Modular Verification of Code with SAT

Greg Dennis Felix Sheng-Ho Chang Daniel Jackson
Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Laboratory
Cambridge, MA 02139, USA
{gdennis, fschang, dnj}@mit.edu

ABSTRACT

An approach is described for checking the methods of a class against a full specification. It shares with traditional model checking the idea of exhausting the entire space of executions within some finite bounds, and with traditional verification the idea of modular analysis, in which a method is analyzed, in isolation, for all possible calling contexts.

The analysis involves an automatic two-phase reduction: first, to an intermediate form in relational logic (using a new encoding described here), and second, to a boolean formula (using existing techniques), which is then handed to an off-the-shelf SAT solver.

A variety of implementations of the Java Collections Framework's List interface were checked against existing JML specifications. The analysis revealed bugs in the implementations, as well as errors in the specifications themselves.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*model checking, validation*; F.4.3 [**Mathematical Logic**]: Formal Languages—*Alloy*

General Terms

Verification, Reliability, Languages, Algorithms

Keywords

software model checking, SAT, formal methods, first-order logic, formal verification, Alloy

ISSTA'06, July 17–20, 2006, Portland, Maine, USA.

1. INTRODUCTION

Robust components are essential to the construction of dependable software systems. Although the early vision of a software component industry [25] has not been realized [22], there is at least widespread use of data structure libraries, such as the Java Collections Framework [27]. At the same time, much progress has been made in the field of program verification, especially in the use of theorem proving [14], static analysis [23, 35], and model checking [7, 8, 13, 15, 34, 12, 26, 16].

A major obstacle to verifying components against their specifications still remains, however. It seems that neither theorem proving nor static analysis can automatically check the kinds of rich properties that appear in the specifications of data structure libraries, so their use tends to be restricted to simpler (but nonetheless important) properties, for example that there are no null pointer dereferences or illegal array accesses.

Explicit state model checkers, such as Java PathFinder [34], can actually check rich properties since they perform an explicit search in which arbitrary runtime assertions can be executed, but their analysis is not modular. They can be used to check components in the context of a particular program, but they cannot be used to check a component in isolation against its specification, accounting for all possible contexts of use.

The research reported in this paper is part of a larger project whose aim is to develop a practical technique for modular code verification. Like traditional verification – and in contrast to most work on model checking – it analyzes a method in isolation, for all calling contexts. But like model checking, and unlike techniques based on theorem proving and static analysis, the analysis is exhaustive, but only with respect to some finite bounds. A successful "verification" may therefore fail to find a bug in a component, but because of the size of the search space, this should rarely occur. It seems reasonable to retain the term "verification" to distinguish this kind of analysis from testing and from analyses that do not handle full specifications.

The controversial thesis underlying the project is that code can be verified by a precise reduction to SAT (boolean satisfiability), with careful encodings, but without any abstraction. For checking partial properties, abstraction seems very desirable, but for properties that cover all details of behavior, abstraction is likely to lose crucial information.

The analysis involves an automatic two-phase reduction. First, the code is translated to an intermediate form in Alloy, a relational logic suitable for describing heap-manipulating

code [30, 18]. The translation scheme builds upon prior work on the Jalloy tool [19, 33, 32]. Like Jalloy, the tool bounds the analysis by the size of the heap and the number of loop unrollings explored.

In the second phase of the reduction, existing techniques [17] are used to translate the Alloy logic to a boolean formula, which is then handed to an off-the-shelf SAT solver. Unlike Jalloy, the reduction of the generated relational formula to SAT is not performed by the Alloy Analyzer [5], which was designed as a desktop CAD tool and not as a backend to other analysis tools, but by a new relational engine designed expressly for this kind of application [29]. The features of this engine are exploited by the new first phase translation.

These ideas were applied to a realistic and non-trivial problem: the verification of a variety of implementations of a list datatype. The list was chosen because of its ubiquity in object-oriented programs; because it is one of the simplest datatypes found in libraries (and thus a good starting point for research), and yet a building block for many other datatypes; and because, to our knowledge, despite all this, no other research project has (to our knowledge) succeeded in automatically verifying a practical list implementation.

Our analysis requires the user to write, or otherwise obtain, a specification against which the method will be checked. For this case study, we used a pre-existing specification of the Java List interface written in the Java Modeling Language (JML) [4]. Ten implementations of the JML specification were considered; five of these were variants into which defects had been seeded for an exercise on test coverage in an undergraduate software engineering class [1]. The remaining five were the standard implementation from the Java Collections Framework, two implementations from the GNU Trove library [2], and two implementations from the Apache Jakarta Commons-Collections library [3].

Once a specification is written, a new implementation can be checked against it automatically without additional work from the user, unless the data representation has changed. For each distinct data representation, we wrote a representation invariant and an abstraction function to relate that representation to the abstract values of the specification.

In summary, this paper describes an approach to checking object-oriented code against detailed specifications that is fully automatic, demanding from the user only a specification of the method to be checked, an indication of the size of the space to be searched, and for each distinct representation, an abstraction function and representation invariant.

The focus of the paper is the scheme by which the code is translated into relational logic for subsequent solving. The case study demonstrates the feasibility of the approach, and contributes some evidence to the small scope hypothesis [6] that most defects have small counterexamples.

2. RELATIONAL LOGIC

The logic used for both specification and representation of the code is a core subset of the Alloy modeling language [30]. The analysis solves problems written in the logic with the Kodkod relational engine [29].

2.1 Syntax and Engine

Figure 1 gives the grammar for the relational logic. A *problem* is a set of *relation declarations*, each which specifies the arity of a relation variable, and a *formula* in which the declared relations appear as free variables. The *formula* and *expr* syntactic productions define a standard relational logic with transitive closure, first order quantifiers, and logical connectives. A relation can have any finite arity. Sets are treated as unary relations (relations of arity 1), and scalars are singleton sets.

A *model* of a formula is a binding of the formula's relational variables to sets of *tuples* drawn from a universe of *atoms*. An engine (such as Kodkod) that searches for models of a formula in a finite universe is called a *model finder*.

The problem of assigning teachers to courses, for example, where each course must have exactly one teacher, and each teacher can teach at most one course, can be formulated as follows:

```
Teacher : 1, Course : 1, teach : 2
(all t : Teacher | one t.teach or no t.teach) and
(all c : Course | one teach.c)
```

This problem declares three relational variables: the sets Teacher and Course, representing the teachers and courses, respectively, and the binary relation teach, mapping teachers to courses. The formula constrains teach to be a bijective partial function.

In addition to the problem to be solved, Kodkod requires a universe of atoms to be specified, and an *upper bound* on each relation, consisting of the set of tuples that the relation may contain. A lower bound for each relation — a set of tuples it *must* contain — may also be given. The upper bounds determine the *scope* of the space being analyzed; the lower bounds comprise a partial solution or *partial instance*. The end user does not provide these directly; both are byproducts of the analysis, although the upper bounds are derived in part from the bounds indicated by the user.

Kodkod translates the logic problem, upper bounds, and partial instances into a boolean formula and invokes a SAT solver to find its satisfying solutions. Kodkod's support for partial instances is one of its key advantages over the Alloy Analyzer, and one which our tool exploits in its analysis. Because partial instances reflect fixed parts of the solution that do not need to be discovered, they enable Kodkod to reduce the sizes of the boolean formulas it generates.

2.2 Relational View of the Heap

The translation of object-oriented code into relational logic, described in detail in Section 4.3, is based on a *relational view of the heap* [20]. In this view, fields are binary, functional relations that map elements of their class to elements of their target type; and local variables and arguments are singleton sets.

In this view, field dereference becomes relational join. To illustrate, consider the field `f` represented by the functional relation F, and the variable `x` represented by the singleton set X. Because X is a singleton and F is a function, the join expression X.F yields a singleton set representing the value of the dereference `x.f`. An update to a field is encoded as a relational override. After the statement `x.f = y`, for example, the `f` field can be represented by the expression (F ++ (X −> Y)), meaning the relation F with tuples whose first element is X replaced by the tuple ⟨ X, Y ⟩.

Figure 1 Relational Logic Syntax

problem := relDecl* formula

relDecl := var : arity
varDecl := var : expr

arity := 1 | 2 | 3 | 4 | ...
var := identifier

formula :=	
expr in expr	subset
| expr = expr	equality
| expr != expr	inequality
| some expr	non-empty
| one expr	singleton
| no expr	empty
| not formula	negation
| formula and formula	conjunction
| formula or formula	disjunction
| all varDecl | formula	universal
| some varDecl | formula	existential

expr :=	
var	variable
| expr + expr	union
| expr & expr	intersection
| expr − expr	difference
| expr . expr	join
| expr −> expr	product
| expr ++ expr	override
| ∼ expr	transpose
| ^ expr	closure
| {varDecl | formula}	comprehension
| if formula then expr else expr	conditional

3. EXAMPLE

This section demonstrates the analysis from a user's perspective on a small example: checking a set of integers represented as a linked list against a specification in relational logic (Figure 2). The linked list is singly-linked, circular, and has a dummy header entry.

The analysis requires the user to provide:

- a specification of the method to be checked;
- a representation invariant on the concrete fields of the datatype; and
- a bound on the size of the heap, integer bit width, and number of loop iterations;

and, if the data representation of the code differs from the representation used in the specification,

- an abstraction function that relates the concrete fields to the abstract values of the specification.

These elements are shown in Figure 2 in a format concocted for the purposes of this example, based on the relational logic of Figure 1. [1]

The `specfield` declaration introduces a "specification field" `elems`, the abstract value of the datatype, which is a binary relation mapping instances of `LinkedIntSet` to integers. The concrete fields `header`, `next`, and `element` are likewise treated as binary relations, in accordance with the relational view of the heap. Following conventional object-oriented practice, references to `this` are omitted, and `elems` and `header` are short for `this.elems` and `this.header` when they appear in formulas.

The `abstraction` tag labels the abstraction function, which relates the concrete `header`, `next`, and `element` fields, represented as binary relations, to the specification field. The expression `(header.^ next - header)` denotes the set of all the entries in the list except the header. The join of this set with the `element` relation evaluates to the set of all the integers in the list.

The representation invariant, marked by the `invariant` tag, constrains the list to be circular by stating that the header entry is reachable from itself; and it constrains the list to not have duplicates by stating that every unequal pair of non-dummy entries has unequal elements.

[1]In the case study, the specifications were presented to the tool in JML, but the abstraction function and invariant were actually written directly in the logic.

Figure 2 Integer Set Implementation & Specification

```
00  class LinkedIntSet {
01
02  /*
03   * @specfield
04   *  elems : set int
05   *
06   * @abstraction
07   *  elems = (header.^next - header).element
08   *
09   * @invariant
10   *  (header in header.^next) and
11   *  (all e1, e2: header.^next - header |
12   *     e1 != e2 => e1.element != e2.element)
13   */
14
15    Entry header;
16
17    /*
18     * @ensures no elems'
19     * @modifies elems
20     */
21    void clear() {
22      this.header.next = this.header;
23    }
24
25    /*
26     * @check for 4 Entry, 4 int, 3 iteration
27     * @ensures (return = true) <=> (i in elems)
28     */
29    boolean contains(int i) {
30      Entry e = this.header.next;
31      while (e != this.header) {
32        if (e.element = i)
33          return true;
34        e = e.next;
35      }
36      return false;
37    }
38  }
39
40  class Entry {
41    Entry next;
42    int element;
43  }
```

In the specification of a method, the `ensures` clause labels the post-condition and the `modifies` clauses lists the fields that a method may modify. The formula `(no elems')` in the `clear` specification states that the set of integers in the list in the post-state is empty, and the `modifies` clause allows no other field besides `elems` to be modified. The specification of `contains` states that the return value is true if and only if the argument is a subset of the set of integers in the list, and the lack of a modifies clause means no field is modified.

The `check` tag indicates that `contains` is the method to be analyzed, and it specifies the scope of the analysis, limiting the heap to at most 4 instances of the `Entry` class, the integers to those available within a bit width of 4; and it bounds the number of iterations of a loop to 3.

The tool completes this analysis of the `contains` method in a few seconds and reports that no specification violations are found. This is not a proof that the method satisfies its specification on all inputs, only that it does so on all inputs within scope. The user's confidence can be increased by running the analysis again with a larger bounds.

The method in this case is correct, so increasing the bounds will not reveal bugs. But let's suppose that the developer accidentally swaps `true` and `false` in the `contains` method. Now, the tool finds a trace of the code that violates the specification. Though it currently lacks the ability to display a trace graphically, it can produce textual output like the following:

```
pre-state:
        this = S0
        i = 1
        header = <S0, E0>
        next = <E0, E1>, <E1, E2>, <E2, E0>
        element = <E0, 0>, <E1, 2>, <E2, 1>
        elems = <S0, 1>, <S0, 2>
Line 30 Entry e = this.header.next:
        e = E1
Line 31 while (e != this.header):
        true
Line 32 if (e.element = 1):
        false
Line 34 e = e.next:
        e = E2
Line 31 while (e != this.header):
        true
Line 32 if (e.element = 1):
        true
Line 33 return false:
        return = false
post-state:
        e = E2
        return = false
```

The beginning of the output displays the value of every parameter and field in the pre-state. In this execution, `contains` is called with an integer argument 1 and a receiver argument `S0`, whose abstract value is the set containing integers 1 and 2. Next, the output shows the lines of code executed, the values assigned to variables and fields at those lines, and the boolean result of evaluating the conditions on branches. Lastly, it shows the post-state values of the variables and fields that are modified during the execution. In the trace above, the method returns false, contrary to its specification.

4. APPROACH

4.1 Basics

The basic idea underlying the approach is as follows. From the code of a procedure, the tool automatically obtains a formula P(s, s') in relational logic that constrains the relationship between a pre-state s and a post-state s', and holds whenever an execution exists from s that terminates in s'. A second formula S(s, s') is obtained from a user-provided specification, and its negation is conjoined to the first, obtaining the formula

$$P(s, s') \text{ and not } S(s, s')$$

which is true exactly for those executions that are possible but which violate the specification. These counterexamples are models of the formula, obtained by translating the relational formula to a boolean formula, and applying a SAT solver to it.

The formula P is obtained by a translation that assumes that each loop is executed at most some small number of times. An additional approximation is introduced by a bound on the size of heap, which the user specifies as a limit on the number of instances of each type that may exist. Both of these approximations are underapproximations — they eliminate possible behaviors. Thus, any counterexample generated will be valid, either demonstrating a defect in the code or a flaw in the specification, but defects may be missed.

When checking the code of an abstract type such as `LinkedIntSet`, the (abstract) representation used in the specification will not be the same as the (concrete) representation used in the code. The standard technique for bridging the gap involves the user providing an abstraction function [11] that interprets each concrete value as an abstract one. In this logical setting, the abstraction is provided as a formula A(c, a) that relates a concrete and an abstract state.

Procedure specifications have an implied pre-condition that their datatype arguments are well-formed. This well-formedness property is captured by a representation invariant R(c). Thus, to check the procedure for conformance to the specification, the analysis searches for solutions to the following formula:

$$R(c) \text{ and } P(c, c') \text{ and } A(c, a) \text{ and } A(c', a') \text{ and not } S(a, a') \quad (1)$$

A solution, if one exists, witnesses a pair of concrete states whose corresponding abstract states do not satisfy the specification – a counterexample to the claim that the procedure satisfies the specification.

When checking multiple implementations against the same specification, the specification need only be written once, while a new abstraction function and representation invariant is needed for each unique representation. The bounds on the analysis is trivial to enter and change.

4.2 Preprocessing

Before translating a procedure into relational logic, our tool automatically preprocesses the procedure's code into a form that is more amenable to analysis. First, each loop in the procedure is unrolled into a nested if-statement, with an `assume` statement containing the negation of the loop condition appended to the end of the innermost if-statement. Unrolling the loop in the `contains` method (Figure 2) for two iterations yields the following:

```
if (e != this.header) {
  if (e.element = i)
    return true;
  e = e.next;
  if (e != this.header) {
    if (e.element = i)
      return true;
    e = e.next;
    assume e == this.header;
  }
}
```

After loop unrolling, our tool resolves dynamic dispatch by transforming each virtual call into a series of tests on the type of the receiver argument. Each test is followed by an invocation of the concrete method provided by that type. Currently, a simple dataflow analysis is used to determine the potential target methods of an invocation, though a more sophisticated analysis could be easily plugged in.

Next, the procedure is transformed into a control-flow graph in which all expressions are side-effect free. The syntax for the CFG, shown in Figure 3, should be thought of as describing a directed acyclic graph in which statements can be shared, rather than a tree. A procedure in this syntax has any number of parameter variables, a single "return" variable, and a body which is a statement. Note that there are no traditional "return" statements in the syntax; assignments to the return variable take their place.

Finally, all procedure calls are inlined. Recursion is unrolled in a manner analogous to loop unrolling.

4.3 From Code to Logic

After these preprocessing steps, the transformed procedure is encoded as a relational logic problem. A binary relation is declared for each field and a unary relation for each class and procedure parameter, along with the predefined unary relations nullRel, trueRel, and falseRel to represent the program constants `null`, `true`, and `false`.

In an analysis of the `contains` method in Figure 2, for example, the encoding would create the following relation declarations:

```
LinkedIntSet : 1, Entry : 1, int : 1,
header : 2, next : 2, element : 2,
this : 1, i : 1, nullRel : 1, trueRel : 1, falseRel : 1
```

The relation introduced for each class represents the *extent* of that class, the set of all instances of the class that can exist over the lifetime of the procedure. The declared relations for variables and fields, later referred to as "pre-relations", will represent values in the pre-state (when the procedure is called). The encoding will construct a "post-expression" for each variable and field, representing the value of that variable or field in the post-state (when the procedure terminates).

The encoding also builds a *termination condition*, a formula over the pre-relations and post-expressions that must be true for the procedure to terminate. Finally, the tool constructs Formula 1 (Section 4.1), where the pre-relations form the concrete pre-state c, the post-expressions form the concrete post-state c', and the termination condition becomes P. Together with the relation declarations, this formula comprises the logic problem that is delegated to Kodkod.

To construct the post-expressions and the termination condition, the encoding performs a symbolic execution [21] of the code. For each program point, the execution maintains 1) an environment mapping variables and fields to relational expressions for their values, and 2) a constraint on those expressions that must be true for the program to reach that point. Initially, the environment maps each field and variable to its pre-relation, and the constraint states that each field pre-relation is a function and that each parameter pre-relation is a singleton. For the `contains` method, the initial constraint would be:

```
(all x : LinkedIntSet | one x.header) and
(all x : Entry | one x.next) and
(all x : Entry | one x.element) and
(one this) and (one i)
```

The symbolic execution rules (Figure 4) describe how the final environment and constraint are obtained in terms of 3 functions: U (for "update"), C (for "constraint"), and X (for "expression"). When provided a statement and an initial environment, U yields a new environment, and C yields a formula to conjoin to the current constraint. When provided a (side-effect free) `JExpr` and an environment, X yields an relational expression for the value of that `JExpr` in that environment. The function *extent* maps classes to their extent relations.

Not all statement kinds are shown; calls are assumed to have been inlined (Section 4.2), and constructors are discussed informally below (Section 4.3.2). For any other statement S for which a rule for U or C is not given explicitly, $U[\![S, E]\!] = E$ and $C[\![S, E]\!] = \mathit{trueRel}$. The treatment of integers is also discussed informally below (Section 4.3.1).

The definition of X follows straightforwardly from the relational view of the heap. Note that in all these rules, the relational operators are *syntactic*. For example, in the rule for a dereference expression

$$X[\![e_1.e_2, E]\!] = (X[\![e_1, E]\!]).(X[\![e_2, E]\!])$$

the dot on the right-hand side constructs a new join expression from the results of the calls to X.

To illustrate U, consider the rule for a field update:

$$U[\![e_1.y = e_2, E]\!] = \\ E[y \mapsto (E(y) \ ++ \ (X[\![e_1, E]\!] \ -> \ X[\![e_2, E]\!]))]$$

This rule states that the expression for the field y after the update is a new override expression whose left operand is the current expression for y and whose right operand is a new cross product expression of the current expressions for e_1 and e_2. In other words, y is overridden with a mapping from e_1 to e_2.

The `clear` method in Figure 2 would trigger this field update rule. Assuming the initial environment binds each field and parameter to relations of the same name, the environment immediately after the statement would bind the `next` field to the relational expression

```
next ++ (this.header -> this.header)
```

The encoding of branches makes use of the conditional expressions in the logic. Consider the simple if-statement

```
if (c) then x = a; else x = b;
```

If the environment prior to this if-statement maps `a`, `b`, and `c` to expressions of the same name, then the environment after the if-statement would map `x` to the conditional expression (`if c then a else b`).

Figure 3 Procedure Syntax with Side-Effect Free Expressions

```
Procedure  ::= procid (Var*) returns Var { Stmt }

Stmt       ::= ElemStmt | BranchStmt | ElemStmt;Stmt
ElemStmt   ::= AssignStmt | UpdateStmt | CallStmt |
               NewStmt | AssumeStmt
BranchStmt ::= if JExpr then Stmt else Stmt
AssignStmt ::= Var = JExpr
UpdateStmt ::= JExpr.Field = JExpr
CallStmt   ::= [Var =] procid(JExpr*)
NewStmt    ::= Var = new Class
AssumeStmt ::= assume JExpr
```

```
JExpr           ::= Var | ConstantExpr | JExpr.Field |
                    BinaryExpr | UnaryExpr | InstanceOfExpr
ConstantExpr    ::= null | true | false | 0 | 1 | -1 | 2 | -2 | ...
BinaryExpr      ::= JExpr BinaryOp JExpr
UnaryExpr       ::= UnaryOp JExpr
InstanceOfExpr  ::= JExpr instanceof Class
Var,Field,Class ::= identifier

BinaryOp        ::= && | || | = | < | > | + | − | * | \
UnaryOp         ::= - | !
```

Figure 4 Semantics of the Symbolic Execution

$Env = (Var + Field) \rightarrow expr$

$U : Stmt \rightarrow Env \rightarrow Env$
$C : Stmt \rightarrow Env \rightarrow formula$
$X : JExpr \rightarrow Env \rightarrow expr$

$extent : Class \rightarrow Relation$
$nullRel,\ trueRel,\ falseRel : Relation$

$U[\![v = e, E]\!] = E[v \mapsto X[\![e, E]\!]]$

$U[\![e_1.y = e_2, E]\!] =$
$\quad E[y \mapsto (E(y)\ +\!+\ (X[\![e_1, E]\!] \text{ -> } X[\![e_2, E]\!]))]$

$U[\![\textit{if } e \textit{ then } S_1 \textit{ else } S_2, E]\!] = E' \textit{ s.t}$
$\quad \textit{let } E_1 = U[\![S_1, E]\!],\ E_2 = U[\![S_2, E]\!],\ f = e2f(X[\![e, E]\!])$
$\quad\quad \forall\ v : Var \cup Field \mid E'(v) = \textit{if } f \textit{ then } E_1(x) \textit{ else } E_2(x)$

$U[\![S_1; S_2, E]\!] = U[\![S_2, U[\![S_1, E]\!]]\!]$

$C[\![\textit{assume } e, E]\!] = e2f(X[\![e, E]\!])$

$C[\![\textit{if } e \textit{ then } S_1 \textit{ else } S_2, E]\!] = F' \textit{ s.t.}$
$\quad \textit{let } F_1 = C[\![S_1, E]\!],\ F_2 = C[\![S_2, E]\!],\ f = e2f(X[\![e, E]\!])$
$\quad\quad F' = (f \textit{ implies } F_1) \textit{ and } ((\textit{not } f) \textit{ implies } F_2)$

$C[\![S_1; S_2, E]\!] =$
$\quad C[\![S_1, E]\!] \textit{ and } C[\![S_2, U[\![S_1, E]\!]]\!]$

$X[\![v, E]\!] = E(v)$
$X[\![e_1.e_2, E]\!] = (X[\![e_1, E]\!]).(X[\![e_2, E]\!])$
$X[\![e_1 = e_2, E]\!] = f2e(X[\![e_1, E]\!] \textit{ in } X[\![e_2, E]\!])$
$X[\![e_1 \ \&\&\ e_2, E]\!] = f2e(e2f(X[\![e_1, E]\!]) \textit{ and } e2f(X[\![e_2, E]\!]))$
$X[\![e_1 \ ||\ e_2, E]\!] = f2e(e2f(X[\![e_1, E]\!]) \textit{ or } e2f(X[\![e_2, E]\!]))$
$X[\![!e, E]\!] = f2e(\neg e2f(X[\![e, E]\!]))$
$X[\![e \textit{ instanceof } c, E]\!] = f2e(X[\![e, E]\!] \textit{ in } extent(c))$
$X[\![null, E]\!] = nullRel$
$X[\![true, E]\!] = trueRel$
$X[\![false, E]\!] = falseRel$

$e2f(e) = e \textit{ in } trueRel$
$f2e(f) = \textit{if } f \textit{ then } trueRel \textit{ else } falseRel$

To illustrate the purpose of C consider applying the rule

$$C[\![\textit{assume } e]\!] = e2f(X[\![e, E]\!]),\quad \text{where } e2f(e) = e \textit{ in } trueRel$$

to the assume statement

```
assume x.f;
```

where the field `f` is of type boolean. The result is the constraint (x.f in trueRel). The helper functions *e2f* and *f2e* convert back and forth between boolean expressions and formulas. They are required because the procedure syntax, like the grammar for most programming languages, treats boolean formulas as expressions, while the logic does not.

4.3.1 Integers

To accommodate integer arithmetic, for which the relational logic provides no built-in support, several predefined relations are declared at the outset:

- a relation representing the set of all integers, the extent of the integer type;
- inc, a binary relation that totally orders the integers from the smallest to the greatest: for all integers i except the greatest integer, i.inc equals i + 1;
- add, a ternary relation mapping the two integer operands to the two's complement representation of their sum, so that the addition of i and j can be written j.(i.add);
- and a ternary relation for all other binary arithmetic operators (subtract, multiply, divide, bit shift, etc.)

Inequalities are expressed with inc: i > j is encoded as (i in j.^ inc), where j.^ inc is the transitive closure of the inc relation, and j.^ inc, therefore, is the set of all integers greater than j.

This approach to handling arithmetic in SAT, in contrast to a more conventional approach in which arithmetic operators are represented with boolean circuits over bit-strings, exploits Kodkod's partial instance facility. The tool completely pre-computes each of the integer relations for all the integers within the user-provided bit width. It then provides each set of tuples to Kodkod as both the upper and lower bounds for its respective relation. (Recall from Section 2.1 that an *upper bound* is a set of tuples that a relation *may* contain, and a *lowerbound* is a set of tuples that it *must* contain). In effect, the integer relations are treated as constants, whose possible values Kodkod does not need to explore. This approach is far easier to implement than the conventional approach, although its efficiency remains to be determined.

4.3.2 Constructors

To encode the creation of new objects, our tool declares at the outset a new binary relation for each class that totally orders all the instances in the extent of that class. The total order represents the sequence in which objects of the class will be constructed over the course of the procedure.

The symbolic execution environment, which maps each field and variable to an expression for its value, is augmented by an additional mapping from each class to an expression for the last instance created of that class. When a `New` statement is reached, an expression is constructed representing the new object, from the ordering and the environment.

This trick allows object creation to be represented deterministically as an expression; the inherent non-determinism in the semantics of constructors is reflected in the fact that the ordering is unconstrained. Because object references are uninterpreted, symmetry can be broken, and, using the same partial instance technique employed for integers, the ordering can actually be given a pre-computed constant value.

5. EXPERIMENTS

The analysis was applied to a variety of linked list implementations of the `java.util.List` interface, drawn from the Sun Collections Framework, the GNU Trove library, the Apache Jakarta Commons-Collections library, and some variants of the Sun implementation that had been seeded with bugs for an MIT software engineering class.

JML specifications for the following twelve `List` methods were mechanically translated into the relational logic: `add(int, Object)`, `add(Object)`, `clear`, `contains`, `get`, `indexOf`, `isEmpty`, `lastIndexOf`, `remove(int)`, `remove(Object)`, `set`, and `size`. For each unique representation, of which there were three, we wrote a representation invariant and an abstraction function to relate the concrete fields of the representation to the abstract values of the JML List specification.

We used our tool to check the implementations for conformance to the JML specifications. The analyses revealed bugs in the implementations, as well as errors in the specifications themselves.

5.1 JML Specifications

Figure 5 gives the JML specification of the `get` method in the Java `List` interface. The specification contains a single *model field* of type `JMLEqualsSequence` named `theList`. JML model fields represent the abstract values of their types; they are the terms in which JML method specifications are written. The `JMLEqualsSequence` is one of several "model collections" that come built-in with JML and that serve as the primitive building blocks upon which more complex JML specifications are written.

The JML specification for `get` is divided into two cases: the *normal_behavior* (when the index argument is within the bounds of the list) and the *exceptional_behavior* (when the index is out of bounds). In the normal case, the result is equal to the element at the specified index in the model field, and in the exceptional case the method signals an `IndexOutOfBoundsException`. Both cases include the "assignable \nothing" clause, meaning no model field is modified; therefore, as the specification redundantly indicates, the method is pure.

The specifications of the model collections themselves, including `JMLEqualsSequence`, consist only of informal English text, so an automatic translation of JML to relational logic must include built-in encodings of them. The tool formally represents a `JMLEqualsSequence` as a binary relation that maps each nonnegative integer to an object (or null) and contains no gaps between the integers that it maps.

Figure 5 JML List Specification

```
public interface List extends Collection {

    /*+@ public model instance non_null
       @                 JMLEqualsSequence theList;
       @*/

    /*+@ public normal_behavior
      @   requires 0 <= index && index < size();
      @   assignable \nothing;
      @   ensures \result == theList.get(index);
      @ also
      @ public exceptional_behavior
      @   requires !(0 <= index && index < size());
      @   assignable \nothing;
      @   signals_only IndexOutOfBoundsException;
      @*/
    /*@ pure @*/ Object get(int index);

}
```

We label this binary relation from integers to objects in the pre-state as theList and in the post-state as theList'. Our mechanical translation of JML to relational logic yields the following specification of the `get` method:

```
((lte(0, index) and lt(index, length(theList))) =>
    (result = index.theList and throw = null)) and
(!(lte(0, index) and lt(index, length(theList))) =>
    (throw in IndexOutOfBoundsException)) and
(theList = theList')
```

The index relation is the singleton set representing the value of the integer argument to the procedure. The result and throw relations are the sets representing the return value and the "thrown" value, respectively. If a procedure does not raise an exception, throw is `null` and result is a singleton; if an exception is raised, throw is a singleton and result is unconstrained. Finally, IndexOutOfBoundsException is the extent of — a set containing all the instances of — the `IndexOutOfBoundsException` class.

The lt and lte functions are built-in representations of the less-than and less-than-or-equal comparisons that package the inc relation discussed in Section 4.3.1, and the length function accepts a list represented as a binary relation from integers to objects and returns the length of that list. Since index is a singleton set containing an integer, and theList is a functional relation from integers to objects, the join index.theList evaluates to the object to which the index is mapped in theList.

5.2 Abstraction Functions and Rep Invariants

For each unique list representation, we wrote an abstraction function and a representation invariant. There were three such unique representations, and all three were circular and doubly-linked, and they each required a (non-null) dummy node at the front of the list. This typical representation is shown in the `LinkedList` class in Figure 6.

Figure 6 Typical Representation of Linked List

```
class LinkedList {
  Entry header;
}

class Entry {
  Entry next, prev;
  Object element;
}
```

The representation invariant of `LinkedList` says that it is circular, doubly-linked, and that the header points to a non-null node:

```
all e: this.header.^next + this.header | e.next.prev = e
```

A suitable abstraction function (with the relations representing **concrete fields** and *abstract values* shown in distinct fonts) might be:

```
let indices = theList.(Object + null)
let entries = header.^next - header
let pred = prev - (header.next -> header)
   all i: indices | some e: entries |
      (#(e.^pred)) = i and i.theList = e.element
```

which says that for any index of the abstract list, there is an entry in the concrete list with a matching element, and whose number of predecessors is equal to the index. Since the set cardinality operator # has not yet been implemented in Kodkod, the actual abstraction functions were a bit more involved, and took the form

```
let indices = theList.(Object + null)
let entries = header.^next
map: indices one -> one entries

all i: indices {
  let currEntry = i.map
  let nextIndex = i.inc
  let nextEntry = currEntry.next
  i.theList = currEntry.element
  some (nextIndex & indices) =>
    nextIndex -> nextEntry in next else
    nextEntry = header
}
```

using a bijection map to draw a correspondence between abstract and concrete elements. Both of these formulas make use of features of Alloy that are supported by Kodkod but omitted from the grammar of Figure 1, in particular the let shorthand, and the multiplicity constraint on map making it bijective.

5.3 Results

Our analyses of the implementations were conducted within a scope of 4 list entries, 4 objects, integers ranging from -8 to 7 (4 bits), and 3 loop unrollings. All experiments were run on a 2.2GHz Intel Pentium 4 machine with 1GB RAM and Ubuntu GNU/Linux 5.10. The complete timing results, shown in Table 1, include all the preprocessing steps.

5.3.1 Sun Java Collection Framework

Our first experiment analyzed the `java.util.LinkedList` implementation provided by Sun. We checked each of the 12 List methods against the JML specifications, finding no specification violations. As shown in Table 1, no analysis exceeded two minutes.

5.3.2 GNU Trove

GNU Trove is a library of collection implementations designed to yield better performance than the Java Collections Framework in special situations. The library includes a class called `TLinkedList`, a linked list implementation that accepts elements implementing an interface that provides four methods: `getNext`, `getPrevious`, `setNext`, and `setPrevious`. `TLinkedList` uses the elements added to it as the actual nodes in the linked list. This is intended to avoid the extra cost of constructing separate node instances. One disadvantage of this design is that it disallows duplicate list entries.

We checked two versions of the `TLinkedList` class for conformance to the JML specifications. The first version we checked is distributed with Trove 1.1b5, the most recent version of the library. The second version is from a version of Trove released four years ago, version 0.1.2, which contained a bug in an inner iterator class in `TLinkedList`, according to the project's CVS logs.

Upon checking the most recent version of `TLinkedList`, we found two of its methods to violate the JML specifications, `remove(Object)` and `add(int,Object)`. When the argument to `remove(Object)` is not already contained in the list, the method can behave incorrectly. Upon inspection of the Trove API, we found this behavior to be deliberate; its specification for `remove(Object)` includes a precondition that the element must be contained in the list. We temporarily amended our JML specification to include this precondition and found no further violation of this specification.

The specification violation in `add(int,Object)`, however, was not deliberate and constitutes a genuine bug in the implementation that was apparently unknown to the developers. The method contains a subtle off-by-one error when inserting into the middle of the list.

Checking the older version of `TLinkedList`, we found half of its methods to violate their specifications. The CVS logs for the project only mentioned a bug in the inner iterator's `remove` method.

5.3.3 Jakarta Commons-Collections

Commons-Collections is a library offered by the Apache Jakarta project that contains a number of collection implementations not found in the standard Sun library. It includes two linked list implementations, both of which we checked. The first, `AbstractLinkedList`, is a standard linked list that provides the same functionality as the Sun implementation, except that it is written with finer-grained procedural abstraction, giving potential subclasses more flexible implementation support. Although it is an abstract class, it contains no abstract methods, so we were able to check it directly for its conformance to the JML specifications.

The second class analyzed was `NodeCachingLinkedList`. Like the `TLinkedList` class in GNU Trove, this Apache implementation attempts to mitigate the cost of constructing new nodes on each addition to the list. To do so, it maintains a separate linked list of nodes that have already been

Table 1 Duration of Method Analyses (seconds)

	add(i,o)	add(o)	clear	contains	get	indexOf	isEmpty	lastIndexOf	remove(i)	remove(o)	set	size
Sun	23.5	16.0	12.6	84.8	16.6	17.2	15.1	62.6	19.8	77.5	18.0	18.7
Trove1.1b5	***18.3***	15.0	12.6	14.6	16.9	20.0	15.6	18.3	22.2	***11.1****	24.0	13.0
Trove0.1.2	***14.2***	13.8	12.8	15.3	***13.0***	19.4	15.6	***15.0***	***13.3***	***20.4***	***13.9***	12.9
JakartaAbstract	20.8	20.2	13.0	64.5	16.4	25.2	20.9	94.9	37.4	34.2	20.9	29.4
JakartaCaching	25.3	24.1	14.9	119.3	23.8	22.9	22.5	84.8	28.5	100.2	24.8	19.1
MITSeeded	***10.1***					***10.7***, ***10.1***	***10.4***		***11.0***			

Displayed above are the times (in seconds) to check each method against its specification. Each analysis was bounded by 4 list entries, 4 objects, integers ranging from -8 to 7 (4 bits), and 3 loop unrollings. Specification violations are displayed in ***this font***, and the * indicates the violation was intentional and documented by the developer. In the seeded implementations only the seeded methods were checked, and those times are shown collectively in the last row. Two seeded implementations had bugs in `indexOf`.

constructed and reuses nodes from this cache whenever possible. Nodes are added to the cache upon element removal, but the cache is constrained to not exceed a preset maximum size. We found no specification violations in either implementation.

5.3.4 MIT Seeded Implementations

As an exercise on test coverage, students in an MIT undergraduate course in software engineering were asked to write comprehensive test suites of the Java `List` interface. To measure the coverage of each student's suite, the suites were executed on a series of mutant versions of the standard Java list. Each of these mutant versions contained a single `LinkedList` field to which all but one of their methods directly delegated; the remaining method was seeded with a bug surrounding the delegation.

Five of these mutants contained bugs in one of the 12 methods considered in this experiment. We used our tool to check the mutant methods and successfully detected the bug in each one. We did not check the remaining methods, because they directly delegated to the Sun implementation, which we had already determined did not violate the specifications.

5.4 Scope Effects

We ran further analyses to determine the smallest scope needed to detect each of the specification violations. No violation required more than a single loop unrolling to be revealed, and all but one violation was detected when linked lists were limited to a length of 2 and integers to a bit width of 3. The remaining violation – the apparently unknown bug in the latest version of the Trove library – required 3 list entries and 4-bit integers for its detection. These results contribute evidence to the small scope hypothesis [6].

To evaluate the scalability of the analysis, we re-analyzed the `add(int, Object)` method in the Sun implementation for progressively larger scopes. When bounded by 5 list entries, 5 objects, 4 loop unrollings, and 4 bits to an integer, the analysis completes in about 2 minutes. When increased to 6 list entries, 6 objects, and 5 loop unrollings, it takes about 20 minutes. If the bounds are increased to 7 entries and 7 objects and integers are increased to 5 bits, the analysis continues for one hour before timing out. Note, however, that at the time of this experiment, Kodkod supported only a Java-based SAT solver that was not the first in its class. Since then, both Kodkod and the Java SAT solver have undergone significant performance improvements, and Kodkod now supports even faster solvers written in C and C++.

5.5 Specification Errors

In the process of checking these implementations against the published JML specifications, our tool revealed two errors in the specifications themselves. These errors were corrected during the course of the analysis, and the data shown in Table 1 is for the corrected specifications.

The first error, found in the specification of the `add(int, Object)` method, was discovered when our tool reported a specification violation in the Sun implementation. The JML specification states that the method adds the specified element at the specified index in the list when the following condition is true:

```
requires 0 <= index && index < this.size();
```

— and that it throws an exception otherwise. However, the Sun specification states that the element should be added even when the index is *equal* to the size, in which case it should be added to the very end of the list.

A second error was found in the `indexOf` method when the tool failed to find a bug in a seeded implementation. The JML specification says, correctly, that if the method does not return -1, then the specified element must be in the list. However, it omits the necessary inverse: if the method *does* return -1, the element must *not* appear in the list.

6. RELATED WORK

6.1 Modular vs. Whole-Program Analysis

An analysis is modular if it allows components to be considered one at a time, in isolation. The result of analyzing a component is a summary that is then used as a surrogate for the component itself in an analysis of the component's clients. Some modular analyses are capable of generating summaries automatically; others require that the user provide them as specifications, which then overapproximate the component's behavior (giving some decoupling that makes the client less sensitive to changes in the used component).

For verifying a component such as a library class, a modular analysis is required. Such an analysis, like the one described in this paper, takes a method of the class and accounts for all possible invocations of the method. A whole-program analysis, in contrast, could only be applied if the class were used in a complete program, and would then only account for the usage of the class in that particular context.

Model checkers perform whole-program analyses, and are therefore not ideally suited to the verification of components. Because they admit non-deterministic constructs, however,

it is possible to write a driver that simulates a variety of potential invocations of the library methods. But the task of writing such a driver would pose a significant burden.

One might think, for example, that a driver to test an implementation of a set datatype might simply use the *add* method to generate all sets up to a certain size. However, such an approach would not provide good coverage, for it would cover only the *abstract* pre-states of a method. The approach described in this paper, in contrast, will consider all *concrete* representations, and to generate these the driver may need not only to apply additions in different orders, but also to apply the *remove* method.

Furthermore, if the bound on the analysis is large and/or the datatype has a weak invariant (e.g. a graph), the number of well-formed pre-states could be huge, possibly on the order of millions. An explicit model checker would likely be incapable of generating a state space of this size, and even a symbolic model checker may choke if it must handle long sequences of constructor calls to generate the states.

Alternatively, one could write a driver which constructs pre-states, not through API calls, but by manipulating the representation of the datatype directly. The test-case generators TestEra [24] and Korat [9] follow this approach. Given a representation invariant on a datatype, they search for an exhaustive set of non-isomorphic test inputs — within a finite bound — that satisfy the invariant, while employing aggressive pruning techniques to narrow the search space. Pruning is critical to the tools' feasibility. When exploring linked lists up to size four, for instance, there are billions of possible configurations, only a fraction of which are well-formed, and only a handful of those — less than 50 — could constitute an equivalent non-isomorphic subset.

For the case study presented in this paper, Korat and TestEra would probably have sufficed, and achieved similar coverage to our tool. For analyzing a method that takes two lists rather than one, however, these approaches would likely not achieve the coverage of our SAT-based approach, since the number of possible pre-states would grow too large to make explicit enumeration and execution feasible.

6.2 Related Approaches

A number of code analysis tools are capable of checking programs against JML specifications [10]. Some of these tools perform dynamic analyses, such as checking runtime assertions or unit testing; but being dynamic, they cannot attain an exhaustive level of coverage. The static analyses that handle JML can be placed on a spectrum, from those that are mostly automated but handle a small subset of JML, such as ESC/Java [14], to those that require more extensive user guidance but support nearly all of JML, such as LOOP [31]. None of the available tools seems to achieve the combination of exhaustive coverage, support for the entirety of JML, and full automation possible with our approach.

TVLA [23], a dataflow analysis tool, has been used to successfully verify programs that operate on data structures such as singly- and doubly-linked lists. Given an input program, it can automatically generate a conservative abstract program state for each program point. While the analysis is complete, it is not modular, and it is not easily extended to full structural specifications. Saturn [35], another static analysis, uses a SAT-solver as its underlying engine like our tool, but it is designed for a more limited specification language.

Model checkers such as Java PathFinder [34], SLAM [7], BLAST [15], Magic [12], SMV [26], SPIN [16], BMC [8], and NuSMV [13] have been successful in verifying control properties of systems and programs, but they tend not to be suited for the kind of data structure properties expressible in JML. As explained in Section 6.1, they are also not conducive to modular analysis.

7. CONCLUSIONS

The results of the experiments are encouraging. The approach appears to be viable, although much work remains to be done. The major obstacle to extending the analysis to all methods of the List interface is the handling of generic collections in methods such as `addAll`. In future work, we plan to extend the tool to use abstract specifications for methods called on generic objects, rather than attempting to infer behavior from the code of subclasses. We could also combine the analysis with techniques for inferring specifications of called procedures, such as Taghdiri's iterative refinement [28], to avoid the complete inlining of procedure calls.

The translation from JML to Alloy was straightforward; the only problem being the formalization of the JML model classes, which are defined only informally in JML itself. The writing of the abstraction functions and representation invariants was challenging, but once a style had been developed, it took little time to create them for each new representation.

All of the seeded defects were revealed. A previously unknown bug was discovered in the latest version of the GNU Trove library, as well as a deliberate specification violation, and several bugs were detected in an earlier version of the library. The analysis scaled adequately for the task at hand, in all cases terminating within a few minutes.

While increasing the scope to include lists of length 6 caused the analysis to time out, large scopes may not be required to detect most bugs in practice . In these experiments, no violation required more than a single loop unrolling nor lists longer than length 3 for their detection, providing further evidence to the small scope hypothesis that most defects have small counterexamples.

The experiments also revealed errors in the JML specifications themselves, suggesting that this is the first successful attempt to check any list implementation against full JML specifications.

8. REFERENCES

[1] 6.170 Laboratory in Software Engineering, Fall 2001. http://www.ocw.mit.edu/OcwWeb/Electrical-Engineering-and-Computer-Science/6-170Fall-2005/CourseHome/.

[2] GNU Trove: High performance collections for Java. http://trove4j.sourceforge.net/.

[3] Jakarta Commons-Collections. http://jakarta.apache.org/commons/collections/.

[4] JML Specifications for the Java Collections Framework. http://www.cs.iastate.edu/ leavens/JML-release/javadocs/java/util/Collection.html.

[5] The Alloy Analyzer. http://alloy.mit.edu/.

[6] A. Andoni, D. Daniliuc, S. Khurshid, and D. Marinov. Evaluating the "Small Scope Hypothesis". Technical Report MIT-LCS-TR-921, MIT CSAIL, 2003.

[7] T. Ball and S. K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In *POPL '02: Proceedings of the 29th ACM Symposium on the Principles of Programming Languages*, New York, NY, USA, 2002. ACM Press.

[8] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99), Amsterdam, The Netherlands*, 1999.

[9] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated Testing Based on Java Predicates. In *ISSTA '02: Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, July 2002.

[10] L. Burdy, Y. Cheon, D. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *Software Tools for Technology Transfer*, 7(3):212–232, 2005.

[11] C.A.R. Hoare. Proofs of Correctness of Data Representations. *Acta Informatica*, 1(4), 1972.

[12] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular Verification of Software Components in C. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 385–395, Washington, DC, USA, 2003.

[13] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A New Symbolic Model Verifier. In *11th International Conference on Computer Aided Verification (CAV'99), Trento, Italy*, pages 495–499, July, 2003.

[14] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'2002)*, pages 234–245, 2002.

[15] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 58–70, New York, NY, USA, 2002. ACM Press.

[16] G. J. Holzmann. The Model Checker SPIN. In *IEEE Trans. Softw. Eng.*, volume 23, 1997.

[17] D. Jackson. Automating First-Order Relational Logic. In *SIGSOFT '00/FSE-8: Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 130–139, New York, NY, USA, 2000. ACM Press.

[18] D. Jackson, I. Shlyakhter, and M. Sridharan. A Micromodularity Mechanism. In *Proceedings of the ACM SIGSOFT Conference on the Foundations of Software Engineering / European Software Engineering Conference (FSE / ESEC '01)*, 2001.

[19] D. Jackson and M. Vaziri. Finding Bugs with a Constraint Solver. In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2000.

[20] D. Jackson. Object Models as Heap Invariants. In *Essays on Programming Methodology*, edited by Carroll Morgan and Annabelle McIver. Springer Verlag, 2000.

[21] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[22] B. W. Lampson. Software Components: Only the Giants Survive. In K. S. J. Andrew Herbert, editor, *Computer Systems: Papers for Roger Needham*, Lecture Notes in Computer Science, pages 59–65. Springer-Verlag Berlin, 2004.

[23] T. Lev-Ami and M. Sagiv. TVLA: A System for Implementing Static Analyses. In *SAS 2000: Static Analysis, 7th International Symposium, Santa Barbara, CA, USA, volume 1824 of Lecture Notes in Computer Science*, pages 280–302, 2000.

[24] D. Marinov and S. Khurshid. TestEra: A Novel Framework for Automated Testing of Java Programs. In *ASE '2001: 16th IEEE International Conference on Automated Software Engineering*, pages 22–31, 2001.

[25] M. D. McIlroy. Mass-Produced Software Components. In J. M. Buxton, P. Naur, and B. Randell, editors, *Software Engineering Concepts and Techniques (1968 NATO Conference of Software Engineering)*, pages 88–98. NATO Science Committee, Oct 1968.

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

[27] Sun Microsystems. Java Collections Framework. http://java.sun.com/j2se/1.5.0/docs/guide/collections/.

[28] M. Taghdiri. Inferring Specifications to Detect Errors in Code. In *ASE '2004: 19th IEEE International Conference on Automated Software Engineering, Linz, Austria*, pages 144–153, 2004.

[29] E. Torlak and D. Jackson. The Design of a Relational Engine. Submitted for publication, 2006.

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

[31] J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. *Lecture Notes in Computer Science*, 2031:299+, 2001.

[32] M. Vaziri. *Finding Bugs in Software with a Constraint Solver*. PhD thesis, Massachusetts Institute of Technology, MA, USA, Feb. 2004.

[33] M. Vaziri and D. Jackson. Checking Heap-Manipulating Procedures with a Constraint Solver. In *9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003)*, 2003.

[34] W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In *ASE '2000: 15th IEEE International Conference on Automated Software Engineering*, pages 3–11, 2000.

[35] Y. Xie and A. Aiken. Saturn: A SAT-Based Tool for Bug Detection. In *17th International Conference on Computer Aided Verification (CAV 2005), Edinburgh, Scotland, UK*, 2005.