Finding Bugs Efficiently with a SAT Solver

Julian Dolby
IBM T.J. Watson Research Center
P. O. Box 704
Yorktown Heights, NY 10598
dolby@us.ibm.com

Mandana Vaziri
IBM T.J. Watson Research Center
P. O. Box 704
Yorktown Heights, NY 10598
mvaziri@us.ibm.com

Frank Tip
IBM T.J. Watson Research Center
P. O. Box 704
Yorktown Heights, NY 10598
ftip@us.ibm.com

ABSTRACT

We present an approach for checking code against rich specifications, based on existing work that consists of encoding the program in a relational logic and using a constraint solver to find specification violations. We improve the efficiency of this approach with a new encoding of the program that effectively *slices* it at the logical level with respect to the specification. We also present new encodings for integer values and arrays, enabling the verification of realistic fragments of code that manipulate both. Our technique can handle integers of much larger ranges than previously possible, and permits large sparse arrays to be handled efficiently.

We present a soundness proof for our slicing algorithm and a general condition under which relational formulae may be sliced. We implemented our technique and evaluated it by checking data structure invariants of several classes taken from the Java Collections Framework. We also checked for violations of Java's equality contract in a variety of open-source programs, and found several bugs.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Model Checking; F.3.1 [**Logics And Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms

Theory, Verification

Keywords

Model Checking, SAT Solving, Specification, Slicing

1. INTRODUCTION

Researchers have developed a wide range of automated techniques to find bugs in code, including testing and static analysis. Testing approaches exercise a subset of all possible program behaviors, and have the advantage of generating concrete witnesses for bugs. However, testing approaches may miss problems due to incomplete coverage. Static analysis techniques over-approximate all program behaviors, and is capable of proving the absence of an error. However, static analyses may generate spurious error reports due to imprecision.

A compromise between testing and static analysis is *systematic under-approximation*, which analyzes a finite space of program behaviors *exhaustively* with respect to a target property. It uses a formal model of a methodically-chosen finite space of program behaviors. There are many approaches to systematic under-approximation [8, 4, 23, 21, 35, 20, 31, 9]. These include software model checking, as well verification techniques based on relational logic [18] and constraint solving.

In this paper, we focus on checking expressive properties of heap-manipulating object-oriented code, using relational logic and constraint solving. We build on work applying the *small scope hypothesis* [18] to heap structures as a basis for systematic under-approximation. The small scope hypothesis holds that a small heap—just a few objects per type—provides effective coverage for testing heap-manipulating code. For instance, for a procedure that sorts a linked list, it likely suffices to test lists of small sizes and further tests become redundant.

Existing relational approaches [20, 31, 26, 9] encode the program in a first-order relational formula [18] representing all executions restricted to small finite loop counts. This is turned into propositional logic, imposing a bound on the number of objects per type. A user further provides a specification in first-order logic. A satisfying assignment for the free variables in the conjunction of the translated code and the negation of the specification indicates a violation of the specification. Modern SAT solvers can search for such counterexamples effectively in large formulae.

These approaches benefit from modularity over traditional software model checkers, i.e., they allow components to be analyzed in isolation. Traditional model checkers must enumerate initial contexts from which code is exhaustively tested, but relational techniques can search all possible contexts, and thus can uncover subtle unexpected bugs.

Previous relational approaches have two shortcomings. First, the code and specification are translated separately, so the encoding cannot benefit if the specification concerns only small portions of the code. Second, integers are treated as objects, representing every integer explicitly. Thus, the range of integers for analysis is severely limited and large numbers—constants or array indices—are prohibitively ex-

ESEC/FSE'07, September 3–7, 2007, Cavtat near Dubrovnik, Croatia.

pensive to analyze. Such numbers are needed for many properties of heap-manipulating code, such as e.g., those involving hashtable implementations that need large hash codes and arrays of buckets.

We address these shortcomings as follows. First, our translation from code to first-order relational logic performs a type of *slicing* at the logical level, based on the variables and instance fields present in the specification, and only relevant portions of the program are encoded. Second, we encode integers implicitly but precisely, as sums of powers of two. This allows us to represent integers in much larger ranges than previously possible. Our tool can handle 16-24 bit integers, a considerable improvement over previous approaches which handled 4 bits. Third, we encode arrays so that they can be sparsely indexed over a large size.

We built *Miniatur*, an implementation of our technique, and used it on two sets of benchmarks. The first consists of checking invariants of the Java collections classes, including `HashMap` and `TreeSet`, which demonstrates our ability to handle larger integers and arrays. Our tool handles the library code with very little change. The second involves checking the Java equality contracts for a variety of open-source Java programs, confirming many violations, which we had identified in previous work [32].

The contributions of this paper are the following:

- A translation of code into first-order relational logic that performs slicing with respect to the specification to be checked.
- A proof of soundness for our slicing algorithm, and a general condition under which a relational formula may be sliced.
- An encoding for integers based on sums of powers of two, enabling checking realistic numerical fragments of code; and an encoding of arrays with efficient support for sparse arrays.
- An implementation based on the WALA open-source framework [34] and the KodKod model finder [28], and experiments on a variety of benchmarks, including open-source programs, that demonstrate much improved scalability and uncover several bugs.

2. MOTIVATION

In this section, we will discuss some concrete examples to illustrate the capabilities of our technique. Miniatur takes as input a procedure and parameters used to finitize code including: number of loop unwindings, integer bit-width, and number of objects in the initial heap. The user may indicate specifications via Java assertions, or special assertions containing relational formulae. If there exists an assertion violation, Miniatur's output is an initial heap and parameters. An execution starting from these is a counterexample for that assertion.

Java gives each object a unique *identity*, but also allows the programmer to customize identity by overriding the `equals()` and `hashCode()` methods following a contract [10] that requires, among other things, that the equality relation be reflexive, transitive, and symmetric and that equal objects have identical hash-codes. However, the contract is unenforced and also error-prone [32]. The Java Collections API require that the types of objects stored in collections

```
public class Tester {
  public static void equalsTester(Object a, Object b) {
    if (a.equals(b)) assert a.hashCode()==b.hashCode();
  }
}
public class Point2D {
  public Point2D(int x, int y){
    this.x = x; this.y = y;
  }
  public boolean equals(Object o){
    if (o instanceof Point2D){
      Point2D other = (Point2D)o;
      return (this.x == other.x && this.y == other.y);
    }
    return false;
  }
  public int hashCode(){
    return 64*x + y;
  }
  private int x;
  private int y;
}
class Point3D extends Point2D {
  public Point3D(int x, int y, int z){
    super(x, y); this.z = z;
  }
  public boolean equals(Object o){
    if (o instanceof Point3D){
      Point3D other = (Point3D)o;
      return (this.z == other.z && super.equals(other));
    }
    return false;
  }
  public int hashCode(){
    return 256*z + super.hashCode();
  }
  private int z;
}
```

(a) Contract Example Program

```
// takes a bag and makes it into a set
Point2D[] bagToSet(Point2D[] values) {
  Collection result = new HashSet(16);
  for (int i = 0; i < values.length; i++)
    if (!result.contains(values[i]))
      result.add(values[i]);
  return (Point2D[]) result.toArray(new Point2D[result.size()]);
}
protected void testHarness(Point2D[] valuesBag) {
  Point2D[] valuesSet = bagToSet(valuesBag);
  for (int i = 0; i < valuesSet.length; i++)
    for (int j = 0; j < valuesSet.length; j++)
      if (i != j)
        assert !valuesSet[i].equals(valuesSet[j]);
}
```

(b) Collection Example Program

Counterexample for (a)

```
Parameter a
  [[Point2D_3]]
  field x = 11264
  field y = 12

Parameter b
  [[Point3D_3]]
  field x = 11264
  field y = 12
  field z = 16
```

Counterexample for (b)

```
Parameter values
  [[ArrayOfPoint2D_3]]
  index 0:
    [[Point2D_2]]
    field x = 2
    field y = 6
  index 1:
    [[Point3D_1]]
    field x = 2
    field y = 6
    field z = 61695
```

(c) Counterexamples

Figure 1: Examples

obey the equality contract and exhibit surprising behavior otherwise.

Our techniques can be used to check contracts, including Java's equality contract, and produce concrete counterexamples. Figure 1(a) shows a small hierarchy of classes: `Point2D` and its subclass `Point3D`. Though the `equals()` and `hashCode()` methods look reasonable, they do not satisfy the part of the contract which requires that equal objects have the same hash-code. We use Miniatur on the harness `equalsTester()` with 16 integer bits, and 4 objects, to obtain the counterexample shown in the left of Figure 1(c), in 23 seconds. The output shows the values of each parameter, where `[[Point2D_3]]` indicates an object of type `Point2D`. One may verify that `new Point2D(11264, 12)` and `new Point3D(11264, 12, 16)` produce two objects that are equal but have different hash codes. Large integer values are crucial for finding this violation which can only be exposed by having at least 8 bits for integers. Generating large numbers is vital to any good hash function. Miniatur's support for large integers is a key contribution of our technique.

As another example, consider Figure 1(b), in which `BagToSetExample.bagToSet()` takes an array `values` of `Point2D` objects and returns an array that is a set containing distinct elements of `values`, that is no two elements are equal (via `equals()`). The `contains()`, `add()` and `toArray()` methods of a `HashSet` are used to create an array containing a set of elements with no duplicates. A method `testHarness()` is used to check the desired output.

Analyzing this code with Miniatur produces the counterexample shown on the right of Figure 1(c). `ArrayOfPoint2D_3` is an array object containing two `Point2D` objects at indices 0 and 1. One may verify that invoking `bagToSet()` on an array containing these elements will return an array containing the exact same elements. It is unexpected that this execution is a counterexample to the assertion, since at first glance the output contains different elements. However, the assertion is indeed violated since:
`[[Point2D_2]].equals([[Point3D_1]])` is true. This example touches on another problem in this code: the `equals()` method in the `Point2D` class hierarchy is not symmetric, leading to subtle unexpected behavior.

This example illustrates that we can precisely analyze the manipulation of heap data structures in the Java Collections Framework. In particular, it highlights our ability to reason about arithmetic and bitwise operations performed in `java.util.HashSet` to distribute its elements over an array of hash buckets. Furthermore, the `HashSet` constructor call specifies the initial size of the array to be 16; even though only a few objects need be inserted into it, the analysis must be able to represent the fact that the array is that large. Our mechanism for encoding arrays enables this. This example was performed with 16 bits for integers, 4 atoms for initial heap objects, and took 2 minutes.

3. APPROACH

Our approach is similar to the previous approaches based on relational logic and constraint solving [20, 31, 9] We encode the behavior of a procedure in a logical formula that is satisfiable if there exists an execution that violates the specification. We then use a model finder [28] (based on a SAT solver) to find a solution.

$$
\begin{aligned}
\mathit{Form} ::= {} & \mathit{Expr}\ \{=,\neq,\subset,\not\subset\}\ \mathit{Expr} \mid \\
& \mathit{Form}\ \{\wedge,\vee,\Rightarrow\}\ \mathit{Form} \mid \neg\ \mathit{Form} \mid \\
& \{\forall,\exists\}\ x : \mathit{Expr}\ .\ \mathit{Form} \mid \{no,one\}\ \mathit{Expr} \mid \\
& \mathit{Bitset}\ \{\texttt{=},\texttt{!=},<,>,\leq,\geq\}\ \mathit{Bitset}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Expr} ::= {} & r \mid \mathit{Expr}\ \{\texttt{.},\texttt{->},\texttt{++},\texttt{+},\texttt{-}\}\ \mathit{Expr} \mid \{x{:}\mathit{Expr} \mid \mathit{Form}\} \\
& \mid \texttt{if}\ \mathit{Expr}\ \texttt{then}\ \mathit{Expr}\ \texttt{else}\ \mathit{Expr} \mid \mathit{Expr}^* \mid |\mathit{Expr}|
\end{aligned}
$$

$$
\mathit{Bitset} ::= v \mid \mathit{Bitset}\ \{\texttt{+},\texttt{-},*,/,\wedge_b,\vee_b\}\ \mathit{Bitset} \mid \mathit{sum}\ \mathit{Expr}
$$

Figure 2: Relational logic

```
if (x.next == z)
  x.next = null;
if (y.next != z)
  y = y.next;
assert x.next != z;
```

(a) Original Code

```
1: if (x0.next0 == z)
2:   U(next1,next0,x0,null);
3: next2 = φ(next0,next1);
4: if (y0.next2 != z)
5:   y1 = y0.next2;
6: y2 = φ(y0, y1);
7: assert x0.next2 != z;
```

(b) SSA Converted

Figure 3: Translation Example

3.1 Background

3.1.1 Relational Logic

We use a subset of Alloy [18, 19], a relational first-order logic. Its grammar is given in Figure 2. The basic concept in this logic is a *relation*: a set of tuples of equal size, composed of *atoms* from a given universe. The *arity* of a relation is the number of elements in each of its tuples, and its *bound* is the domain from which its tuples take value. For the purposes of analysis we will consider limited bounds later on.

The relational logic allows operations on relations, such as join `.`, product `->`, update `++`, set union `+`, set difference `-`, and transitive reflexive closure `*`, which form expressions. Union, difference, and closure have normal meanings. A join of two tuples $(a_0, \cdots, a_n).(a_n, \cdots, a_m)$ is $(a_0, \cdots, a_{n-1}, a_{n+1}, \cdots, a_m)$, and join of two relations is a pairwise join of their elements. A product of two tuples $(a_0, \cdots, a_n)$ `->` $(a_m, \cdots, a_l)$ is $(a_0, \cdots, a_n, a_m, \cdots, a_l)$, and the product of two relations is the pairwise product of their elements. The update operator allows changing a relation at a particular value. The expression r `++` $\{(a,\ b_1,\ \cdots,\ b_n)\}$ results in a relation where all tuples of the form $(a, \cdots)$ have been removed and replaced with $(a, b_1, \cdots, b_n)$. The formula $e_1 \subset e_2$ means that all tuples of expression $e1$ are included in e_2. The formulae *no Expr* and *one Expr* are true for empty and singleton relations respectively.

The relational logic also manipulates bitsets. These may be compared in a formula. The usual arithmetic and bitwise operations ($\wedge_b$,$\vee_b$) are available on them. Atoms may denote integer values, and *sum Expr* returns the sum of the integers in *Expr* as a bitset. The expression $|Expr|$ gives the cardinality of an expression as a bitset, i.e. the number of tuples in it.

3.1.2 SSA Form and the CDG

Our encoding exploits *Static Single Assignment* (SSA) Form [7] and Control Dependence Graph (CDG) [7] to efficiently encode data- and control-dependence information.

SSA form gives every value one definition and expresses merges with ϕ-nodes. We apply SSA form to fields as well as variables. The CDG has statements as nodes and edges from conditional statements to the statements that are directly control-dependent [7] on them. Each edge is labeled with the condition that causes the execution of its target statement. The CDG also has a root node.

Consider the example of Figure 3. Note that the SSA form has new names for every new definition of every variable and every field. A field update $e_1.f = e_2$ is rewritten as $\mathcal{U}(f_2, f_1, e_1, e_2)$ to capture the pre (f_1) and post (f_2) names of f. The CDG for this example is shown in Figure 4, where each box is a node containing the statement numbers from Figure 3, and edges are labeled with the conditions that cause the execution of their target nodes.

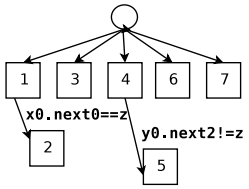

Figure 4: CDG

We define a function *guard*(s) that takes a statement s and returns an expression, which is the conjunction of all the control conditions associated with the edges of the reverse path from s to the root of the CDG. We also define a function $guard_\phi(x, x_i)$ that takes two variable (or field) names and returns the condition under which the ϕ statement $x = \phi(\cdots, x_i, \cdots)$ assigns x_i to x.

3.2 Translation

We consider a simple core Java-like language with no method calls, to illustrate our translation scheme. It has the syntax shown in Figure 5. The `return` statement indicates termination. The statement `assert` *Form* introduces a specification at a particular point in the code, where *Form* is given in the relational logic presented in Section 3.1.1.

$$
\begin{aligned}
JStmt ::= {} & JStmt;JStmt \mid \texttt{if } C \texttt{ then } JStmt \texttt{ else } JStmt \mid \\
& v \texttt{ = } JExpr \mid JExpr.f = JExpr \mid \\
& a[JExpr] = JExpr \mid \texttt{return} \mid \texttt{assert } Form
\end{aligned}
$$

$$
\begin{aligned}
JExpr ::= {} & v \mid JExpr.f \mid a[JExpr] \mid \\
& JExpr \ \{+, -, *, / \ \} \ JExpr
\end{aligned}
$$

$$
\begin{aligned}
C ::= {} & JExpr \texttt{ == } JExpr \mid !C \mid C \ \{\&\&, \|\} \ C \mid \\
& JExpr \ \{<, >, \leq, \geq\} \ JExpr
\end{aligned}
$$

Figure 5: Syntax

3.2.1 Basic Encoding

Each type in Java is represented by a universe of atoms. A field f of type B in class A is represented by a relation taking value in A `->` B, similar to a points-to relationship. This relation is a total function: each atom of A maps to exactly one atom in B (modeling the fact that the field points to a specific B-object), or the special `Null` atom (which represents the `null` value). A variable v of type A is represented by a unary relation taking value in A. This relation is a singleton set: it represents either an atom of A or the `Null` atom.

The basic encoding works as follows. First, we translate the code to SSA form and build a control dependence graph (CDG). We rely on a translation function $\mathcal{T}$ (defined below) that takes an expression and returns a relational expression in terms of initial state. Thus for any variable or field at any point in the code (in SSA form), $\mathcal{T}$ gives its value in terms of the initial state.

To simplify the exposition, assume that there is a single *assert* f statement in the code. We produce the logical formula:

$$
\begin{aligned}
initial \wedge \mathcal{T}(guard(\textit{assert } f)) \quad & \wedge \quad \neg f \\
& \wedge \quad x_1 = \mathcal{T}(x_1) \\
& \cdots \\
& \wedge \quad x_n = \mathcal{T}(x_n)
\end{aligned} \tag{1}
$$

where $x_1, \cdots, x_n$ are the free variables in f, denoted by $Vars(f)$. Formula *initial* constrains the initial state of the heap to be such that fields are functions of the appropriate types. Informally, formula (1) is true if the execution of the code reaches the *assert* statement, i.e. its guard is satisfied, and the assertion is not true. The guard is computed from the CDG as described in Section 3.1.2. Each of the $x_i = \mathcal{T}(x_i)$ formulae constrains variable x_i to be equal to its value in terms of the initial state. Formula (1) considers only a *slice* of the code in the sense that only the definitions of the variables needed for (dis)proving the assertion are included.

Translation function $\mathcal{T}$.

The translation function $\mathcal{T}$ is defined in Figure 6. The partial function *def* takes a variable or field (in SSA form) and returns the statement in the code that defines its value. Function *def* is undefined for initial variables and fields.

For a variable or field x (in SSA form), $\mathcal{T}(x)$ results in a relational expression giving the value of x in terms of initial state. $\mathcal{T}(x)$ takes into account the nature of the statement defining x. There may be none, in which case x is part of the initial state and *def*(x) is undefined. Otherwise the defining statement may be of the form of a variable assignment $x = e$, a field update $\mathcal{U}(x,x_0,e_1,e_2)$, or a ϕ statement $x = \phi(x_1, \cdots, x_n)$.

If *def*(x) is undefined then $\mathcal{T}(x)$ is just x. Otherwise, if *def*(x) is a variable assignment $x = e$ then, $\mathcal{T}(x)$ returns $\mathcal{T}(e)$. If *def*(x) is a field update $\mathcal{U}(x,x_0,e_1,e_2)$, then $\mathcal{T}(x)$ returns $\mathcal{T}(x_0)$ updated at $\mathcal{T}(e_1)$ with $\mathcal{T}(e_2)$. Finally, if *def*(x) is a ϕ statement $x = \phi(x_1, \cdots, x_n)$, then $\mathcal{T}(x)$ results in the union of set comprehensions. The set comprehension $\{ \mathcal{T}(x_i) \mid \mathcal{T}(guard_\phi(x,x_i)) \}$ is equivalent to $\mathcal{T}(x_i)$ if $\mathcal{T}(guard_\phi(x,x_i))$ is true, i.e. the ϕ statement assigns x_i to x, and to the empty set otherwise.

$\mathcal{T}$ translates a field dereference using the relational join operator. The rest of the cases are straightforward. Notice that $\mathcal{T}$ ensures that a variable or field is translated to an expression in terms of initial state.

3.2.2 Soundness Proof

In this section, we show that our basic encoding scheme is sound, meaning that the sliced formula that we generate is equivalent to one that represents the set of all executions of the code conjoined with the negation of the specification.

We can rewrite formula (1), our slice, as *initial* conjoined with the following formula, which we call f_1:

$$
\begin{aligned}
\mathcal{T}(guard(\textit{assert } f)) \quad & \wedge \quad \neg f \wedge \\
(\mathcal{T}(guard(def(x_1))) \quad & \Rightarrow \quad x_1 = \mathcal{T}(x_1)) \wedge \\
& \cdots \\
(\mathcal{T}(guard(def(x_n))) \quad & \Rightarrow \quad x_n = \mathcal{T}(x_n))
\end{aligned}
$$

$\mathcal{T}$: *JExpr* $\rightarrow$ *Expr*

e	$\mathcal{T}(e)$
x	if *def*(x) is: x — undefined $\mathcal{T}(e)$ — x=e $\mathcal{T}(x_0)$ ++ $\mathcal{T}(e_1)$->$\mathcal{T}(e_2)$ — $\mathcal{U}(x, x_0, e_1, e_2)$ $\{\mathcal{T}(x_1) \mid \mathcal{T}(guard_\phi(x,x_1))\}$ + $\cdots$ + $\{\mathcal{T}(x_n) \mid \mathcal{T}(guard_\phi(x,x_n))\}$ — x=$\phi(x_1,\cdots,x_n)$
$e.f$	$\mathcal{T}(e)$. $\mathcal{T}(f)$
e_1 && e_2	*if* ($\mathcal{T}(e_1)$=*True* $\wedge$ $\mathcal{T}(e_2)$=*True*) *then True else False*
e_1 || e_2	*if* ($\mathcal{T}(e_1)$=*True* $\vee$ $\mathcal{T}(e_2)$=*True*) *then True else False*
!e	*if* ($\mathcal{T}(e)$=*False*) *then True else False*
e_1 == e_2	*if* ($\mathcal{T}(e_1)$=$\mathcal{T}(e_2)$) *then True else False*
`null`	*Null*
`true`	*True*
`false`	*False*

Figure 6: Translation function $\mathcal{T}$

since $\mathcal{T}(guard(assert\ f)) \Rightarrow \mathcal{T}(guard(def(x_i)))$, for $1 \leq i \leq n$.

Let f_2 be the following formula:

$$\begin{array}{rcl} (\mathcal{T}(guard(def(x_{n+1}))) & \Rightarrow & x_{n+1} = \mathcal{T}(x_{n+1})) \\ \wedge & \cdots & \wedge \\ (\mathcal{T}(guard(def(x_m))) & \Rightarrow & x_m = \mathcal{T}(x_m)) \end{array}$$

where $x_{n+1}, \cdots, x_m$ are all the variables *not* appearing in specification f.

The set of all executions of the code conjoined with the negation of the specification is $initial \wedge f_1 \wedge f_2$. We want to show that $initial \wedge f_1$ (our slice) is satisfiable if and only if $initial \wedge f_1 \wedge f_2$ is satisfiable.

The intuition behind the proof is that f_1 and f_2 have no variables in common besides initial variables. Moreover, f_2 represents fragments of code that consist of: (i) statements involving irrelevant data, and (ii) any conditionals that cannot control whether or not the assertion is violated. So for a sufficiently large number of atoms, and an assignment to initial variables satisfying *initial*, the assignment can be extended to satisfy f_2. We capture this property with the definition below where we say that f_2 is *valid with respect* to *initial*. If we find a model for our slice, $initial \wedge f_1$, we can extend this model to satisfy f_2, the key being once again that f_1 and f_2 have no other variables in common besides those in *initial*.

Recall that in the relational logic, a variable denotes a set of tuples of atoms drawn from some universe. Given a set of variables V, let $\mathcal{M}_V$ denote an assignment of tuples of atoms to the variables in V, where the atoms are drawn from bounded domains. We call $\mathcal{M}_V$ a partial instance or model. For two disjoint sets of variables V and V', the partial model $\mathcal{M}_V \cup \mathcal{M}_{V'}$ assigns tuples to variables in V according to $\mathcal{M}_V$, and to variables in V' according to $\mathcal{M}_{V'}$. We call $\mathcal{M}_V \cup \mathcal{M}_{V'}$ an *extension* of $\mathcal{M}_V$. We write $\mathcal{M}_V \models f$, where $Vars(f) \subseteq V$, to denote that $\mathcal{M}_V$ satisfies formula f.

Definition. Let f and f' be two formulae and V a set of variables such that $V = Vars(f)$. We say that f' is *valid with respect to* f if and only if

$$\forall \mathcal{M}_V \ \ \exists \mathcal{M}_{Vars(f')-V} \mid \mathcal{M}_V \models f \Rightarrow \mathcal{M}_V \cup \mathcal{M}_{Vars(f')-V} \models f'$$

Informally, formula f' is valid with respect to f if and only if all models that assign to the variables of f and satisfy f can be extended to satisfy f' as well.

Lemma. Let f and f' be two formulae such that f' is valid with respect to f. Then $f \wedge f'$ is satisfiable if and only if f is satisfiable.

Proof.
$\Rightarrow$ Trivial.
$\Leftarrow$ Assume that f is satisfiable. Then there exists a partial model $\mathcal{M}_{Vars(f)}$ such that $\mathcal{M}_{Vars(f)} \models f$. Since f' is valid with respect to f, there exists a model $\mathcal{M}_{Vars(f')-Vars(f)}$ such that $\mathcal{M}_{Vars(f)} \cup \mathcal{M}_{Vars(f')-Vars(f)} \models f'$. Therefore $\mathcal{M}_{Vars(f)} \cup \mathcal{M}_{Vars(f')-Vars(f)} \models f \wedge f'$, and $f \wedge f'$ is satisfiable. □

Theorem. Let $\mathcal{I}$, f_1, and f_2 be formulae such that f_2 is valid with respect to $\mathcal{I}$, and $Vars(f_1) \cap Vars(f_2) \subseteq Vars(\mathcal{I})$. Then f_2 is also valid with respect to $\mathcal{I} \wedge f_1$.

Proof. Assume that f_2 is valid with respect to $\mathcal{I}$, and $Vars(f_1) \cap Vars(f_2) \subseteq Vars(\mathcal{I})$. Consider an arbitrary partial model $\mathcal{M}_{Vars(\mathcal{I} \wedge f_1)}$ such that $\mathcal{M}_{Vars(\mathcal{I} \wedge f_1)} \models \mathcal{I} \wedge f_1$. Let $\mathcal{M}_{Vars(\mathcal{I})}$ and $\mathcal{M}_{Vars(f_1)-Vars(\mathcal{I})}$ be two partial models such that $\mathcal{M}_{Vars(\mathcal{I})} \cup \mathcal{M}_{Vars(f_1)-Vars(\mathcal{I})} = \mathcal{M}_{Vars(\mathcal{I} \wedge f_1)}$.

Since f_2 is valid with respect to $\mathcal{I}$, there exists a model $\mathcal{M}_{Vars(f_2)-Vars(\mathcal{I})}$ such that $\mathcal{M}_{Vars(\mathcal{I})} \cup \mathcal{M}_{Vars(f_2)-Vars(\mathcal{I})} \models f_2$. Since $Vars(f_1) \cap Vars(f_2) \subseteq Vars(\mathcal{I})$, we can form a partial model $\mathcal{M}_{Vars(\mathcal{I})} \cup \mathcal{M}_{Vars(f_2)-Vars(\mathcal{I})} \cup \mathcal{M}_{Vars(f_1)-Vars(\mathcal{I})}$ which satisfies f_2. Therefore, $\mathcal{M}_{Vars(\mathcal{I} \wedge f_1)} \cup \mathcal{M}_{Vars(f_2)-Vars(\mathcal{I})} \models f_2$.

We have that $Vars(f_2) - Vars(\mathcal{I}) = Vars(f_2) - Vars(\mathcal{I} \wedge f_1)$. So let $\mathcal{M}_{Vars(f_2)-Vars(\mathcal{I} \wedge f_1)}$ be equal to $\mathcal{M}_{Vars(f_2)-Vars(\mathcal{I})}$. Therefore $\mathcal{M}_{Vars(\mathcal{I} \wedge f_1)} \cup \mathcal{M}_{Vars(f_2)-Vars(\mathcal{I} \wedge f_1)} \models f_2$, and f_2 is valid with respect to $\mathcal{I} \wedge f_1$. □

Recall that the set of all executions of the code conjoined with the negation of the specification is $initial \wedge f_1 \wedge f_2$. We want to show that $initial \wedge f_1$ (our slice) is satisfiable if and only if $initial \wedge f_1 \wedge f_2$ is satisfiable. We know that the variables in common for f_1 and f_2 are in *initial*, and that f_2 is valid with respect to *initial*. So f_2 is also valid with respect to $initial \wedge f_1$ by the Theorem. Therefore by the Lemma, $initial \wedge f_1$ (our slice) is satisfiable if and only if $initial \wedge f_1 \wedge f_2$ is satisfiable.

3.2.3 Encoding Integers

Two encodings of integers co-exist in our approach. First, we take advantage of bitsets in the relational logic, which allow a representation with efficient operations. This encoding is good for manipulating integers in isolation but cannot be used for placing them in relations (because relations are between atoms). So we also provide a representation as sums of powers of two. Instead of having an atom to represent each integer, we have one atom for each power of two within some user-given range. We represent an integer as a set of such atoms, where its value is their sum. We define two functions *toAtoms*() and *toBits*() to go back and forth between

$toAtoms : (BitSet + Integer) \rightarrow Integer$
$toBits : (BitSet + Integer) \rightarrow BitSet$

$$toAtoms(e) = \begin{cases} \{B{:}\ Integer \mid (e \wedge_b (sum\ B))\ != 0\} & if\ e\ is\ a\ BitSet \\ e & otherwise \end{cases}$$

$$toBits(e) = \begin{cases} e & if\ e\ is\ a\ BitSet \\ sum\ e & otherwise \end{cases}$$

e	$\mathcal{T}(e)$
x	$\begin{cases} \ldots & \text{if } def(x) = \\ \mathcal{T}(f_1)\texttt{++}\mathcal{T}(e_1)\texttt{->}toAtoms(\mathcal{T}(e_2)) & \mathcal{U}(f_1,f_2,e_1,e_2) \end{cases}$
$e_1\ op\ e_2$	$toBits(\mathcal{T}(e_1))\ op\ toBits(\mathcal{T}(e_2))$ $op \in \{+, -, *, /, >, <, \leq, \geq\}$

Figure 7: Additional translation rules for Integers

the two encodings as needed. Figure 7 shows the additional rules for $\mathcal{T}$ in the presence of integers.

We use the set *Integer* for the universe of atoms representing powers of two. The expression *sum* e, where e is a set of *Integer*s, returns a bitset representing the sum of all the atoms in that set. When e is a bitset, $toAtoms(e)$ returns the set of all atoms B such that the result of the bitwise-and of e with the sum of B is different from 0. This operation effectively gathers the corresponding atom for each non-zero bit in e in the resulting set.

3.2.4 Encoding Arrays

Our encoding of arrays is designed to support large sparse arrays, but accommodating large (integer-valued) indices. We represent each array object with an atom from the set *ArrayObj*. We use integers encoded as sets of powers of two to index into arrays. Since each integer is represented as a set, it cannot be used directly for this purpose. Instead we do indexing indirectly through special atoms from a set *IndexAtom* that indicate what indices in an array have a non-default value.

Three special relations are used to encode arrays:
T: $ArrayObj \rightarrow IndexAtom$
I: $ArrayObj \rightarrow IndexAtom \rightarrow Integer$
V: $ArrayObj \rightarrow IndexAtom \rightarrow ArrayValue$

Relation T indicates what indices have non-default values for each array object; I what integer corresponds to each pair of array object and index atom; and V what value is stored at each pair of array object and index atom. The set *ArrayValue* represents the set of all possible types that may be contained in an array in our code.

We impose some initial conditions on these relations (which we include in the *initial* formula described above). These state that: (i) No two index atoms map to the same integer; (ii) An index atom appears in a tuple of T if and only if it appears in a tuple of V; (iii) If an index atom appears in I, it must appear in T; (iv) All index atoms must be mapped to integers between 0 and the length of the corresponding array. Notice that it may be the case that an index atom appears in T and V but not in I. In this case, that index atom is interpreted to represent index 0.

At each array store, the post-values of these relations are such that these conditions are preserved. First, we check if an index atom exists at the desired integer-valued index. If so, we update V at that index atom. If not, we create such an atom, and update all three relations accordingly. If we are storing a default value (null for references and 0 for integer-valued arrays), then we remove the tuples containing the index atom corresponding to the desired integer-valued index from all three relations. Points at which an array contains default values are therefore represented implicitly.

At each array load, we first check if there exists a corresponding index atom at the desired integer-valued index. If so, the array load translates to the appropriate value from V and if not to the default value.

4. EVALUATION

4.1 Implementation

Our prototype implementation of Miniatur can analyze most of the Java language. Its inputs consist of a procedure containing assertions, as well finitization parameters such as: number of loop unwindings, integer bit-width, number of objects in the initial heap, and number of atoms used to index arrays. The assertions may be either traditional Java assertions, or special assertions containing a first-order logic formula[1]. The output of our tool is an initial heap and parameters, from which the execution of the code results in an assertion violation, if one exists.

Miniatur starts by building a call-graph, inlining all method calls, and unwinding loops up to the user-provided number. For example, the code `while(n!=null) n=n.next;` becomes:

```
if (n!=null) {
  n=n.next;
  if (n!=null) n=n.next; else return;
}
```

for 2 unwindings. This causes a loop requiring more than 2 unwindings to immediately terminate the code.

Miniatur determines the types needed in the analysis and bounds on these types as follows. To minimize the types required, we generate only those that may be needed by the program. These fall into four categories: (i) types named explicitly in type checking operations (i.e., casts and `instanceof` operations), (ii) types used in dynamic dispatch operations, that is, potential receivers of virtual calls, (iii) types used in field reads and writes, and (iv) types of arguments and transitively reachable fields from those arguments, obtained using pointer analysis information.

To bound each type, we combine the user-provided number of objects on the initial heap with additional atoms to represent the objects allocated in the program being translated. This has the advantage that we cannot get into situations where the program itself is unsatisfiable simply because there are too few objects. For the additional atoms, we count all `new` operations for each type encountered during encoding. For each such `new` operation we allocate a fresh atom to represent that object.

We implemented the encodings of Sections 3 using the WALA program analysis framework [34], the relational encoding API of KodKod [28], and the MiniSAT solver.

4.2 Experiments

To evaluate our approach, we used two sets of benchmarks: checking equality contract properties on a variety of open-source programs, and structural properties of

[1]We use the KodKod API to specify the special assertions in our experiments.

classes taken from the Java Collections Framework such as `java.util.HashMap`. In the experiments discussed below, we allow up to 4 objects of every concrete type in the initial heap. For other parameters, we use 16 bits[2] to represent Java integers, 10 array index atoms and unrolled loops 3 times. We report measurements of time spent creating the initial call graph and related structures (denoted *CFA*), time spent encoding in our model (denoted *Encode*), and time spent in the SAT solver (denoted *Solve*). All reported times were obtained on machines with 2 dual-core 3.8GHz Intel Xeon processors and 5GB of RAM running the standard 1.5.0_06 JDK for IA32 and the Linux operating system. We also report a sum of source lines of the program with inlined methods, but ignoring the effect of loop unrolling (denoted *Size* in all tables).

4.2.1 Checking Method Contracts

We used Miniatur to check most aspects of the Java identity contracts given in the JavaDoc for `equals()` and `hashCode()` methods of `java.lang.Object` and the `compareTo` method of `java.lang.Comparable` [10]. In particular, we check that: (i) `equals()` methods are reflexive, (ii) symmetric, and (iii) transitive, and that (iv) `x.equals(null)` is false for non-`null` `x`. We also check that: (v) any two equal objects (via `equals()`) have the same `hashCode()`. We check all four aspects of the `compareTo` contract: (vi) it is anti-symmetric, and (vii) transitive. Moreover, (viii) if `x.compareTo(y)==0` then `signum(x.compareTo(z)) == signum(y.compareTo(z))` for all `z`, and (ix) it is consistent with `equals()`.

For our experiments, we encoded these contract requirements directly with a Java `assert` statement, and we checked each property for each applicable type in the program being verified. For instance, we used a test function for property `(v)` defined as follows:

```
public static void equalsTester(Object a, Object b) {
  if (a.equals(b)) assert a.hashCode()==b.hashCode();
}
```

We chose open-source programs with interesting `equals()`, `hashCode()` and/or `compareTo()` methods: *Antlr* (the popular parser generator), *Bcel* (Apache Bytecode Engineering Library), *Hsqldb* (database server), and *JavaCup* (an LALR(1) parser generator). We did not check any code that required floating point numbers, and commented it out as needed, and left the code otherwise unmodified.

Figure 8 presents the contract violations found (left) and Miniatur's performance (right). We list classes exhibiting violations of the contracts. In general, while such violations indicate fragile code and potential bugs, they may not manifest themselves when the application is executed. `JavaCup` had a single violation: the `terminal_set` class overrides `equals()` but not `hashCode()`. This bug would be trivial to find with cheaper techniques [17]; however, it is notable because it required the solver to find instances of a `java.util.BitSet` field of `terminal_set`, which stresses the model of integers and of arrays. The two classes in *Hsqldb* with violations are related by inheritance and have `equals()` methods that use `instanceof` tests for their respective classes, and hence they violate the symmetry requirement (ii).

[2]Notice this is a considerable improvement over previous approaches which could handle at most 4-bit integers.

The violations of the `compareTo` contract (properties (vi) and (vii) in *Antlr* are subtler: `compareTo()` in `Label` subtracts an integer field: `return this.label-((Label)o).label;`, which looks fine at first glance. However, exhaustive analysis reveals this code has symmetry and transitivity violations involving the smallest possible integer value. Due to modulo arithmetic MIN_VALUE − 0 is negative, but so is 0 − MIN_VALUE, violating (vi). This can be used to construct violations of (vii) as well. The violations of (v) are mixed: for some types the error is not obvious at all at first glance, while others are trivially wrong.

The symmetry and transitivity violations (properties (ii) and (iii)) in *BCel* indicate fragile code. The `equals()` methods depend upon an implicit relationship between concrete classes and the value of an integer field defined in a base class used as a tag (the `type` field for the classes ending in Type and `opcode` for `IF_CMPEQ`). These tag fields are intricately related to a multi-level class hierarchy in each case. Applications that extend `BCel` classes must be aware of these relationships because subtle contract-related bugs might arise otherwise. The instruction classes are especially vulnerable since their `equals()` method delegates to a user-specifiable comparator object, obscuring the vital role of the `opcode` field. The violations of (v) in *BCel* occur in these same classes, and involve classes with asymmetric `equals()` methods that also define `hashCode()`.

The performance results in Figure 8 report the time taken to analyze each contract for each benchmark. The question marks in this figure indicate tests where our analysis took too long or did not terminate. Overall, most times are reasonable, the average being within roughly two minutes. The major exceptions are checking `equals()` /`hashCode()` consistency (property v) for *BCel* and *Hsqldb*, and checking symmetry (property (ii) for *Hsqldb*. This is due to more complex data structures and non-trivial arithmetic. The number of lines of code analyzed range from a few lines to a few thousand and illustrate the improved scalability of our technique over previous work [20, 31, 26, 9].

4.2.2 Checking Class Invariants in `java.util`

We also evaluated our technique by checking class invariants in the `java.util` package (JDK 1.4). The classes were unmodified, except for portions of `HashMap` and `HashSet` that use floating point numbers.

We present two sets of experiments. In the first, we checked a class invariant: that the number of objects in a collection is equal to its `size` integer field, with a client that performs a series of insertions in the collection, from an array of objects given as argument. For each collection, the specification that we checked is given in Figure 9. We use transitive closure and set cardinality to obtain the number of elements in each data structure. Figure 9 also shows time to check those properties. More complex collections take longer to check than the simpler ones, but in little more than a minute at most. The large size of the `TreeMap` and `TreeSet` is due to the inlining of many method calls that are required for keeping the red-black tree structure well-formed. Notice that our tool handles over 12000 lines of such code in about a minute.

The second set of experiments involves a synthetic client that tests both the client and the library code to find concrete examples of misuse of the collections API (see Fig-

Violations

Prop.	Type	Time (seconds) Encode	Time (seconds) Solve	size (lines)
	BCel (*all types in* `org.apache.bcel`)			
(ii)	`generic.ArrayType`	1.7	113.7	1748
	anonymous `ReferenceType`	1.1	89.2	1656
(iii)	`generic.IF_CMPEQ`	1.2	102.8	1844
	anonymous `Type`	2.5	75.8	1662
	`generic.ArrayType`	1.2	82.7	1846
(v)	`generic.ArrayType`	0.9	2493.1	580
	`generic.LocalVariableGen`	0.9	1589.5	512
	anonymous `Type`	0.9	1318.5	443
	`generic.MONITOREXIT`	.9	1757.4	531
	Antlr (*all types in* `org.antlr`)			
(v)	`analysis.Label`	3.8	272.5	212
	`analysis.Transition`	4.8	386.3	249
	`analysis.SemanticContext.NOT`	3.8	251.5	272
	`misc.Interval`	3.5	295.5	171
	`stringtemplate.language.FormalArgument`	3.6	350.7	179
(vi)	`analysis.Label`	6.9	0	26
	`analysis.Transition`	3.5	0	30
(vii)	`analysis.Label`	3.9	0	21
	`analysis.Transition`	3.3	0	29
	Hsqldb (*all types in* `org.hsqldb`)			
(ii)	`CachedRow`	1.5	580.1	1097
	`Row`	1.5	593.1	1097
	JavaCup			
(v)	`java_cup.terminal_set`	0.5	6.3	286

Performance

Part	Ranges (smallest/average/biggest) Size (lines)	Encode (secs)	Solve (secs)
	Antlr (18 `equals`, 2 compareTo)		
(i)	3/39/173	3.1/3.5/6.7	0/1.2/4.7
(ii)	657/693/827	5.2/5.5/6.6	0/94.7/451.3
(iii)	656/728/996	5.1/5.8/10.1	0/56.5/337.9
(iv)	3/39/173	3.1/3.4/4.3	0/0.8/4.9
(v)	164/208/350	3.5/3.8/4.8	0/108/386.3
(vi)	26/28/30	3.49/5.2/6.9	0/0/0
(vii)	21/25/29	3.3/3.6/4	0/0/0
(viii)	31/35/39	3.3/3.7/4.1	0/0.1/0.1
(ix)	26/45/64	4.2/6.9/9.5	20.5/20.7/21
	BCel (19 `equals`)		
(i)	3/89/438	0.7/1.1/2.3	0/2/31
(ii)	1649/1735/2084	1.1/1.7/3.1	0/56.3/169
(iii)	1648/1820/2518	1.14/1.52/2.5	0/49.9/102.8
(iv)	3/86/438	0.7/1/3.2	0/1.9/33.9
(v)	433/541/871	0.86/1/1.8	0/896.4/5705.3
	Hsqldb (8 `equals`)		
(i)	3/137/983	1/1.4/3.5	0/1.5/7.1
(ii)	1085/1219/2065	1.4/1.7/2.9	48.2/335.6/756.6
(iii)	1084/1353/3044	1.3/1.4/1.8	54.4/124/524
(iv)	3/137/983	1/1.6/3.4	0/1.5/7.7
(v)	996/??/??	3.6/??/??	??/??/??
	JavaCup (14 `equals`)		
(i)	5/56/123	0.2/0.2/0.3	1.9/2.4/2.8
(ii)	378/404/437	0.3/0.4/0.5	3/3.5/4.3
(iii)	377/429/495	0.3/0.4/0.6	2.9/3.4/4.5
(iv)	7/32/66	0.2/0.3/0.9	2.3/2.7/4.2
(v)	247/290/362	0.3/0.3/0.5	3.2/4.5/25.2

Figure 8: Equals Contract Checking Results. Missing rows indicate no applicable violations or methods.

ure 1(b) for a client using a `HashSet`). The client removes duplicates from an array of values given as its argument, to produce a set (bag to set transformation). We used this client with four collection types—`ArrayList`, `LinkedList`, `HashSet`, `TreeSet`—and with element types that were correct and ones that contained violations of `equals()`, `hashCode()` and/or `compareTo()` contracts.

The results are shown in Figure 10. Cases with a client misuse of the collections API are analyzed more quickly than those without one. `HashSet` and `TreeSet` are much slower than the simpler collections, due to their complex structures and having much more complex `toArray` methods involving multiple levels of derived collections and iterators. The case of no misuse of `HashSet` is an outlier, taking more than half and hour to solve. This is probably because the complex computations performed by `hashCode()` made SAT solver heuristics less effective. Notice that the `TreeSet` benchmarks consist of over 13000 lines of code and run in a few minutes.

class	*formula*
`LinkedList`	$this.size = \|this.header.next^* - null\|$
`HashMap`	$this.size = \|this.table[].next^* - null\|$
`HashSet`	$this.map.size = \|(this.map.table[].next^* - null\|$
`TreeMap`	$this.size = \|count(this.root.(left + right)^* - null\|$
`TreeSet`	$this.m.size = \|this.m.root.(left + right)^* - null\|$

class	Time (seconds) *CFA*	*Encode*	*Solve*	*size (lines)*
`LinkedList`	7.2	0.2	0.8	111
`TreeMap`	6.7	4.4	56.3	12592
`TreeSet`	7.8	3.6	68.6	12667
`HashMap`	10	0.6	19.4	1107
`HashSet`	11.4	0.8	22.6	1129

Figure 9: Structural properties of `java.util` and verification results

5. RELATED WORK

We build on work checking heap-manipulating code with Alloy [18, 19] and SAT solving. Jalloy [20, 31] represents control edges and intermediate data values explicitly with variables. It optimizes the functional relations that represent fields. Taghdiri [26] uses a similar approach that discovers method specifications at the logical level and provides better scalability. Dennis et al. [9] improve on the explicit representation of Jalloy, and evaluate their technique by checking linked list benchmarks against JML specifications. They use KodKod [28], a relational model finder with powerful optimizations, which we also use. We improve on

Class	Time (seconds) *CFA*	*Encode*	*Solve*	*found bug?*	*size (lines)*
ArrayList	9.1	.3	3.2	yes	445
ArrayList	11.0	.3	62.0	N/A	262
HashSet	9.1	0.4	136.1	yes	1180
HashSet	9.1	0.4	1946.0	N/A	874
LinkedList	11.3	0.4	2.1	yes	438
LinkedList	10.5	0.4	83.9	N/A	258
TreeSet	8.2	4.0	117.1	yes	13114
TreeSet	7.0	4.6	172.7	N/A	13024

Figure 10: Bag-To-Set Analysis Results

these approaches with much better scalability due to a slicing algorithm at the logical level, and can handle realistic fragments of code that manipulate both heap-allocated objects and integers.

Model checkers such as SLAM [2], BLAST [16], SMV [24], NuSMV [5], BMC [3] are very successful at checking event sequences and temporal logic formulae, whereas we focus on structural properties of heap-manipulating code. We also support modular checking of specifications and contracts, meaning that components can be checked in an open-world against all contexts. SATURN [35] is also based on SAT solving and performs slicing at the constraint level. Our slicing algorithm is similar at a high level, but we exploit the machinery of SSA form and the CDG to capture data- and control-dependencies more precisely in our initial formulation, so we do not need later optimizations such as their use of BDDs to simplify control flow guards. SATURN does not support the rich specification language that we consider in our technique.

Approaches such as TestEra [23], Korat [4], Java PathFinder [21] generate all non-isomorphic inputs to a program within some bounds using datatype invariants, and check every path exercised by each input by executing the code. Glass-box model checking [8] improves on these techniques by further pruning the search space, and soundly removing redundant inputs and operations. These techniques are successful at checking certain data structure properties, but not suitable for checking contract violations such as the `equals()` /`hashCode()`, especially if the classes being checked have integer-valued fields.

ESC/Java [11] uses a theorem prover to check for array bounds errors and null pointer dereferences. It does not support the specification language we consider in our technique. TVLA [22] is a shape analysis tool that provides an over-approximation of code's behavior and may therefore issue false error reports. It also does not support modular verification. It has been used successfully to check linked data structures such as linked lists and doubly linked lists, but not hashmaps and treesets, and properties involving both structure and integers considered in this paper.

A program slice [27] was originally defined as the set of statements that may affect the values computed for a set of variables at a designated point of interest. We are aware of several model-checking projects where variations on program slicing are applied to a source-level or intermediate-level representation of a program before deriving formulae from the program that are analyzed by a model-checker to verify correctness.

Heitmeyer et al. [15] present an approach where the Spin model checker is used to compare a property-based specification against a state machine that describes a system operationally. Heitmeyer et al. use a form of slicing in which the set of variable names that occur in the logical formula that encodes the property being checked are used to remove unnecessary variables and their definitions from the analysis.

Hatcliff et al. [14] present a system for model checking of various temporal properties of concurrent programs. In this work, a variation on Weiser's slicing algorithm is used to remove program parts that are irrelevant for the properties under consideration. The Bandera model checker [6] features a program slicer that eliminates the parts of a program that are irrelevant for the verification of temporal properties such as deadlock freedom in concurrent Java programs. This slicer employs a specialized algorithm that accommodates Java's concurrency model [13] and operates on the intermediate representation of Soot [30]. The Bandera program slicer was also used in the context of Java PathFinder [33].

Millett and Teitelbaum [25] present an approach for slicing programs written in Promela, the input language for the Spin model checker [1], and demonstrate that slicing away computations unrelated to assertions of interest can yield significant speedups. In contrast to our work, where slicing is performed at the level of the generated formulae, Millett and Teitelbaum extend System Dependence Graphs (SDGs) to account for the communication constructs in Promela and use a variation on the SDG-based slicing algorithm in CodeSurfer [12] to slice programs at the source level.

The main difference between the use of slicing-like techniques in our work and in the model checking approaches discussed above [14, 25, 33] is the fact that we perform slicing at the level of the generated relational formulae, and is similar in this respect to the technique used in [35]. If we were to use slicing at the source level the slices would have to be executable programs, otherwise it would not be clear how to guarantee counter-examples that correspond to actual executions. Slicing at the logical level includes only statements involved in computations that the specification depends on.

Uzuncaova and Khurshid [29] present a slicing technique for Alloy [18] based on a partition of core vs. derived relations. Their approach is heuristical, however, and they do not give a precise condition for slicing relational formulae as we do.

6. CONCLUSIONS

We presented a method for finding bugs in code based on relational logic and SAT solving, which provides much better scalability than prior work. It can check a mixture of structural and numerical properties written in a rich specification language on realistic fragments of programs. To the best of our knowledge, ours is the first fully automated tool that checks equality contract violations of real programs. In the future, we plan to extend this technique to leverage different program analyses (such as points-to analyses) to reduce the search space of the SAT solver, and attempt to improve the performance of arithmetic operations.

Acknowledgments. We are grateful to Mark Wegman for many useful discussions, and Emina Torlak for her help with the KodKod tool. We also thank Stephen Fink, Christian Hammer, Daniel Jackson, Adam Kieżun, Alexey Loginov, and Emina Torlak for their feedback.

7. REFERENCES

[1] Spin model checker. `http://spinroot.com/spin/whatispin.html`.

[2] Ball, T., and Rajamani, S. K. The SLAM project: Debugging system software via static analysis. In *POPL'02: Proceeding of the Symposium on the Principles of Programming Languages* (2002).

[3] Biere, A., Cimatti, A., and Clarke, E. Symbolic model checking without BDDs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (1999).

[4] Boyapati, C., Khurshid, S., and Marinov, D. Korat: Automated testing based on Java predicates.

In *ISSTA'02: Proceedings of the International Symposium on Software Testing and Analysis* (2002).

[5] Cimatti, A., Clarke, E., Groce, A., Jha, S., and Veith, H. NuSMV: A new symbolic model verifier. In *CAV'99: Proceeding of the International Conference on Computer-Aided Verification* (1999).

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

[7] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Efficiently computing static single assignment form and the control dependence graph. *ACM TOPLAS 13*, 4 (1991), 451–490.

[8] Darga, P., and Boyapati, C. Efficient software model checking of data structure properties. In *OOPSLA'06: Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (2006).

[9] Dennis, G., Chang, F. S.-H., and Jackson, D. Modular verification of code with sat. In *ISSTA'06: Proceedings of the International Symposium on Software Testing and Analysis* (2006).

[10] Java `equals()` and `compareto()` contracts. `http://java.sun.com/j2se/1.5.0/docs/api/java/lang/package-summary.html`.

[11] Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R. Extended static checking for Java. In *PLDI'02: Proceedings of the International Conference on Programming Language Design and Implementation* (2002).

[12] Grammatech, I. Codesurfer. `http://www.grammatech.com/products/codesurfer/index.html`.

[13] Hatcliff, J., Corbett, J. C., Dwyer, M. B., Sokolowski, S., and Zheng, H. A formal study of slicing for multi-threaded programs with JVM concurrency primitives. In *SAS '99: Proceedings of the International Symposium on Static Analysis* (1999).

[14] Hatcliff, J., Dwyer, M. B., and Zheng, H. Slicing software for model construction. *Higher Order Symbol. Comput. 13*, 4 (2000), 315–353.

[15] Heitmeyer, C., James Kirby, J., Labaw, B., Archer, M., and Bharadwaj, R. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Trans. Softw. Eng. 24*, 11 (1998), 927–948.

[16] Henzinger, T. A., Jhala, R., Majumdar, R., and Sutre, G. Lazy abstraction. In *Symposium on the Principles of Programming Languages* (2002).

[17] Hovemeyer, D., and Pugh, W. Finding bugs is easy. *SIGPLAN Notices 39*, 12 (2004), 92–106.

[18] Jackson, D. Automating first-order relational logic. In *FSE'00: Proceedings of the International Symposium on Foundations of Software Engineering* (2000).

[19] Jackson, D., Shlyakhter, I., and Sridharan, M. A micromodularity mechanism. In *FSE / ESEC'01: Proceedings of the International Symposium on Foundations of Software Engineering / European Software Engineering Conference* (2001).

[20] Jackson, D., and Vaziri, M. Finding bugs with a constraint solver. In *ISSTA'00: Proceedings of the International Symposium on Software Testing and Analysis* (2000).

[21] Khurshid, S., Pasareanu, C., and Visser, W. Generalized symbolic execution for model checking and testing. In *TACAS'03: Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2003).

[22] Lev-Ami, T., and Sagiv, M. TVLA: A system for implementing static analyses. In *Proceedings of the International Symposium on Static Analysis* (2000).

[23] Marinov, D., and Khurshid, S. TestEra: A novel framework for automated testing of Java programs. In *ASE'01: Proceedings of the International Conference on Automated Software Engineering* (2001).

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

[25] Millett, L. I., and Teitelbaum, T. Slicing Promela and its applications to model checking. In *Proceedings of the 4th International SPIN Workshop* (1998).

[26] Taghdiri, M. Inferring specifications to detect errors in code. In *Proceedings of the International Conference on Automated Software Engineering* (2004).

[27] Tip, F. A survey of program slicing techniques. *Journal of Programming Languages 3*, 3 (1995), 121–189.

[28] Torlak, E., and Jackson, D. Kodkod: A relational model finder. In *TACAS'07: Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2007).

[29] Uzuncaova, E., and Khurshid, S. Kato: A program slicing tool for declarative specifications. In *International Conference on Software Engineering* (2007).

[30] Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., and Sundaresan, V. Soot - a Java bytecode optimization framework. In *CASCON '99: Proceedings of the Conference of the Centre for Advanced Studies on Collaborative research* (1999).

[31] Vaziri, M., and Jackson, D. Checking properties of heap-manipulating procedures with a constraint solver. In *TACAS'03: Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2003).

[32] Vaziri, M., Tip, F., Fink, S., and Dolby, J. Declarative object identity using relation types. In *ECOOP'07: Proceedings of the European Conference on Object-Oriented Programming* (2007), pp. 54–78.

[33] Visser, W., Havelund, K., Brat, G. P., Park, S., and Lerda, F. Model checking programs. *Autom. Softw. Eng. 10*, 2 (2003), 203–232.

[34] Watson Libraries for Analysis (WALA). `http://wala.sourceforge.net/`.

[35] Xie, Y., and Aiken, A. Saturn: A scalable framework for error detection using boolean satisfiability. *Transactions on Programming Languages and Systems*. to appear.