Efficient Software Model Checking of Soundness of Type Systems

Michael Roberson Melanie Harries Paul T. Darga Chandrasekhar Boyapati

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

Abstract

This paper presents novel techniques for checking the soundness of a type system automatically using a software model checker. Our idea is to systematically generate every type correct intermediate program state (within some finite bounds), execute the program one 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 detecting similarities in this search space and pruning 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 hold for the language (albeit for program states of at most some finite size). Our preliminary experimental results on several languages—including a language of integer and boolean expressions, a simple imperative programming language, an object-oriented language which is a subset of Java, and a language with ownership types—indicate that our approach is feasible and that our search space pruning techniques do indeed significantly reduce what is otherwise an extremely large search space. Our paper thus makes contributions both in the area of checking soundness of type systems, and in the area of reducing the state space of a software model checker.

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 and Verifying and Reasoning about Programs

General Terms Languages, Reliability, Verification

Keywords Software Model Checking, Type Soundness

OOPSLA'08, October 19–23, 2008, Nashville, Tennessee, USA.

1. Introduction

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, 28, 45, 46, 54].

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., [20]) or machine checkable proofs (e.g., [47])—require significant manual effort.

This paper presents an alternate approach for checking type soundness *automatically* using a 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 detecting similarities in this search space and pruning 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 [50, 56] hold for the language (albeit for program states of at most some finite size).

Our experimental results on several languages—including the language of integer and boolean expressions from [50, Chapters 3 & 8], a typed version of the imperative language IMP from [55, Chapter 2], an object-oriented language which is a subset of Java, and a language with ownership types [1, 6, 13]—indicate that our approach is feasible and that our search space pruning techniques do indeed significantly reduce what is otherwise an extremely large search space. This paper thus offers a promising approach for checking type soundness automatically, thereby enabling the design of novel type systems. In particular, this can enormously help programming language designers in debugging their language specifications. Currently there is no other technology around to automate this task effectively.

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* [35], has been experimentally verified in several domains. Our preliminary experiments using mutation testing [49, 41] 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.

This paper also makes contributions in improving the state of art in software model checking [3, 4, 11, 14, 15, 21, 24, 53, 29, 44]. Model checking is a formal verification technique that exhaustively tests a circuit/program on all possible inputs (sometimes up to a given size) to handle *input nondeterminism* and on all possible nondeterministic schedules to handle *scheduling nondeterminism*. For hardware, model checkers have been successfully used to verify fairly complex finite state control circuits with up to a few hundred bits of state information; but not circuits in general that have large data paths or memories. Similarly, for software, model checkers have been primarily used to verify control-oriented programs (with scheduling nondeterminism) with respect to temporal properties; but not much work has been done to verify data-oriented programs (with input nondeterminism) with respect to complex data-dependent properties.

Thus, while there is much research on state space reduction techniques for software model checkers such as partial order reduction [23, 24] and tools based on predicate abstraction [26] such as Slam [3], Blast [29], or Magic [11], none of these techniques seem to be effective in reducing the state space when checking the soundness of a type system—where one must deal with input nondeterminism (to check every input program state) and data-dependent properties (type correctness properties that depend on input program states). In fact, because of input nondeterminism, it is difficult to even formulate the problem of automatically checking type soundness in the context of most model checkers.

This paper describes techniques for efficiently checking type soundness automatically using a software model checker by significantly reducing the state space of the model checker. It thus contributes to improving the state of art in software model checking. This paper builds on our recent previous work on model checking properties of data structures [16]. This paper improves on the techniques presented in [16] and applies them to checking soundness of type systems.

The rest of this paper is organized as follows. Section 2 illustrates our approach with an example. Section 3 describes our software model checker for checking of soundness type systems. Section 4 presents our experimental results. Section 5 discusses related work and Section 6 concludes.

2. Example

This section illustrates our approach with an example. Consider the language of integer and boolean expressions in [50, Chapters 3 & 8]. The syntax of the language is shown in Figure 1. The small step operational semantics and the type checking rules for this language are in [50]. 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 2 shows three abstract syntax trees (ASTs) `t1`, `t2`, and `t3`. AST `t1` represents 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`'. Figure 2 shows the ASTs 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

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

Figure 1. Abstract syntax of the language of integer and boolean expressions from [50, Chapters 3 & 8].

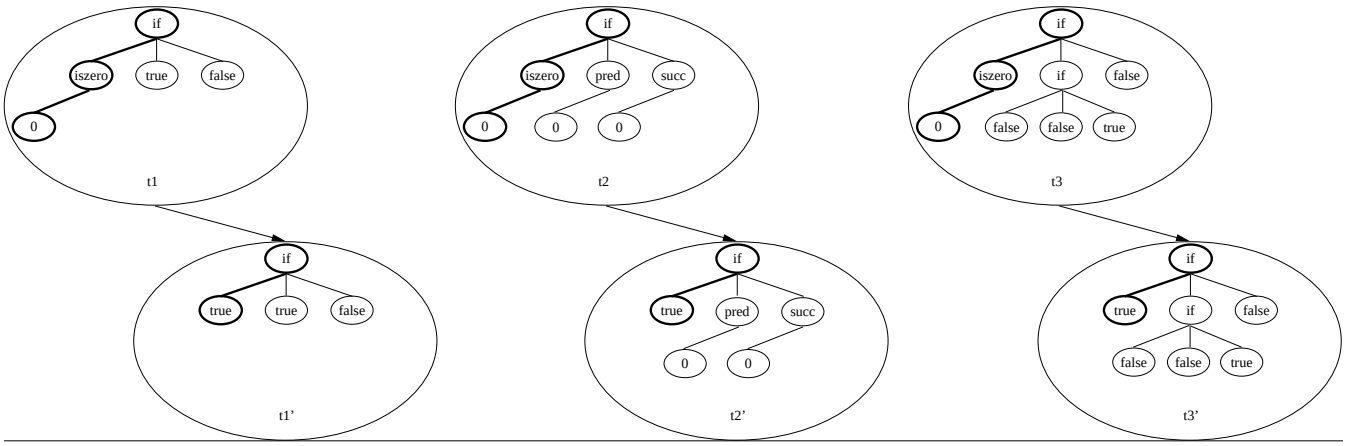

Figure 2. Three abstract syntax trees (ASTs) for the language in Figure 1, before and after a small step evaluation. The tree path touched by each evaluation is highlighted. Note that the tree path is the same in all three cases. Once our system checks the progress and preservation theorems on AST `t1`, it determines that it is redundant to check them on ASTs `t2` and `t3`.

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 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.

3. Model Checking Type Soundness

While the basic idea presented in Section 2 is simple, one must overcome several technical challenges to make it work well in practice. This section describes our approach.

3.1 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 (c.f. Section 3.6). The type system must be specified in a declarative language to facilitate our static analysis (c.f. Section 3.7). The operational semantics, however, 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 [40].

In our current system, we use Java as our executable language for specifying an operation semantics. We use a variant of Java as our declarative language for specifying a type system. We use Java similarly to our previous work [16] for specifying finite bounds on the size of intermediate program states. Figure 3 shows an example implementation of the expression language in Figure 1 in our system. An object of class `ExpressionLanguage` represents an intermediate program state of the expression language. Every such class that implements `Language` must have two 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; and ii) a *declarative* method `wellTyped` that returns true iff the corresponding intermediate program state is well typed. Declarative methods are annotated as `Declarative`. A declarative method may not contain object creations, assignments, loops, or exception handlers, and may only call other declarative methods. A declarative method may however contain implications, universal quantifications, and existential quantifications to facilitate writing first order logic formulas. We allow Java methods to call declarative methods—we automatically translate a declarative method into executable code before running it. Finally, the `Tree` annotations on Lines 14 and 70 denote that the expression structure forms a tree, similarly to [16]. Such annotations reduce the search space of our model checker because it does not have to check non-tree structures.

We note that our model checking techniques are not tied to our above choice of specification languages and can also be made to work with other languages (e.g., Ott [51]).

```
1  public 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           return
19             syntaxOk() &&
20             ( kind == TRUE   ==> true                              ) &&
21             ( kind == FALSE  ==> true                              ) &&
22             ( kind == ZERO   ==> true                              ) &&
23             ( kind == SUCC   ==> e1.wellTyped() && e1.type() == INT ) &&
24             ( kind == PRED   ==> e1.wellTyped() && e1.type() == INT ) &&
25             ( kind == ISZERO ==> e1.wellTyped() && e1.type() == INT ) &&
26             ( kind == IF     ==> e1.wellTyped() && e1.type() == BOOL && e2.wellTyped() && e3.wellTyped() && e2.type()==e3.type() );
27         }
28
29         Expression smallStep() throws StuckException {
30             if ( isValue()     ) {                      return this;                }
31             if ( e1 == null    ) {                      throw new StuckException(); }
32             if ( !e1.isValue() ) { e1 = e1.smallStep(); return this;                }
33
34             if ( kind == PRED   && e1.kind == ZERO  ) return e1;
35             if ( kind == PRED   && e1.kind == SUCC  ) return e1.e1;
36             if ( kind == ISZERO && e1.kind == ZERO  ) return True();
37             if ( kind == ISZERO && e1.kind == SUCC  ) return False();
38             if ( kind == IF     && e1.kind == TRUE  ) return e2;
39             if ( kind == IF     && e1.kind == FALSE ) return e3;
40
41             throw new StuckException();
42         }
43
44         // Helper functions
45
46         @Declarative
47         boolean syntaxOk() {
48           return
49             ( ( kind == TRUE || kind == FALSE || kind == ZERO   ) && e1 == null && e2 == null && e3 == null ) ||
50             ( ( kind == SUCC || kind == PRED  || kind == ISZERO ) && e1 != null && e2 == null && e3 == null ) ||
51             ( ( kind == IF                                      ) && e1 != null && e2 != null && e3 != null );
52         }
53
54         @Declarative
55         int type() {
56             if        ( kind == TRUE || kind == FALSE || kind == ISZERO )   return BOOL;
57             else if   ( kind == ZERO || kind == SUCC  || kind == PRED   )   return INT;
58             else    /*( kind == IF                                      )*/ return e2.type();
59         }
60
61         @Declarative
62         boolean isValue() {
63             return kind == TRUE || kind == FALSE || kind == ZERO || kind == SUCC && e1.isValue();
64         }
65
66         static Expression True () {Expression e = new Expression(); e.kind = TRUE;  return e;}
67         static Expression False() {Expression e = new Expression(); e.kind = FALSE; return e;}
68     }
69
70     @Tree Expression root;
71
72     @Declarative
73     public boolean wellTyped()                       { return root != null && root.wellTyped(); }
74
75     public void    smallStep()    throws StuckException { root = root.smallStep();                }
76
77     public boolean isFinalState()                    { return root.isValue();                    }
78 }
```

Figure 3. An implementation of the language of integer and boolean expressions in Figure 1 in our system.

Field	Domain
n0.kind	{TRUE, FALSE, ZERO, SUCC, PRED, ISZERO, IF}
n1.kind	{TRUE, FALSE, ZERO, SUCC, PRED, ISZERO, IF}
n2.kind	{TRUE, FALSE, ZERO, SUCC, PRED, ISZERO, IF}
n3.kind	{TRUE, FALSE, ZERO, SUCC, PRED, ISZERO, IF}
n4.kind	{TRUE, FALSE, ZERO, SUCC, PRED, ISZERO, IF}
n5.kind	{TRUE, FALSE, ZERO, SUCC, PRED, ISZERO, IF}
n6.kind	{TRUE, FALSE, ZERO, SUCC, PRED, ISZERO, IF}
n7.kind	{TRUE, FALSE, ZERO, SUCC, PRED, ISZERO, IF}
n8.kind	{TRUE, FALSE, ZERO, SUCC, PRED, ISZERO, IF}
n9.kind	{TRUE, FALSE, ZERO, SUCC, PRED, ISZERO, IF}
n10.kind	{TRUE, FALSE, ZERO, SUCC, PRED, ISZERO, IF}
n11.kind	{TRUE, FALSE, ZERO, SUCC, PRED, ISZERO, IF}
n12.kind	{TRUE, FALSE, ZERO, SUCC, PRED, ISZERO, IF}
n0.e1	{null, n1}
n0.e2	{null, n2}
n0.e3	{null, n3}
n1.e1	{null, n4}
n1.e2	{null, n5}
n1.e3	{null, n6}
n2.e1	{null, n7}
n2.e2	{null, n8}
n2.e3	{null, n9}
n3.e1	{null, n10}
n3.e2	{null, n11}
n3.e3	{null, n12}

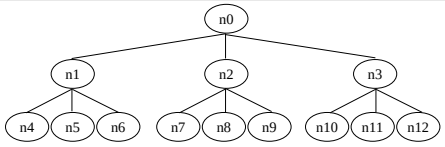

Figure 4. Search space for the language implemented in Figure 3 with ASTs of height at most 3.

Field	Value
n0.kind	IF
n1.kind	ISZERO
n2.kind	IF
n3.kind	FALSE
n4.kind	ZERO
n5.kind	*
n6.kind	*
n7.kind	FALSE
n8.kind	FALSE
n9.kind	TRUE
n10.kind	*
n11.kind	*
n12.kind	*
n0.e1	n1
n0.e2	n2
n0.e3	n3
n1.e1	n4
n1.e2	null
n1.e3	null
n2.e1	n7
n2.e2	n8
n2.e3	n9
n3.e1	null
n3.e2	null
n3.e3	null

Field	Value
n0.kind	ISZERO
n1.kind	FALSE
n2.kind	*
n3.kind	*
n4.kind	*
n5.kind	*
n6.kind	*
n7.kind	*
n8.kind	*
n9.kind	*
n10.kind	*
n11.kind	*
n12.kind	*
n0.e1	n1
n0.e2	null
n0.e3	null
n1.e1	null
n1.e2	null
n1.e3	null
n2.e1	*
n2.e2	*
n2.e3	*
n3.e1	*
n3.e2	*
n3.e3	*

Figure 5. Two elements of the search space in Figure 4. The first element represents the term '`if (iszero 0) then (if false then false else true) else false`'. The second element represents the term '`iszero false`'. The symbol * denotes don't care.

3.2 Search Space

Traditional software model checkers [3, 11, 15, 21, 24, 53, 29, 44] explore a state space by starting from the initial state and systematically generating and checking every successor state. While this approach works well to check software with scheduling nondeterminism, it is not convenient to check software with input nondeterminism. In fact, it is difficult to even formulate the problem of checking type soundness in the context of most software model checkers. Instead, our model checker organizes its search space as follows.

Consider the language in Figure 3. Suppose our system must check the progress and preservation theorems on all ASTs up to a maximum height h=3. Figure 4 shows the corresponding search space. The search space consists of all possible assignments to the fields, where each field gets a value from its corresponding domain. Every element of this search space is an AST. For example, the first element in Figure 5 corresponds to the AST '`if (iszero 0) then (if false then false else true) else false`'. In Figure 4, there are thirteen fields with seven elements in their domains and twelve fields with two elements in their domains, so the size of this search space is $7^{13} * 2^{12}$. In general, for ASTs of height at most h, the size of the search space is $7^{\frac{3^h-1}{2}} * 2^{\frac{3^h-3}{2}}$. Note that many elements of this search space are not type correct or even syntactically correct. For example, the second element in Figure 5 is not type correct because `iszero` cannot be invoked on `false`.

In general, the intermediate state of a program can include other components besides an AST, such as a dynamically allocated heap. Our system appropriately constructs a finite search space that includes all such components.

3.3 Search Algorithm

Figure 6 presents the pseudo-code for our search algorithm. Given a language to check for type soundness and finite bounds on the size of its intermediate program states, our system first initializes the search space to the set of all well typed program states within the finite bounds. It then systematically explores this space by repeatedly selecting a program state `w` from the search space, running its analyses to identify a set of program states `W'` (including `w`) on which `smallStep` (described in Section 3.1) behaves similarly to `w`, checking that the progress and preservation theorems hold on every program state in `W'`, and pruning all the program states in `W'` from the search space. The next sections describe how to perform various steps of the above search efficiently.

3.4 Search Space Representation

Consider the search space of the language in Figure 3, with ASTs of height at most h=8. The size of this search space is about 2^{12487}. Of these, about 2^{2523} ASTs are type correct. However, as our experiments show, our system checks the progress and preservation theorems explicitly on only 41 ASTs. (Our analyses determine that it is redundant to check the theorems on the remaining elements of the search space.) Thus, if we are not careful, search space management itself could take exponential time and negate the benefits of

```
1 void search( SearchSpace S ) {
2     W = { w ∈ S | w.wellTyped() }
3     while ( W ≠ ∅ ) {
4         w = Any element in W
5         W′ = { w′ ∈ W | smallStep behaves similarly on w & w′ }
6         Check progress and preservation on all states in W′
7         W  = W - W′
8     }
9 }
```

Figure 6. Pseudo-code for the search algorithm.

our search space pruning techniques. We avoid this by using a compact representation of the search space (that is, the set of intermediate program states). We explored two different approaches for representing the search space: (i) using a *reduced ordered binary decision diagram* [10] or BDD, as in our previous work [16], and (ii) using an incremental SAT solver, MiniSat [22]. In our experiments, we found that while the BDD-based approach performs slightly better on tree-based type constraints, the SAT-based approach performs much better on languages that include non-tree-based type constraints (that is, on languages whose program states include components other than ASTs). We therefore discuss only our SAT-based approach in the rest of this paper.

Our SAT-based approach works as follows. We represent a set of program states as a finite propositional logic formula. For example, for the search space in Figure 4, the formula (`n0.kind=IF` $\wedge$ `n1.kind=ISZERO` $\wedge$ `n4.kind=ZERO` $\wedge$ `n0.e1=n1` $\wedge$ `n0.e2=n2` $\wedge$ `n0.e3=n3` $\wedge$ `n1.e1=n4`) represents the set of all the terms of the form '`if (iszero 0) then x1 else x2`', where `x1` and `x2` are any two terms. This includes the terms represented by ASTs `t1`, `t2`, and `t3` in Figure 2. Every satisfying assignment of the formula represents a member of the set. If the formula is unsatisfiable, then the set is empty. We use $\lceil \log_2 d \rceil$ bits to encode a field with domain size d. For example, we use 3 bits to encode a `kind` field , and 1 bit to encode an `e1`, `e2`, or `e3` field. We use an incremental SAT solver to find satisfying assignments of the propositional logic formula. Line 7 in Figure 6, computing the difference of two sets, thus takes time linear in the size of the formula because it simply injects clauses into the incremental SAT solver. Line 3, checking if a set is empty, and Line 4, choosing an element of a non-empty set, could be expensive operations because they invoke the SAT solver.

3.5 Search Space Initialization

Our search begins by initializing the search space to the set of all well typed intermediate program states (Line 2 in Figure 6). We do this by automatically translating the declarative method `wellTyped` (described in Section 3.1) and the declarative methods it transitively invokes, given the finite bounds on the size of intermediate program states, into a finite propositional logic formula. The translation process is somewhat similar to that of AAL [37]. However, because our declarative methods do not contain object creations, assignments, loops, or exception handlers, the formulas for declarative methods 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 translates Java into Alloy [34] and translates Alloy into propositional logic.

3.6 Dynamic Analysis

This section presents our basic search space pruning technique. Consider the language implemented in Figure 3. Consider checking the progress and preservation theorems on the AST represented by the first element in Figure 5. The theorems hold on the AST. As our system evaluates the AST a small step forward, it monitors the fields that the small step evaluator reads. In this case, `smallStep` reads `n0.kind`, `n1.kind`, `n4.kind`, `n0.e1`, and `n1.e1`. That means, regardless of the values of the remaining fields, the small step evaluator will still behave similarly if the values of the fields that were read do not change. Our system then determines that regardless of the values of the remaining fields, if the AST is well typed before the small step evaluation, then the AST will be well typed after the small step evaluation. Our system therefore prunes all elements of the search space where (`n0=IF` $\wedge$ `n1=ISZERO` $\wedge$ `n4=ZERO` $\wedge$ `n0.e1=n1` $\wedge$ `n1.e1=n4`). This is the basic idea that makes our approach of exhaustive testing within a large but finite domain feasible.

3.7 Static Analysis

The dynamic analysis described above in effect detects *don't care* fields in a well typed program state w, and suggests that all states w' that differ from w only at the don't care fields be pruned from the search space. The goal of the static analysis is to prove that it is indeed safe to prune those states. To see why the static analysis is necessary, consider the following simple but artificial example where `wellTyped` returns true iff `a` implies `b`. Suppose we invoke `smallStep` on `a=false` and `b=true`. `wellTyped` returns true before and after the execution of `smallStep`. `smallStep` reads only the field `a` while the field `b` is a don't care. The dynamic analysis above suggests that the progress and preservation theorems might hold on all states where `a=false` (and therefore those elements be pruned from the search space). But the suggestion is incorrect because the preservation theorem does not hold on `a=false` and `b=false`. `wellTyped` returns `true` before the evaluation of `smallStep` but returns `false` after.

```
1  class WhyStaticAnalysis extends Language {
2    private boolean a, b;
3    @Declarative
4    public  boolean wellTyped() {return a ==> b;}
5    public  void    smallStep() throws StuckException {a = !a;}
6  }
```

Our static analysis works as follows. Consider checking the progress and preservation theorems on a program state w, with fields $\mathtt{f}_{1..n}$. Of these, without loss of generality, suppose `smallStep` neither writes to nor reads the original values of

fields $f_{1..m}$, writes to but does not read the original values of fields $f_{(m+1)..k}$, and both writes to and reads the original values of fields $f_{(k+1)..n}$. Our dynamic analysis then identifies fields $f_{1..k}$ as don't cares. Let the values of fields $f_{1..n}$ be $v_{1..n}$ before `smallStep`, and $v'_{1..n}$ after `smallStep`. Recall from Section 3.5 that our system automatically translates the declarative method `wellTyped`, given the finite bounds on the size of program states, into a finite propositional logic formula. Let `wellTyped`($f_{1..n}$) denote that formula. Our static analysis then attempts to prove that for all values of $f_{1..k}$ in the bounded domain, `wellTyped`($f_{1..k}$ $v_{(k+1)..n}$) implies `wellTyped`($f_{1..m}$ $v'_{(m+1)..n}$). Our system invokes the SAT solver to check if the implication holds. If the implication holds, our system safely prunes the search space as described in Section 3.6. If the implication does not hold, then there is an error in the type system. An instance satisfying the negation of the implication exposes the error.

The above analysis requires careful handling if `smallStep` rearranges fields. Consider the search space in Figure 4. Consider checking the term '`if true then (if false then false else false) else true`'. A small step of evaluation yields the term '`if false then false else false`'. Note that even though `smallStep` does not read the fields of nodes `n2`, `n7`, `n8`, and `n9` in Figure 4, it moves the nodes into the positions of nodes `n0`, `n1`, `n2`, and `n3`. Thus, for example, the field `n2.kind` before `smallStep` moves into the position of `n0.kind` after `smallStep`. Our system tracks such rearrangement of fields and appropriately constructs the formula for the static analysis described above.

3.8 Symbolic Execution

The dynamic analysis in Section 3.6 identifies a usually large set W' of program states on which `smallStep` behaves similarly (Line 5 in Figure 6). The static analysis in Section 3.7 efficiently checks that the progress and preservation theorems hold on all the program states in W' (Line 6 in Figure 6), so that W' can be pruned from the search space. This section describes optimizations that enable our system to identify and prune an even larger set W' from the search space.

Consider `smallStep` in the example below. Suppose it is invoked on `b1=true`, `b2=true`, `b3=true`, `b4=true`, and `b5=true`. The dynamic analysis in Section 3.6 detects that `smallStep` only reads fields `b1`, `b2`, `b3`, and `b5`, and thus determines that if the values of those fields remain unchanged, `smallStep` will behave similarly. Suppose the declarative method `wellTyped` gets automatically translated into the formula `wellTyped(b1, b2, b3, b4, b5)`, as described in Section 3.7. The static analysis in Section 3.7 then attempts to prove using a SAT solver that for all values of `b4`, `wellTyped(true, true, true, b4, true)` implies `wellTyped(true, true, true, false, true)`. If the implication holds, our system prunes from the search space all states where (`b1=true` $\wedge$ `b2=true` $\wedge$ `b3=true` $\wedge$ `b5=true`).

```
1  class WhySymbolicExecution extends Language {
2    private boolean b1, b2, b3, b4, b5;
3    @Declarative
4    public boolean wellTyped() {...}
5    public void smallStep() throws StuckException {
6        if ( b1 == b2 ) if ( b1 == b3 ) b4 = !b5;
7    }
8  }
```

In the above example, even though the field `b5` is read, it does not affect the control flow of `smallStep`. Moreover, even though the fields `b1`, `b2`, and `b3` are read, the control flow of `smallStep` remains the same if (`b1=b2` $\wedge$ `b1=b3`). Our system uses symbolic execution [39] to identify that `smallStep` behaves similarly, that is, follows the same control flow path, on all program states where (`b1=b2` $\wedge$ `b1=b3`). Our system uses symbolic execution to also identify that the value of the field `b4` after `smallStep` is the negation of the value of the field `b5` before `smallStep`. Our static analysis then attempts to prove using a SAT solver that for all values of `b1`, `b2`, `b3`, `b4`, and `b5`, `wellTyped(b1, b2, b3, b4, b5)` and (`b1=b2` $\wedge$ `b1=b3`) implies `wellTyped(b1, b2, b3,` $\neg$`b5, b5)`. If the implication holds, our system prunes from the search space all states where (`b1=b2` $\wedge$ `b1=b3`).

Our system currently symbolically executes assignments, comparisons, and boolean operations. Our system also symbolically executes declarative methods by automatically translating declarative methods into propositional logic formulas, as described in Section 3.5, and replacing every call to a declarative method with its corresponding propositional logic formula. For example, when symbolically executing Line 30 or Line 32 in Figure 3, our system replaces the call to the declarative method `isValue` with its corresponding propositional logic formula. Our system uses symbolic execution to build a path constraint formula which when true guarantees that `smallStep` will follow the same control flow path. Our system then performs a static analysis as described above and prunes all states that satisfy the formula. If our system is unable to symbolically execute an operation on a field `f` with original value `v`, it simply treats the field concretely instead of symbolically and adds the clause (`f=v`) to the path constraint formula (that is, our system falls back to the dynamic analysis in Section 3.6 w.r.t. that field).

3.9 Isomorphism Analysis

Consider a language whose intermediate program states include an AST and a dynamically allocated heap. Figure 7 presents an example of such a search space, where every object contains one pointer, the AST has height at most 2, and there are at most 4 heap objects. Consider the two elements of the above search space in Figure 8. These two elements are isomorphic because `o0` and `o1` are equivalent memory locations. Therefore, once we check the progress and preservation theorems on the first element, it is redundant to check the theorems on the second element. Our system avoids checking isomorphic structures as follows. Suppose the small step evaluator reads only `n0.value`, `n1.value`,

Field	Domain
n0.value	{EQUALS, ..., o0, o1, o2, o3, null}
n1.value	{EQUALS, ..., o0, o1, o2, o3, null}
n2.value	{EQUALS, ..., o0, o1, o2, o3, null}
o0.value	{o0, o1, o2, o3, null}
o1.value	{o0, o1, o2, o3, null}
o2.value	{o0, o1, o2, o3, null}
o3.value	{o0, o1, o2, o3, null}

AST Heap

Figure 7. Search space for a language whose intermediate program states include an AST and a heap.

Field	Value
n0.value	EQUALS
n1.value	o0
n2.value	o1
o0.value	null
o1.value	null
o2.value	null
o3.value	null

Field	Value
n0.value	EQUALS
n1.value	o1
n2.value	o0
o0.value	null
o1.value	null
o2.value	null
o3.value	null

Figure 8. Two isomorphic elements of the space in Figure 7.

and `n2.value` when evaluating the first element, and suppose that the analyses in the previous sections conclude that all states where (`n0.value=EQUALS` $\wedge$ `n1.value=o0` $\wedge$ `n2.value=o1`) can be pruned. Our isomorphism analysis then determines that all structures that satisfy the following formula can also be safely pruned: (`n0.value=EQUALS` $\wedge$ `n1.value=o1/o2/o3` $\vee$ `n0.value=EQUALS` $\wedge$ `n1.value=o0` $\wedge$ `n2.value=o2/o3`).

In general, given a program state `w`, our system constructs such a formula $\mathtt{I}_w$ denoting the set of states isomorphic to `w` as follows. Recall from Section 3.8 that our symbolic execution on `w` builds a path constraint formula, say $\mathtt{P}_w$. Suppose during symbolic execution our system encounters a fresh object `o` that a field `f` points to, and the path constraint built so far is $\overline{\mathtt{P}}_w$. Our isomorphism analysis includes in $\mathtt{I}_w$ all states that satisfy ($\overline{\mathtt{P}}_w \wedge$ `f=o`$'$), for every `o`$'$ in the domain of the field `f` that is another fresh object. Our system then prunes all the states denoted by $\mathtt{I}_w$ from the search space.

Note that some software model checkers also prune isomorphic program states using heap canonicalization [32, 43]. 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 our system, once our checker checks a state `w`, it computes a compact formula $\mathtt{I}_w$ denoting a (often exponentially large) set of states isomorphic to w, and prunes $\mathtt{I}_w$ from the search space. Our checker *never visits* the (often exponentially many) states in $\mathtt{I}_w$.

3.10 Handling Special Cases

Our system also handles the following special cases.

3.10.1 Handling Term Cloning

Consider the following semantics for the `while` statement of the imperative language IMP from [55, Chapter 2], which clones the entire loop body. σ contains values of variables.

$\langle$`while c do b`$, \sigma\rangle \rightarrow \langle$`if c then (b; while c do b)`$, \sigma\rangle$

The cloning of different loop bodies could make `smallStep` follow different control flow paths. However, in one iteration of the loop in Figure 6, 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 loop in Figure 6, our system provides a special construct to implement cloning and replaces a cloning operation with an automatically generated formula during symbolic execution.

Other examples of cloning include method calls that have a method inlining semantics (e.g., in Featherweight Java [31]).

3.10.2 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.

3.10.3 Handling Nondeterministic Languages

The discussion so far assumes deterministic languages. Consider a language `L` with 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), w.r.t. every transition that is enabled on the state.

```
1  class L extends NondeterministicLanguage {
2    @Declarative
3    public boolean wellTyped() {...}
4    public void smallStep(int x) throws StuckException {...}
5  }
```

4. Experimental Results

This section presents our preliminary experimental results. We implemented a rudimentary software model checker as described in this paper. We extended the Polyglot [48] compiler framework to automatically instrument the operational semantics of languages to perform our dynamic analyses.

We used MiniSat [22] 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 present results for the following languages, each with increasing complexity:

1. The language of integer and boolean expressions from [50, Chapters 3 & 8], as implemented in Figure 3.
2. A typed version of the imperative language IMP from [55, 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 [31].

   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, 6, 13], 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 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.

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

Figure 9. Experimental results for 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.

Finally, Figure 10 presents our experimental 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* [35, 41, 49]. 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.

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

Figure 10. 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 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 (c.f. [5, Figure 24]) into Ownership Java such that the *owners as dominators* property does not hold. Our checker correctly detected the bug.

The results in Figure 10, while preliminary, do indicate that exhaustive testing within a small finite domain is an effective approach for checking soundness of type systems. 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. This lends credibility to the validity of the small scope hypothesis in practice.

5. 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) to handle *input nondeterminism* and all possible nondeterministic schedules to handle *scheduling nondeterminism*. There has been much research on model checking of software. Verisoft [24] is a stateless model checker for C programs. Java PathFinder (JPF) [53, 38] is a stateful model checker for Java programs. XRT [27] checks Microsoft CIL programs. Bandera [15] and JCAT [18] translate Java programs into the input language of model checkers like SPIN [30] and SMV [42]. Bogor [21] is an extensible framework for building software model checkers. CMC [44] 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.

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 scheduling nondeterminism) with respect to temporal properties; but not much work has been done to verify data-oriented programs (with input nondeterminism) 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 programs with scheduling nondeterminism. Tools such a Slam [3], Blast [29], and Magic [11] use heuristics to construct and check an abstraction of a program (usually predicate abstraction [26]). 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 [24] and dynamic [23] partial order reduction systems for concurrent programs. There are many other symmetry-based reduction techniques as well (e.g., [33]). However, none of the above techniques seem to be effective in reducing the state space of a model checker when checking the soundness of a type system—where one must deal with input nondeterminism (to check every input program state) and data-dependent properties (type correctness properties that depend on input program states). In fact, 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.

Tools such as Alloy [34, 36] and Korat [4] systematically generate all test inputs that satisfy a given precondition. A version of JPF [38] uses lazy initialization of fields to essentially simulate the Korat algorithm. However, these tools generate and test every valid state and so do not achieve as much state space reduction as our system.

Jalloy [52] and Miniatur [19] translate a Java program and its specifications into a SAT formula and verify it with a SAT solver. We experimented with a similar approach by translating both the operational semantics and the type system of a language into a SAT formula and verifying it with a SAT solver. However, translating operational semantics into SAT usually led to large formulas and the approach was less efficient than the model checker described in this paper.

This paper builds on our recent previous work on model checking properties of tree-based data structures [16]. This paper improves on the techniques presented in [16] and applies them to checking soundness of type systems.

A recent paper [12] describes a technique for checking properties of programming languages specified in αProlog, using a bounded backtracking search in an αProlog interpreter.

However, [12] does not use our search space reduction techniques and does not scale as well as our model checker.

6. Conclusions

This paper presents a software model checker 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. 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.

We have tested our system on several small to medium sized languages that include several features such as term and type level substitution, explicit heap, objects, etc., and found our approach to be feasible. We expect our 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.

This paper presents techniques that significantly reduce the state space of a model checker for checking type soundness. This paper thus makes contributions both in the area of checking soundness of type systems, and in the area of reducing the state space of a software model checker.

Acknowledgments

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

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

[5] 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.

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

[7] C. Boyapati, B. Liskov, L. Shrira, C. Moh, and S. Richman. Lazy modular upgrades in persistent object stores. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 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] C. Boyapati, A. Salcianu, W. Beebee, Jr., and M. Rinard. Ownership types for safe region-based memory management in Real-Time Java. In *Programming Language Design and Implementation (PLDI)*, June 2003.

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

[11] 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.

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

[13] 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.

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

[15] 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.

[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] 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.

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

[21] 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.

[22] N. Een and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *Theory and Applications*

of Satisfiability Testing (SAT), June 2005.

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

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

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

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

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

[28] 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.

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

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

[31] 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.

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

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

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

[35] 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.

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

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

[38] 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.

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

[40] 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.

[41] 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.

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

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

[44] 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.

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

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

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

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

[49] 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.

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

[51] 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.

[52] 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.

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

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

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

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