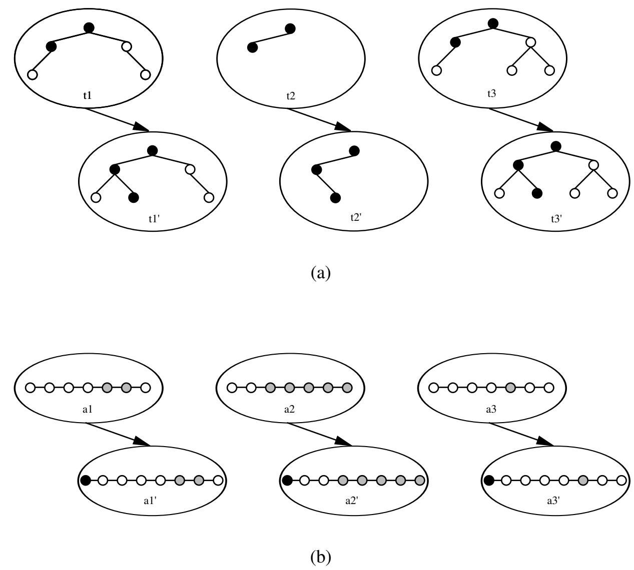


**Figure 3.** (3a) Three search trees (code in Figure 4), before an after an `insert` operation. The tree path touched by the operation is highlighted in each case. Note that the tree path is the same in all three cases. Once PIPAL checks the `insert` operation on tree `t1`, it determines that it is redundant to check the same `insert` operation on trees `t2` and `t3`. (3b) The corresponding abstract maps (code in Figure 5). The list nodes in gray correspond to tree nodes that are not reachable.

### 2.3 Checking Using the Abstraction

Once PIPAL 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 `most_frequent_i` and `max_frequency` correspond to the most frequent integer in the map and its frequency, respectively. When checking `IntCounter`, PIPAL substitutes `SearchTree` with `AbstractMap`. Otherwise, the checking proceeds as above. PIPAL 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.

```
class SearchTree implements Map {
    static class Node implements PipalList.ListNodeSource {
        int key;
        Object value;
        @Tree Node left;
        @Tree Node right;

        Node(int key, Object value) {
            this.key = key;
            this.value = value;
        }

        AbstractMap.Node abstraction() {
            return new AbstractMap.Node(key, value);
        }
    }

    @Tree Node root;

    Object get(int key) {
        Node n = root;
        while (n != null) {
            if (n.key == key)
                return n.value;
            else if (key < n.key)
                n = n.left;
            else
                n = n.right;
        }
        return null;
    }

    void insert(int key, Object value) {
        Node n = root;
        Node parent = null;
        while (n != null) {
            if (n.key == key) {
                n.value = value;
                return;
            } else if (key < n.key) {
                parent = n;
                n = n.left;
            } else {
                parent = n;
                n = n.right;
            }
        }

        n = new Node(key, value);
        if (parent == null)
            root = n;
        else if (key < parent.key)
            parent.left = n;
        else
            parent.right = n;
    }

    @Declarative
    boolean repOk() {
        return isOrdered(root, null, null);
    }

    @Declarative
    static boolean isOrdered(Node n, Node low, Node high) {
        if (n == null) return true;
        if (low  != null && low.key  >= n.key) return false;
        if (high != null && high.key <= n.key) return false;
        if !(isOrdered(n.left,  low, n   ))    return false;
        if !(isOrdered(n.right, n,   high))    return false;
        return true;
    }

    AbstractMap abstraction() {
        return new AbstractMap(Pipal.ListFromTree_BF(root));

        // ListFromTree_BF returns a PipalList corresponding
        // to a breadth first traversal of the tree.
    }
}
```

**Figure 4.** A simple search tree implementation.

```
class AbstractMap implements Map {
    static class Node {
        Object key;
        Object value;

        Node(Object key, Object value) {
            this.key = key;
            this.value = value;
        }

        @Declarative
        boolean equalTo(Node n) {
            return n.key.equals(key) && n.value == value;
        }
    }

    PipalList list;

    AbstractMap(PipalList l) {
        list = l;
    }

    Object get(Object key) {
        PipalList.Node pnode = list.head();

        while (pnode != null) {
            Node n = (Node)pnode.data();
            if (n.key.equals(key)) {
                return n.value;
            } else {
                pnode = pnode.next();
            }
        }
    }

    void insert(Object key, Object value) {
        PipalList.Node pnode = list.head();

        while (pnode != null) {
            Node n = (Node)pnode.data();
            if (n.key.equals(key)) {
                n.value = value;
                return;
            } else {
                pnode = pnode.next;
            }
        }

        list.add(new Node(key, value));
    }

    @Declarative
    public boolean equalTo(AbstractMap m) {
        return list.equalTo(m.list);
    }
}
```

**Figure 5.** An abstract map implementation.

Using `AbstractMap` instead of `SearchTree` has several advantages. First, our state space reduction techniques are more effective on `AbstractMap`. In Figure 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 (c.f. Section 3.9).

## 3. Approach

This section presents PIPAL's modular checking approach. Section 4 contains a formal description of our approach.

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

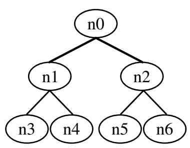


**Figure 6.** Search space for the binary tree in Figure 4 with tree height at most 3 and at most 10 keys and 4 values.

### 3.1 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 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 4), using a balanced binary tree such as a red-black tree, using a hash table, or using 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 4 against the abstract map in Figure 5, programmers only need to specify the representation invariant of the search tree (repOk in Figure 4), the abstraction function (abstraction in Figure 4), and finite bounds on the size of the search trees (similarly to [8]). PIPAL 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 3.10.

| Field | Value |
|---|---|
| root | n0 |
| n0.left | n1 |
| n0.right | n2 |
| n1.left | n3 |
| n1.right | null |
| n2.left | null |
| n2.right | n6 |
| n0.key | 5 |
| n1.key | 2 |
| n2.key | 7 |
| n3.key | 0 |
| n4.key | |
| n5.key | |
| n6.key | 9 |
| n0.value | a |
| n1.value | d |
| n2.value | a |
| n3.value | b |
| n4.value | |
| n5.value | |
| n6.value | d |
| *method* | insert |
| get.key | |
| insert.key | 3 |
| insert.value | a |

| Field | Value |
|---|---|
| root | n0 |
| n0.left | n1 |
| n0.right | null |
| n1.left | n3 |
| n1.right | null |
| n2.left | |
| n2.right | |
| n0.key | 6 |
| n1.key | 7 |
| n2.key | |
| n3.key | 4 |
| n4.key | |
| n5.key | |
| n6.key | |
| n0.value | b |
| n1.value | d |
| n2.value | |
| n3.value | c |
| n4.value | |
| n5.value | |
| n6.value | |
| *method* | insert |
| get.key | |
| insert.key | 3 |
| insert.value | a |

**Figure 7.** Two elements of the search space in Figure 6. The first element represents insert(3,a) on an ordered tree. The second element represents the same operation, but on an unordered tree. Values of unreachable fields are left blank.

The current implementation of PIPAL checks Java programs. However, our underlying checking technique is general and can be used to check programs in other languages as well.

Note that in Figures 4 and 5, the methods repOk and equalTo are annotated as Declarative. Declarative methods are described in Section 3.9. Declarative methods use only a subset of Java and do not contain side effects. PIPAL translates declarative methods into boolean formulas for efficient model checking. PIPAL requires that the repOk and equalTo methods always be declarative. Finally, the Tree annotations in Figure 4 denote that the Nodes form a tree, similarly to [8]. Such annotations reduce PIPAL's search space because it does not have to check non-tree structures.

### 3.2 Search Space

Traditional software model checkers [1, 4, 7, 16, 21, 39, 48] explore a state space by starting from the initial state and systematically generating and checking every successor state. But this approach does not work (c.f. [8, 43]) for software model checkers that use the glass box technique. Instead, PIPAL organizes its search space as follows.

Consider checking the binary search tree implementation in Figure 4 against the abstract map in Figure 5. Suppose PIPAL 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 6. 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 an operation on a binary tree. Figure 7 shows two elements of this search space.

```
1 void search( BoundedSearchSpace B ) {
2     S = Set of all valid elements in B
3     while ( S ≠ ∅ ) {
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 ) {
9             Print bug trace s′
10        }
11        S  = S - S′
12    }
13 }
```

**Figure 8.** Pseudo-code for the glass box search algorithm.

| Line | Symbolic Value of Branch Condition | Concrete Value of Branch Condition |
|---|---|---|
| 36 | root≠null | true |
| 37 | n0.key=key | false |
| 40 | key<n0.key | true |
| 36 | n0.left≠null | true |
| 37 | n1.key=key | false |
| 40 | key<n1.key | false |
| 36 | n1.right≠null | false |
| 50 | false | false |
| 52 | key<n1.key | false |

**Figure 9.** Symbolic and concrete values of the branch conditions encountered during the execution of the `insert(3,a)` operation on the first element in Figure 7. The symbolic values are used to generate the path constraint.

The first element represents the operation `insert(3,a)` on an ordered tree. The second element 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 representation invariant.

### 3.3 Search Algorithm

Figure 8 presents the pseudo-code for the glass box search algorithm. Given a bounded search space `B` PIPAL first initializes the search space `S` to all *valid* states in `B`. For example, given the bounded search space `B` in Figure 6, PIPAL first initializes the search space `S` to all states in `B` on which `repOk` returns true. In Lines 3-12, PIPAL iterates until the search space `S` is exhausted. In each iteration, it selects an unchecked state `s` from `S` and checks the desired property on it. For example, when checking the binary search tree in Figure 4 against the abstract map in Figure 5, Pipal checks that: executing the operation on `s` preserves its representation invariant, the outputs of executing the operation on s and its abstraction are the same, and the abstraction of the final state is equal to the final state of executing the operation on the abstraction (c.f. Figure 1). In Line 6 it discovers a set `S′` of states similar to `s` using its dynamic analysis described in Section 3.6. In Line 7 it checks the entire set of states `S′` using its static analysis described in Section 3.7. If any of the states fails the check, PIPAL prints an explicit bug trace. Finally, PIPAL removes all the checked states `S′` from `S`. The following sections describe the above steps in detail.

### 3.4 Search Space Representation

In the above algorithm, PIPAL performs several operations on the search space, including choosing an unchecked element (Line 4 in Figure 8), constructing a subset (Line 6), checking the subset (Line 7), and pruning the subset from the search space (Line 11). Consider checking the binary search tree in Figure 4 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 propositional logic formula (boolean formula). We use an incremental SAT solver MiniSat [14] to perform the various search space operations.

For example, consider the search space in Figure 6. PIPAL encodes the value of each field using $\lceil \log n \rceil$ boolean variables, where $n$ is the size of the domain of the field. So PIPAL encodes `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`. PIPAL invokes the SAT solver to provide a satisfying assignment to the variables of the formula and then decodes it into a concrete state. Line 3 in Figure 8, checking if a set is empty, and Line 4, choosing an element of a non-empty set, may be expensive operations because they invoke the SAT solver. Line 11 in Figure 8, 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.

### 3.5 Search Space Initialization

In Line 2 of Figure 8, given a bounded search space `B`, PIPAL first initializes the search space `S` to the set of all valid states in `B`. For example, given the bounded search space `B` in Figure 6, PIPAL first initializes 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`. PIPAL accomplishes this by translating the `repOk` method and the 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 4 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 3.9 describes how PIPAL translates such declarative methods into formulas.

| 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` | b is effect free | |
| `if c return x else return y` | c,x,y are effect free | |
| `x=e` | e is effect free | |
| `x.m(..)` | m is declarative | x≠null |
| `x.f` | | x≠null |
| `x[i]` | | x≠null ∧ i in bounds |

**Figure 10.** Java constructs that PIPAL executes symbolically without generating path constraints (except for exception condition). The restrictions indicate the conditions under which PIPAL executes these constructs 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.

### 3.6 Dynamic Analysis

Given an element of the search space, the purpose of the dynamic analysis (Line 6 in Figure 8) 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 3.7.

Consider checking the binary search tree implementation in Figure 4 against the abstract map in Figure 5. Suppose we are given the first element in Figure 7. PIPAL first constructs the state, say `s1`, corresponding to the given element. `Pipal` then runs the corresponding `insert(3,a)` operation on the state `s1` to obtain the state `s2`. As shown in Figure 1, PIPAL also runs the abstraction function (the method `abstraction` in Figure 4) on the states `s1` and `s2` to obtain the states `a1` and `a2`′ respectively, runs the same `insert(3,a)` operation on the state `a1` to obtain the state `a2`, and checks if `a2` and `a2`′ are equal (using the method `equalTo` in Figure 5).

As Pipal concretely executes methods (`insert`, `abstraction`, and `equalTo`) in the above example, it also symbolically executes them [33] 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 36 (in Figure 4), 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` ≠ `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 37, testing `n.key == key`. This symbolically evaluates to `n0.key` = `key`, concretely to `false`. Execution continues in this way. Figure 9 summarizes all branch conditions encountered during execution of the `insert` method.

| Field | Symbolic Value |
|---|---|
| `list` | `l0` |
| `l0.reachable` | `root`≠`null` |
| `l1.reachable` | `root`≠`null` ∧ `n0.left` ≠`null` |
| `l2.reachable` | `root`≠`null` ∧ `n0.right`≠`null` |
| `l3.reachable` | `root`≠`null` ∧ `n0.left` ≠`null` ∧ `n1.left` ≠`null` |
| `l4.reachable` | `root`≠`null` ∧ `n0.left` ≠`null` ∧ `n1.right`≠`null` |
| `l5.reachable` | `root`≠`null` ∧ `n0.right`≠`null` ∧ `n2.left` ≠`null` |
| `l6.reachable` | `root`≠`null` ∧ `n0.right`≠`null` ∧ `n2.right`≠`null` |
| `l0.key` | `n0.key` |
| `l1.key` | `n1.key` |
| `l2.key` | `n2.key` |
| `l3.key` | `n3.key` |
| `l4.key` | `n4.key` |
| `l5.key` | `n5.key` |
| `l6.key` | `n6.key` |
| `l0.value` | `n0.value` |
| `l1.value` | `n1.value` |
| `l2.value` | `n2.value` |
| `l3.value` | `n3.value` |
| `l4.value` | `n4.value` |
| `l5.value` | `n5.value` |
| `l6.value` | `n6.value` |
| *method* | `(SearchTree.)`*method* |
| `get.key` | `(SearchTree.)get.key` |
| `insert.key` | `(SearchTree.)insert.key` |
| `insert.value` | `(SearchTree.)insert.value` |

**Figure 11.** Symbolic state of the abstract map in Figure 5 generated by symbolically executing the abstraction function (`abstraction` in Figure 4) on the first element in Figure 7.

PIPAL generates the path constraint by taking the conjunction of the symbolic branch conditions. (The false branch conditions are first negated.) All states satisfying the path constraint are considered *similar* to each other (Line 6 in Figure 8). 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 include values used in instructions that PIPAL 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 10 summarizes Java constructs that PIPAL executes symbolically without generating path constraints (except for exception condition).

Note from Figure 10 that Pipal executes calls to declarative methods symbolically. Declarative methods do not contain side effects. Given a declarative method and the current symbolic program state, PIPAL generates a symbolic return value of the declarative method by translating the declarative method (and the methods it transitively invokes) into a formula (c.f. Section 3.9). The advantage of using declarative methods for writing specifications is that branches in declarative methods then do not generate any path constraints. This allows PIPAL to identify a larger set of similar states.

| 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`} |

**Figure 12.** 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`.

| Field | Value |
|---|---|
| `o1.value` | `null` |
| `o2.value` | `null` |
| `o2.value` | `null` |
| *method* | `foo` |
| `foo.p1` | `o1` |
| `foo.p2` | `o2` |
| `foo.p3` | `o3` |

| Field | Value |
|---|---|
| `o1.value` | `null` |
| `o2.value` | `null` |
| `o3.value` | `null` |
| *method* | `foo` |
| `foo.p1` | `o2` |
| `foo.p2` | `o1` |
| `foo.p3` | `o3` |

**Figure 13.** Two isomorphic elements of the search space in Figure 12. The two elements are isomorphic because `o1` and `o2` are equivalent memory locations.

### 3.7 Static Analysis

The dynamic analysis above identifies a set $\mathtt{S}'$ of similar states (Line 6 in Figure 8) 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 PIPAL 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 the states in $\mathtt{S}'$ (Line 7 in Figure 8).

Consider checking the binary search tree implementation in Figure 4 against the abstract map in Figure 5. To check that the code works correctly on all the states $\mathtt{S}'$ that follow the same program path, Pipal constructs the formula: $\mathtt{S}' \rightarrow \mathtt{R} \wedge \mathtt{O} \wedge \mathtt{A}$. Recall Figure 1 for notation. `R` asserts that for every state `s1` in $\mathtt{S}'$, `s2.repOk()`. `O` asserts that the outputs of executing the operation on `s1` and `a1` are the same. `A` asserts that `a2.equalTo(a2′)`. PIPAL uses a SAT solver to find a counterexample to this formula, or establish that none exists. If it finds a counterexample, PIPAL decodes it into a concrete state and presents it as an explicit bug trace.

To generate the formulas `A` and `R`, PIPAL symbolically executes the operations and abstraction functions as described in Section 3.6. After symbolic execution, every field and variable contains a symbolic value that is represented by a formula. For example, Figure 11 shows the symbolic state of the abstract map in Figure 5 generated by symbolically executing the abstraction function (the method `abstraction` in Figure 4) on the first element in Figure 7. Section 4 formally describes our symbolic execution technique.

Note that the above abstraction function calls the method `ListFromTree_BF`, which creates a `PipalList` sufficiently long to hold the largest possible tree within the given bounds. It does so without generating path constraints to make model checking efficient. It adds a `reachable` bit to each node of `PipalList` that is set if the corresponding tree node is reachable. PIPAL provides similar methods for conveniently generating `PipalLists` from several data structures.

### 3.8 Isomorphism Analysis

Consider checking a method `foo` with three formal parameters `p1`, `p2`, and `p3`. Figure 12 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 13. These two elements are isomorphic because `o1` and `o2` are equivalent memory locations. Therefore, once PIPAL checks the first element, it is redundant to check the second element. PIPAL avoids checking isomorphic elements as follows. Consider the first element of the search space in Figure 13. Suppose the method `foo` reads only `p1` and `p2` and the analyses in the previous sections conclude that all states where ($\mathtt{p1{=}o1} \wedge \mathtt{p2{=}o2}$) can be pruned. The isomorphism analysis then determines that all states that satisfy the following formula can also be safely pruned: ($\mathtt{p1} \in \{\mathtt{o2,o3}\} \vee (\mathtt{p1{=}o1} \wedge \mathtt{p2} \in \{\mathtt{o3}\})$).

In general, given a program state `s`, PIPAL constructs such a formula $\mathtt{I_s}$ denoting the set of states isomorphic to `s` as follows. Recall from Section 3.6 that the symbolic execution on `s` builds a path constraint formula, say $\mathtt{P_s}$. Suppose during symbolic execution PIPAL encounters 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. PIPAL then prunes 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 [24, 37]. 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 PIPAL checks a state `s`, it computes a compact formula $\mathtt{I_s}$ denoting a (often exponentially large) set of states isomorphic to s, and prunes $\mathtt{I_s}$ from the search space. PIPAL *never visits* the (often exponentially many) states in the set $\mathtt{I_s}$.

### 3.9 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 in PIPAL have the `@Declarative` annotation. A declarative method may not contain object creations, assignments, loops, or exception handlers, and may only call declarative methods. 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 [32]. 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 [26] and then translates Alloy into propositional logic. Section 4 formally describes our translation process.

Non-declarative methods may call declarative methods. If PIPAL encounters a declarative method during symbolic execution, it symbolically executes 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 8. 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.

The above algorithm translates the same declarative method several times. For example, consider checking the code in Figure 4. During search space initialization (c.f. Section 3.5), the above algorithm translates the invocation of the `isOrdered` method on each tree node. Subsequently, during each iteration of the loop in Figure 8, 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, PIPAL caches 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. PIPAL also maintains a list of fields that each cache entry depends on. If any of these fields changes during an operation, that 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 8). If the cache entry is disabled or does not exist, the method must be translated every time. To avoid that, PIPAL uses a *temporary* cache. When a declarative method is called, PIPAL looks up the method and its parameters in the main cache. If the cache misses (because the entry is disabled or was never created) then PIPAL tries the temporary cache. If that misses then PIPAL translates the method and stores 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 PIPAL considerably.

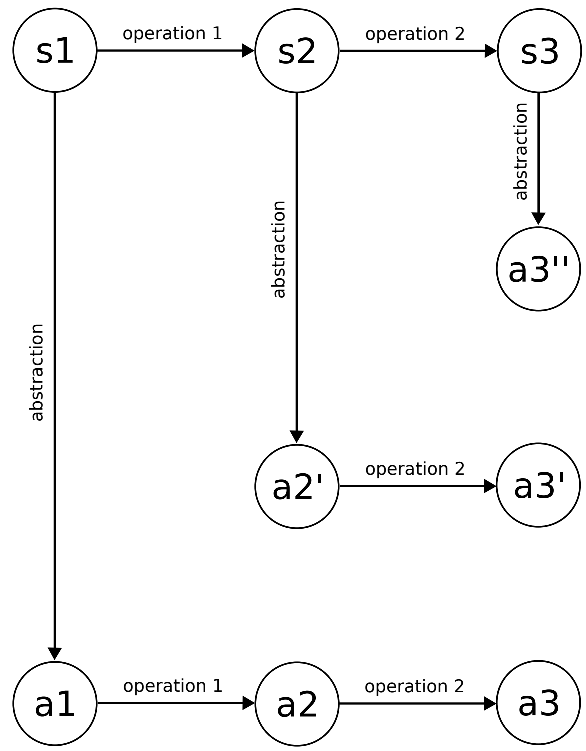


**Figure 14.** Operations on a module and its abstraction.

### 3.10 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, PIPAL checks the following two properties in those finite bounds.

The first property that PIPAL checks is as follows. See Figure 14 for notation. PIPAL checks 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 PIPAL checks is as follows. See Figure 14 for notation. PIPAL checks 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,

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

**Figure 15.** Syntax of a simple Java-like language. We write $\overline{cd}$ as shorthand for $cd_1\ cd_2\ ...\ cd_n$ (with the commas), write $\overline{vd}$ as shorthand for $vd_1$, $vd_2$, ..., $vd_n$ (without the commas), etc., similar to the notation in [23].

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 14 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.

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

**Figure 16.** Syntax of a declarative subset of the language in Figure 15, showing the syntax of declarative methods.

Pipal checks the above two properties efficiently using the search algorithm described in the above sections. To check the first property using the search algorithm in Figure 8, PIPAL 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 PIPAL 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 PIPAL requires `repOk` and `equalTo` to be declarative methods, so that they can be efficiently translated into boolean formulas during search space initialization (c.f. Section 3.5).

# 4. Formal Description

This section formalizes parts of PIPAL's dynamic and static analyses for the simple Java-like language in Figure 15. This language resembles Featherweight Java [23] but it also includes imperative constructs such as assignments to a mutable heap. This section assumes 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.

## 4.1 Symbolic Values and Symbolic State

Consider the simple Java-like language of Figure 15. The concrete values of this language include the expressions `true`, `false`, `null`, and any object allocated on the heap. We define a *symbolic value* as a set of elements of the form $c \to \alpha$, where $c$ is a boolean formula and $\alpha$ is a concrete value. Informally, the formula $c$ is a condition that must hold for the corresponding concrete value to be $\alpha$. For a given symbolic value, we require that all conditions $c$ are mutually exclusive, and that all concrete values $\alpha$ are distinct. As a notational convenience, we use $\{\alpha\}$ as an abbreviation for the singleton set $\{\textit{true} \to \alpha\}$, which is a symbolic value

RC-SEQ

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

RC-FIELD-READ

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

RC-FIELD-WRITE

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

RC-FIELD-WRITE-2

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

RC-IF

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

RC-CALL

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

RC-CALL-2

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

RC-OP

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

**Figure 17.** Congruence reduction rules of symbolic execution for the simple Java-like language in Figure 15.

that represents a single concrete value $\alpha$. Furthermore, we use $\{c \rightarrow \texttt{true}\}$ as an abbreviation for the set $\{c \rightarrow \texttt{true}, \neg c \rightarrow \texttt{false}\}$, which is a boolean symbolic value whose boolean concrete value corresponds to the formula $c$. For a symbolic value $v$ and a concrete value $\alpha$, let $v = \alpha$ denote $c$ if $(c \rightarrow \alpha) \in v$ and *false* otherwise.

Consider an assignment $\Phi$ of truth values to boolean variables. During symbolic execution, such an assignment $\Phi$ is generated by a SAT solver (in Line 4 of Figure 8). Then, let $\Phi(c)$ denote the truth value of the boolean formula $c$ under $\Phi$. For a symbolic value $v$, let $\Phi(v)$ denote $\alpha$ if $\Phi(v = \alpha)$.

We define a *symbolic state* as a triple $\langle H, P, e\rangle$, where $H$ is the heap, $P$ is the current path constraint, and $e$ is the current expression to evaluate. The heap $H$ maps objects and fields to symbolic values. Let $H(f, \alpha)$ denote the symbolic value of the field $f$ of the object $\alpha$ in $H$. Let $H[(f, \alpha) \leftarrow v]$ denote a heap identical to $H$ except that it maps the field $f$ of the object $\alpha$ to $v$. To facilitate dynamic semantics, we extend the expression syntax of Figure 15 to include symbolic values:

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

Initially, the symbolic state is $\langle H_0, \textit{true}, \{\alpha_0\}.m(\overline{v})\rangle$, where $H_0$ is the initial heap of the finite search space, $\alpha_0$ 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 $\overline{\beta}$ (c.f. Figure 6). The formulas $\overline{c}$ are each in terms of $\lceil \log n \rceil$ fresh boolean variables that mutually define a binary index into $\overline{\beta}$.

### 4.2 Symbolic Execution

Figures 17 and 18 define the small-step operational semantics of symbolic execution. The reductions in Figure 17 are congruence rules for evaluating subexpressions. The reductions in Figure 18 define the rules for symbolic execution.

The rules describe how expressions evaluate to symbolic values, and how expressions change the heap and the path constraint. For example, the rule R-FIELD-READ evaluates expressions of the form $v.f$, where $v = \{\overline{c} \rightarrow \overline{\alpha}\}$ is a symbolic value and $f$ is a field. For each non-null $\alpha_i \in \overline{\alpha}$, the symbolic value $H(f, \alpha_i)$ is the result of accessing field $f$ through the object $\alpha_i$. 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. The evaluation reaches an error because of null pointer dereferencing. Otherwise, if $\Phi(v)$ is not `null`, then this fact is added to the path constraint.

R-SEQ

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

R-FIELD-READ

$$\frac{\begin{array}{c} v = \{\overline{c} \to \overline{\alpha}\} \\ v' = \{d \to \beta \mid d = \bigvee\{c_i \wedge d_j \mid (d_j \to \beta) \in H(f, \alpha_i)\}\} \\ \Phi(v) \neq \texttt{null} \end{array}}{\langle H, P, v\,\texttt{.}\,f\rangle \longrightarrow \langle H, P \wedge \neg(v = \texttt{null}), v'\rangle}$$

R-FIELD-WRITE

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

R-IF-T

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

R-IF-F

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

R-WHILE

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

R-NEW

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

R-CALL

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

R-EQUALS

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

R-AND-T

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

R-AND-F

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

R-OR-T

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

R-OR-F

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

R-NOT

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

**Figure 18.** Small-step operational semantics of symbolic execution. The symbolic state includes a heap $H$, a path constraint $P$, and an expression $e$. An assignment $\Phi$ is assumed to be available for converting symbolic values $v$ into concrete values $\alpha$.

RD-FIELD-READ

$$\frac{H \vdash de \Downarrow (v, E) \quad v = \{\overline{c} \rightarrow \overline{\alpha}\} \quad v' = \{d \rightarrow \beta \mid d = \bigvee\{c_i \wedge d_j \mid (d_j \rightarrow \beta) \in H(f, \alpha_i)\}\}}{H \vdash de.f \Downarrow (v', E \vee v = \texttt{null})}$$

RD-IF

$$\frac{\begin{array}{c} H \vdash de_0 \Downarrow (\{b \rightarrow \texttt{true}\}, E_0) \quad H \vdash de_1 \Downarrow (\{\overline{c} \rightarrow \overline{\alpha}\}, E_1) \quad H \vdash de_2 \Downarrow (\{\overline{d} \rightarrow \overline{\beta}\}, E_2) \\ v = \{g \rightarrow \gamma \mid g = \bigvee\{b \wedge c_i \mid \alpha_i = \gamma\} \vee \bigvee\{\neg b \wedge d_i \mid \beta_i = \gamma\}\} \\ E = E_0 \vee (b \wedge E_1) \vee (\neg b \wedge E_2) \end{array}}{H \vdash \texttt{if } de_0 \texttt{ then } de_1 \texttt{ else } de_2 \Downarrow (v, E)}$$

RD-CALL

$$\frac{\begin{array}{c} H \vdash de \Downarrow (\{c \rightarrow \alpha\}, E) \quad mbody(m, \alpha) = \overline{x}.de_0 \quad \alpha \neq \texttt{null} \\ \forall i : H \vdash de_i \Downarrow (v_i, E_i) \\ H \vdash de_0[\overline{v}/\overline{x}, \{\alpha\}/\texttt{this}] \Downarrow (v', E') \end{array}}{H \vdash de.m(\overline{de}) \Downarrow (v', E \vee \bigvee \overline{E} \vee E')}$$

RD-CALL-NULL

$$\frac{H \vdash de \Downarrow (\{c \rightarrow \texttt{null}\}, E)}{H \vdash de.m(\overline{de}) \Downarrow (\{\texttt{null}\}, \textit{true})}$$

RD-CALL-MULTI

$$\frac{\begin{array}{c} H \vdash de \Downarrow (\{c \rightarrow \alpha, \overline{c} \rightarrow \overline{\alpha}\}, E) \\ H \vdash \texttt{if } \{c \rightarrow \texttt{true}\} \texttt{ then } \{\alpha\}.m(\overline{de}) \texttt{ else } \{\overline{c} \rightarrow \overline{\alpha}\}.m(\overline{de}) \Downarrow (v', E') \end{array}}{H \vdash de.m(\overline{de}) \Downarrow (v', E \vee E')}$$

RD-EQUALS

$$\frac{\begin{array}{c} H \vdash de_0 \Downarrow (\{\overline{c} \rightarrow \overline{\alpha}\}, E_0) \quad H \vdash de_1 \Downarrow (\{\overline{d} \rightarrow \overline{\beta}\}, E_1) \\ v = \{\bigvee\{c_i \wedge d_i \mid \alpha_i = \beta_i\} \rightarrow \texttt{true}\} \end{array}}{H \vdash de_0 \texttt{ == } de_1 \Downarrow (v, E_0 \vee E_1)}$$

RD-AND

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

RD-OR

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

RD-NOT

$$\frac{\begin{array}{c} H \vdash de \Downarrow (\{c \rightarrow \texttt{true}\}, E) \\ v = \{\neg c \rightarrow \texttt{true}\} \end{array}}{H \vdash \texttt{!}de \Downarrow (v, E)}$$

**Figure 19.** 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.

The rule R-CALL performs method calls by inlining a method's body at the call site and substituting formal parameters with their actual symbolic values. The function $mbody(m, \alpha)$ denotes $\overline{x}.e$, where $e$ is the body of the method $m$ of object $\alpha$ 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 R-IF-T and R-IF-F for the `if` expressions depend on the concrete value of the branch condition $v$. If the concrete value is `true`, then $v = \texttt{true}$ is added to the path constraint. If the concrete value is `false`, then $v = \texttt{false}$ 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 4.4.

### 4.3 Translation of Declarative Methods

Figure 16 presents a declarative subset of the language in Figure 15, showing the syntax of declarative methods. Declarative methods may not contain object creations, assignments, or loops and may only call declarative methods.

Figure 19 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 $\Phi$. 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 $e$ evaluates 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` and `equalTo`), $v$ 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 $c$ that describes the conditions under which the method returns `true`.

### 4.4 Symbolic Execution of Declarative Expressions

Recall from Figure 18 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, PIPAL executes these operators symbolically without generating path constraints. The same applies to `if` statements. In general, in addition to calls to declarative methods, Pipal symbolically executes declarative expressions without generating path constraints (except for the exception condition) according to the following rule:

R-DECL

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

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. PIPAL updates the path constraint to reflect this requirement.

## 5. Experimental Results

This section presents our experimental results.

We implemented PIPAL as described in this paper. We extended the Polyglot [41] compiler to automatically instrument program modules to perform our dynamic analysis. We used MiniSat [14] 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 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 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, Polyglot compiler does not fully support 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 PIPAL 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 (c.f. Section 3.10) between the above modules and their abstractions. For comparison, we also checked these properties using JPF [48], KORAT [3], and BLAST [21]. We timed out all experiments after an hour.

The results of these experiments are in Figure 20. PIPAL exhaustively checked all `TreeMaps` with up to 63 nodes in under 15 minutes and all `HashMaps` with up to 64 nodes in under 40 minutes. The other checkers did not scale nearly as well. In particular BLAST could find no new predicates and aborted its analysis. This illustrates the ineffectiveness of most other techniques in checking these kinds of properties.

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 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 3.9, implemented using two `TreeMaps`. 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:

| Module | Max Number of Nodes | Time (s) JPF | KORAT | BLAST | PIPAL |
|---|---|---|---|---|---|
| TreeMap | 1 | 1.218 | 0.608 | aborted | 0.188 |
| | 2 | 5.556 | 0.613 | | 0.244 |
| | 3 | memory out | 0.676 | | 0.392 |
| | 4 | | 0.732 | | 0.485 |
| | 5 | | 1.251 | | 0.670 |
| | 6 | | 4.721 | | 0.751 |
| | 7 | | 23.547 | | 0.985 |
| | 8 | | 2202.231 | | 1.124 |
| | 9 | | timeout | | 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 | 0.674 | 0.465 | aborted | 0.176 |
| | 2 | 6.514 | 0.497 | | 0.227 |
| | 3 | memory out | 0.539 | | 0.258 |
| | 4 | | 0.810 | | 0.305 |
| | 5 | | 0.735 | | 0.373 |
| | 6 | | 0.819 | | 0.422 |
| | 7 | | 0.997 | | 0.494 |
| | 8 | | 3.077 | | 0.599 |
| | 9 | | 18.972 | | 0.675 |
| | 10 | | 150.932 | | 0.780 |
| | 11 | | 1203.986 | | 0.953 |
| | 12 | | timeout | | 1.162 |
| | 13 | | | | 1.356 |
| | 14 | | | | 1.708 |
| | 15 | | | | 2.143 |
| | 16 | | | | 2.879 |
| | ... | | | | ... |
| | 32 | | | | 75.139 |
| | 64 | | | | 2004.723 |
| TreeSet | 1 | 0.537 | 0.638 | aborted | 0.195 |
| | 2 | 0.950 | 0.648 | | 0.246 |
| | 3 | 26.816 | 0.693 | | 0.393 |
| | 4 | memory out | 1.020 | | 0.489 |
| | 5 | | 0.699 | | 0.651 |
| | 6 | | 0.876 | | 0.752 |
| | 7 | | 0.943 | | 0.961 |
| | 8 | | 2.773 | | 1.090 |
| | 9 | | 8.208 | | 1.473 |
| | 10 | | 24.921 | | 1.665 |
| | 11 | | 74.966 | | 2.238 |
| | 12 | | 221.719 | | 2.493 |
| | 13 | | 615.524 | | 3.065 |
| | 14 | | 1706.41 | | 3.399 |
| | 15 | | timeout | | 4.481 |
| | ... | | | | ... |
| | 31 | | | | 39.789 |
| | 63 | | | | 796.955 |
| HashSet | 1 | 0.728 | 0.520 | aborted | 0.171 |
| | 2 | 0.781 | 0.511 | | 0.221 |
| | 3 | 6.574 | 0.716 | | 0.248 |
| | 4 | memory out | 0.570 | | 0.299 |
| | 5 | | 0.567 | | 0.363 |
| | 6 | | 0.623 | | 0.414 |
| | 7 | | 0.701 | | 0.478 |
| | 8 | | 1.878 | | 0.579 |
| | 9 | | 3.357 | | 0.644 |
| | 10 | | 9.440 | | 0.773 |
| | 11 | | 29.485 | | 0.948 |
| | 12 | | 82.642 | | 1.068 |
| | 13 | | 238.420 | | 1.344 |
| | 14 | | 593.045 | | 1.608 |
| | 15 | | 952.317 | | 2.029 |
| | 16 | | timeout | | 2.816 |
| | ... | | | | ... |
| | 32 | | | | 68.011 |
| | 64 | | | | 2543.034 |

**Figure 20.** Results of checking modules against abstractions. PIPAL checks `TreeMaps` of up to 63 nodes in under 15 minutes.

- `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 PIPAL to check the given implementations of these programs. We then replaced the maps with abstract maps and checked the programs again with PIPAL. We checked maps with at most $n$ nodes. We checked `IntCounter` with at most $n$ integers 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 in Figure 21. Checking these programs with `AbsMap` is significantly faster than checking them with a `TreeMap` or a `HashMap`.

## 6. Related Work

This section presents related work on software model checking. 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 [16] is a stateless model checker for C programs. Java PathFinder (JPF) [31, 48] is a stateful model checker for Java programs. XRT [20] checks Microsoft CIL programs. Bandera [7] and JCAT [9] translate Java programs into the input language of model checkers like SPIN [22] and SMV [36]. Bogor [13] is an extensible framework for building software model checkers. CMC [39] is a stateful model checker for C programs that has been used to test large software including the Linux implementation of TCP/IP and the ext3 file system. Chess [38] and CalFuzzer [28] 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 [1], Blast [21], and Magic [4] use heuristics to construct and check an abstraction of a program (usually predicate abstraction [19]).

| Module | Max Number of Nodes | Time (s) Original Program | Time (s) 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 |

**Figure 21.** Experimental results of checking programs that use a map internally. Replacing the map with an abstract implementation speeds up the checking considerably.

Abstractions that are too coarse generate false positives, which are then used to refine the abstraction and redo the checking. This technique is known as Counter Example Guided Abstraction and Refinement or *CEGAR*. There are also many static [16] and dynamic [15] partial order reduction systems for concurrent programs. There are many other symmetry-based reduction techniques as well (e.g., [25]). 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 PIPAL to other model checkers support this observation.

Tools such as Alloy [26, 30] and Korat [3] systematically generate all test inputs that satisfy a given precondition. A version of JPF [31] uses lazy initialization of fields to essentially simulate the Korat algorithm. Kiasan [10] 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 PIPAL. 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 [17, 44], Whispec [45], and a version of JPF [47] 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 [46], Miniatur [12], and Forge [11] 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.

This paper uses and extends our previous work on glass box software model checking. Our previous work focused on establishing data structure invariants [8] and type soundness [43]. The work on data structure invariants could only efficiently check *local* properties of data structures, which relate a node to its immediate neighbors. The work on type soundness improved on this by allowing nonlocal properties to be efficiently checked as well and applied it to the checking of type soundness. This paper extends our previous work to support modular glass box checking. It also presents a formal description of our dynamic and static analyses.

## 7. Conclusions

We present PIPAL, a system for modular 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. PIPAL first checks a program module against an abstract implementation, establishing functional equivalence. PIPAL then replaces the program module with the abstract implementation when checking other program modules. Our experimental results indicate that the modular glass box software model checking approach is effective and it significantly outperforms the earlier non-modular glass box software model checking approach. A comparison of PIPAL to other state of the art software model checkers demonstrates that PIPAL is significantly more efficient in checking data-oriented programs with respect to complex data-dependent properties.

## Acknowledgments

This research was supported in part by the AFOSR Grant FA9550-07-1-0077.

## References

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

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

[3] 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.

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

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

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

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

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

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

[10] 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.

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

[12] 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.

[13] 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.

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

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

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

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

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

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

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

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

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

[23] 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.

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

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

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

[27] 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.

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

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

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

[31] 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.

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

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

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

[35] 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.

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

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

[38] M. Musuvathi, 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.

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

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

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

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

[43] 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.

[44] 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.

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

[46] 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.

[47] 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.

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