Bounded Program Verification using an SMT Solver: A Case Study

Tianhai Liu Michael Nagel Mana Taghdiri
Karlsruhe Institute of Technology, Germany
{*liu, s_nagel2, taghdiri*}*@ira.uka.de*

***Abstract*—We present a novel approach to *bounded program verification* that exploits recent advances of SMT solvers in modular checking of object-oriented code against its full specification. Bounded program verification techniques exhaustively check the specifications of a bounded program with respect to a bounded domain. To our knowledge, however, those techniques that target data-structure-rich programs reduce the problem to propositional logic directly, and use a SAT solver as the backend engine. Scalability, therefore, becomes a major issue due to bit blasting problems.**

In this paper, we present a novel approach that translates bounded Java programs and their JML specifications to quantified bit-vector formulas (QBVF) with arrays, and solves them using an SMT solver. QBVF allows logical constraints that are structurally closer to the original program and specification, and can be significantly simplified via high-level reasonings before being flattened in a basic logic. We also present a case study on a large-scale implementation of Dijkstra's shortest path algorithm. The results indicate that our approach provides significant speedups over a SAT-based approach.

***Keywords*-Bounded verification, Scope-bounded checking, SMT, Quantified bit-vector, JML, Dijkstra shortest path.**

I. Introduction

Bounded program verification (a.k.a. static scope-bounded checking) (e.g. [12, 13, 27–29, 33, 34, 36]) has become an increasingly attractive choice for gaining confidence in the correctness of software. Bounded program verification techniques statically check full functional properties of a bounded program (in which loops and recursions are unrolled) with respect to a bounded domain (in which the number of elements of each type is bounded).

Similar to bounded model checkers (e.g. [1, 30]), bounded verification tools perform an exhaustive analysis with respect to the given bounds, are fully automatic, and require little intermediate annotations. However, unlike model checkers that focus on temporal safety properties of entire programs, bounded verification approaches are modular—they can check program methods in isolation, against specifications—and target data-related properties of data-structure-rich programs—those that manipulate object configurations in the heap. Such properties can also be verified by full verification engines such as theorem provers (e.g. [4]), SMT-based proof engines (e.g. [3]), and shape analyses (e.g. [23]). Although such approaches provide proofs, they require extensive user-provided annotations (e.g. loop invariants) and interactions, or are not easily extensible to arbitrary data structures, or do not guarantee conclusive termination.

To our knowledge, all existing bounded verification techniques translate object-oriented programs and their specifications to a boolean satisfiability problem (possibly via an intermediate relational logic), and solve it using an off-the-shelf SAT solver. Although they have been successfully used to find bugs in various programs, such a direct translation to SAT (known as bit blasting) limits their scalability: they can check code with respect to only a few objects and loop unrollings, especially when the code also contains integer expressions and array objects.

In this paper, we present a novel approach to bounded program verification that exploits recent advances in SMT solvers to provide better scalability. We introduce a translation of object-oriented programs and their specifications to quantified bit-vector formulas (QBVF) with arrays, which can be solved efficiently using recent SMT solvers (e.g. Z3 [10] versions 2.17 and later). Compared to SAT-based bounded verifications that only perform boolean-level simplifications (e.g. shared expression detection and symmetry breaking), a QBVF solver performs high-level reasoning and simplifications (such as heuristic quantifier instantiations and template-based model finding), then flattens the formula and analyzes it in a quantifier-free SMT logic, and uses bit-blasting only as a last resort. This significantly improves the scalability of the solver [39].

Of course applying SMT solvers in the context of program verification is not new. Boogie [3] and ESC/Java [14], for example, have successfully used SMT solvers to verify object-oriented programs. These tools, however, analyze programs with respect to unbounded types. Quantifying over such types makes their target logics undecidable, and thus a conclusive analysis is not guaranteed. On the other hand, model checkers such as LLBMC [30] and SMT-CBMC [1] target a decidable SMT logic with bit-vectors and arrays. Their translations, however, are optimized for checking legal memory-accesses and control-related properties; they do not address data-structure properties of object-oriented programs. Armando et.al. [1] report that in CBMC, an SMT solver often performs better than a SAT solver. The comparison, however, is performed only in the context of bounded model checking where all benchmarks consist of only primitive integer types and arrays; object-oriented programs are not considered.

The technique presented in this paper targets Java programs and JML specifications. The user selects a member method of a Java class for analysis and provides its (arbitrarily partial) pre- and post-conditions. He also provides bounds on (1) the number of loop and recursion unrolling, (2) the number of objects of each class, and (3) the bit-width for integer values. Our technique automatically translates the code and specifications to the decidable SMT logic of QBVF with arrays, in which all types are encoded as bit-vectors, and fields as arrays over bit-vectors. Since bit-vectors provide a flat type system with a fixed range of values, translating Java class hierarchy, dynamic object allocation, null value, array objects, and JML reachability expression efficiently becomes challenging. We handle such constructs via additional uninterpreted functions and axioms, and incorporate them into JML specifications when necessary. The generated formulas are handed to the Z3 SMT solver for a possible counterexample. All counterexamples are guaranteed to be sound with respect to the bounds, but lack of a counterexample does not constitute proof.

We have implemented this technique in a prototype tool, InspectJ, that currently handles a subset of Java and JML. In addition to checking post-conditions and class invariants, InspectJ checks for two built-in exceptions of null-pointer dereferencing and array index out of bounds. We have used InspectJ to check properties of a large-scale implementation of the Dijkstra shortest path algorithm [11]. To our knowledge, all previous attempts to verify the implementations of this algorithm either abstracted away most data structures [20] or checked a textbook implementation with basic data structures [24]. The implementation we chose is the basis of various optimizations for computing shortest paths in large graphs (with millions of nodes), and thus depends on interconnected, efficient data structures. We found three previously-unknown bugs. Furthermore, to evaluate the performance of our approach, we compared InspectJ with JForge [12], a SAT-based bounded verification tool, on various methods of the Dijkstra code. The results show that InspectJ can analyze code with respect to larger bounds, and can even achieve better code coverage.

II. Background

A. Programs

We focus on analyzing object-oriented programs, and currently support a basic subset of Java that does not include real numbers, concurrency, and user-defined exceptions. Figure 1 gives a grammar for supported programs, assuming that the expressions are side-effect free.

We support a class hierarchy definition without interfaces and abstract classes. Methods are assumed to have a single return point. Loops are allowed, but are unrolled (based on a user-provided limit) in a pre-processing phase. Unrolling a loop *while (c) s;* twice will generate the nested branch statement *if (c)* {*s; if (c)* {*s; assume(!c)*}}. The pre-processing phase also decomposes allocation statements. That is, the Java statement *x = new T(*e_1*, .., *e_n*);* is broken into two consecutive statements for the actual allocation and for field initialization[1]: *x = new T; x.init(*e_1*, .., *e_n*);* In the rest of this paper, we assume that loops and recursive calls are already unrolled, and allocation statements are decomposed.

```
Prog ::= ClassDcl*
ClassDcl ::= class Class [extends Class]
             {FieldDcl* ProcDcl*}
FieldDcl ::= Access Type Field
ProcDcl ::= Access Type Proc(ParDcl*){Stmt}
ParDcl ::= Type Var
Stmt ::= Var = Expr | Var[Expr] = Expr
   | Expr.Field = Expr | [var =] Proc(Expr*)
   | Var = new Class(Expr*) |new Class()[Expr]
   | if (Expr) Stmt [else Stmt]
   | while (Expr) Stmt | Return [Var]
   | continue | break | Stmt; Stmt
Expr ::= Const | Var | Var[Expr] | Expr.Field
   | Expr instanceof Class | (Class) Expr
   | Expr BinOp Expr | !Expr
BinOp ::= + | - | * | / | >> | << | >>>
   | & | | | < | > | == | != | && | ||
Const ::= null | true |false | 0 | 1 | -1 |...
Type ::= Class | boolean | int
   | Class[] | boolean[] | int[]
Access ::= public | protected | private
Proc, Var, Class, Field ::= Identifier
```

Figure 1: Program Syntax

B. Specifications

We currently check Java programs against specifications written in a basic subset of JML (Java Modeling Language) [18] that does not include model fields and exceptional behaviors. The specifications are categorized using the keywords *requires* (for pre-conditions), *ensures* (for post-conditions), and *invariant* (for class invariants). In addition to user-provided specifications, we check null-pointer dereferencing and array index out of bounds exceptions.

By default, in JML, reference fields and the elements of an array field are assumed to be non-null. The *nullable* modifier is used to allow null values. We support arbitrarily nested universal and existential quantifiers in the specifications. Furthermore, the JML reachability construct is allowed. The expression *\reach(x, T, f)* gives the smallest set of objects of type T that are reachable from the object x via the field f. If x is of type T, it will be included in the set.

C. Target Logic

We translate programs and their specifications to a first-order SMT logic that contains only bounded types. It consists of quantified bit-vectors, arrays over bit-vectors, and uninterpreted functions. Quantified bit-vector formulas (QBVF) were traditionally handled by flattening quantifiers using conjunctions and disjunctions. This could result in losing some high-level information, and generating formulas that were too big to solve. Recent QBVF solvers [39],

[1]This may be done using the Soot framework [32].

however, perform several high-level simplifications such as heuristic-quantifier instantiation [26], miniscoping [16], and rewriting [21] before flattening quantifiers. Thus, the formulas handed to the underlying theory and SAT solvers are significantly more compact and easier to solve. Here, we describe our target logic in the SMT-LIB 2.0 syntax [31] in which expressions are given in a prefix notation.

Basics. We target a many-sorted first-order logic. The command `(declare-fun f (`A_1 `..` A_{n-1}`)` A_n`)` declares a function $f : A_1 \times .. \times A_{n-1} \rightarrow A_n$. Constants are functions that take no arguments. The command `(assert F)` asserts a formula F in the current logical context. Basic formulas are combined using the boolean operators `and`, `or`, `not`, and `=>` (implies). Universal and existential quantifiers are denoted by the keywords `forall` and `exists`.

Fixed-Size Bit-Vectors. All sorts in this theory are of the form `(_ BitVec m)` where m is a non-negative integer number, denoting the size of the bit-vector. Bit-vectors of different sizes represent different sorts in SMT. A constant number n represented by a bit-vector of size m is denoted by `(_ bvn m)`. This theory models the precise semantics of unsigned and of signed two-complements arithmetic. It supports a large number of logical and arithmetic operations on bit vectors. Examples include `bvule` (unsigned less than or equal to), `bvuge` (unsigned greater than or equal to), and `bvadd` (addition).

Extensional Arrays. An array sort is defined as `(Array` S_1 S_2`)` where S_1 and S_2 denote the index sort and the value sort, respectively. This theory supports two basic functions:

$$select : (Array\ S_1\, S_2) \times S_1 \rightarrow S_2 \tag{1a}$$

$$store : (Array\, S_1\, S_2) \times S_1 \times S_2 \rightarrow (Array\ S_1\ S_2) \tag{1b}$$

`(select a x)` returns the value corresponding to an index x of an array a, thus providing a read access. `(store a x y)` returns a copy of an array a in which the index x is mapped to the value y, thus providing a write access. Three properties are guaranteed: (1) updating the value of an index does not update the values of other indices, (2) reading an index yields the last value written in that index, (3) two arrays are identical iff all of their elements are identical.

In our approach, all basic types (including the index and value types of arrays) are fixed-size bit-vectors. Thus despite the arbitrary use of quantifiers, the logic is decidable.

III. Approach

A (preprocessed) method m is translated to an SMT formula based on a set of user-provided bounds b on the size of each type. We use T to denote this translation. That is, $T[m, b]$ produces a tuple (s, s', f, ex) in which s denotes the symbolic pre-state used in the translation, s' denotes the resulting post-state, f denotes an SMT formula whose satisfying solutions represent normal executions of m, and ex denotes an SMT formula whose satisfying solutions represent those executions of m that cause a built-in exception. Furthermore, we use R to denote the translation of specifications. That is, for a JML formula j and a symbolic state s, $R[j, s]$ produces an SMT formula whose satisfying solutions represent those concretizations of s that satisfy j.

Given a set of bounds b, a requires clause req and an ensures clause en for a method m defined in a class with an invariant clause inv, we use T and R to produce the following formula where $T[m, b] = (s, s', f, ex)$:

$$R[req, s] \land R[inv, s] \land f \land (R[\neg en, s'] \lor R[\neg inv, s'] \lor ex)$$

A solution to this formula represents a counter-example to the specification: a pre-state that satisfies req and inv, but its post-state either violates en or inv, or causes an exception.

The translation starts in a pre-state in which all the fields and all the inputs of the analyzed method are mapped to uninterpreted, symbolic constants. This section describes how various program and specification constructs are translated.

A. Encoding Control Flow

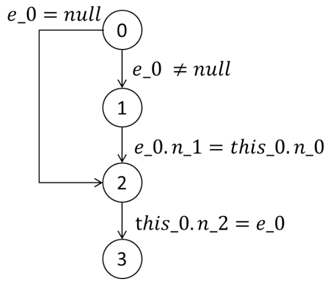
We encode the control flow of the analyzed method using a *computation graph* [36]. The nodes of this graph represent control points in the program, and the edges are either program statements or branch conditions. The graph has a single entry node and a single exit node, and is acyclic due to loop unrolling. All variables and fields are assumed to be renamed so that they are assigned at most once along each path of the graph. Explicit frame conditions are used to avoid underspecification at merge nodes.

Compared to a global-state encoding, the computation graph represents a program state implicitly, as a collection of independent variables and fields. That is, each update to a variable (field) replicates that variable (field) only, without causing the whole global state of the program to be replicated. Furthermore, using computation graphs allows us to encode the control- and the data-flow constraints separately, which prevents deeply-nested formulas and helps produce more readable counterexamples. More details about computation graphs can be found in [36].

Figure 2 provides an example. Suppose that the method *insert* in the *Entry* class of Figure 2(a) is selected for analysis. The computation graph of Figure 2(b) is constructed, assuming that the initial variables and fields are named using the index 0. The index is incremented every time the variable or the field is updated. Figure 2(c) gives the SMT formulas encoding the control flow: we introduce a boolean variable E_i_j to represent an edge from a node i to a node j. If an edge E_i_j is traversed and j has some outgoing edges, at least one of those outgoing edges must be traversed too. Furthermore, at least one of the entry edges must be traversed. The control constraints alone do not prevent infeasible paths (e.g. when both edges of a branch are taken); data flow constraints are also needed for the right semantics. Figure 2(d) gives the frame condition associated

```
class Entry {
 Entry n;
 int d;
 void insert(Entry e){
  if (e != null)
   e.n = this.n;
  this.n = e;
 }
}
```

(a)



(b)

```
(assert (and
 (=> E_0_1 E_1_2)
 (=> E_1_2 E_2_3)
 (=> E_0_2 E_2_3)
 (or E_0_1 E_0_2)))
```

(c)

```
(assert
 (=> E_0_2 (= n_1 n_0)))
```

(d)

```
(assert (and
 (=> E_0_1 (not (= e_0 nullEntry)))
 (=> E_0_2 (= e_0 nullEntry))
 (=> E_1_2 (= n_1(store n_0 e_0
                    (select n_0 this_0))))
 (=> E_2_3 (= n_2 (store n_1 this_0 e_0)))))
```

(e)

Figure 2: Control Flow Encoding: (a) a sample Java class, (b) computation graph, (c) control constraints, (d) frame conditions, (e) data constraints.

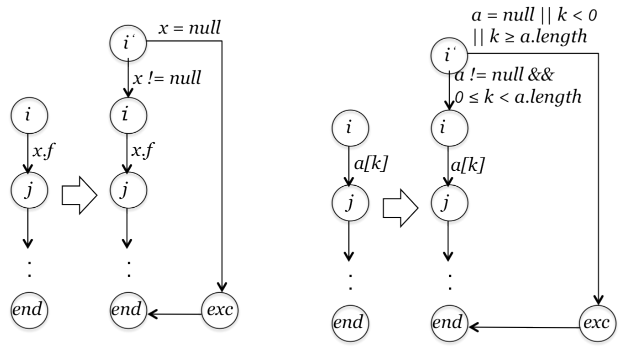
with the merge node 2. Since taking the edge *E_0_2* leaves the field *n_1* underspecified, the frame condition explicitly specifies *n_1 = n_0* along this edge. Figure 2(e) gives the data flow constraints as described in the next section.

Runtime Exception Checking: We check the two runtime exceptions of null pointer dereferencing and array index out of bounds by augmenting the computation graph with additional tests as shown in Figure 3. To check that a field dereference *x.f* at an edge *E_i_j* does not cause a null-pointer exception, we add an extra node i' that has an edge to i for the normal execution ($x \neq null$), and an edge to a unique exception node exc for the exceptional execution ($x = null$). The exc node has one outgoing edge to the final node. For an array access *a[k]*, we check that (1) a is not null, and (2) k is within the legal range. In the example of Figure 2, these checks amount to *(e_0 $\neq$ null) && (this_0 $\neq$ null)* before node 1, and *(this_0 $\neq$ null)* before node 2, both of which can be optimized out.

As a built-in property, we check if the edge from exc to the final node can be taken. To ensure valid counterexamples, an extra constraint specifies that only one of the incoming edges of the final node can be taken.

B. Encoding Data Flow

1) Primitive Types: The primitive *boolean* type is encoded as a 1-bit bitvector, and the *int* type is encoded as `(_ BitVec i)` where i is a user-provided bitwidth. All operations on integers are translated to their corresponding



(a) (b)

Figure 3: Exception handling: (a) null pointer dereferencing, (b) array index out of bounds.

bit-vector operations in SMT. Division and multiplication are computed using arithmetic shift operators.

2) Classes: Encoding the complete Java class hierarchy requires defining an *Object* class as the super-type of all classes. This imposes a significant overhead on the solver. Therefore, we avoid modeling the Object class unless the analyzed code actually reaches it or one of its methods. Thus, we cannot encode *null* as a single value that is compatible with all classes. In the absence of the Object class, any class that does not syntactically inherit a parent is considered a top-level class and has a distinct null value.

A top-level class A is encoded as a bit-vector of size $m = \lceil log(n+1) \rceil$ where n is the user-provided bound on the size of A. The value 0 stands for the null value for type A, denoted by $nullA$. We use an uninterpreted constant *idxA* to denote the last object of type A already allocated in the pre-state: `(declare-fun idxA () (_ BitVec m))`. The additional constraint `(bvule idxA n)` ensures that the bound constraint is met in the pre-state.

$$\underbrace{nullA, 1, 2, \ldots, idxA}_{\texttt{valid range of A}}, .., n, .., 2^m - 1$$

An object of type A is *valid* if it belongs to A's *valid range* of $[0, .., idxA]$. We explicitly constrain all objects of the pre-state (e.g. the receiver object and the method's arguments) to be valid. The receiver object is further constrained to be non-null:

```
(declare-fun this () (_ BitVec m))
(assert (and (not (= this nullA))
             (bvule this idxA)))
```

3) Class Hierarchies: We support class inheritance for concrete classes. We encode a superclass and its subclasses as the same SMT sort, and use additional constraints to maintain subtype semantics. Consider a class hierarchy where A and B extend C. Given user-provided bounds m, n, and w for A, B, and C, respectively (with an additional constraint $m + n \leq w$), we encode all three types as bit-vectors of size $s = \lceil log(w+1) \rceil$. This allows us to treat instances of a subclass as an instance for the superclass.

The null values of all subtypes and their supertype are the same and is represented by the value 0. An uninterpreted idx constant is used to represent the last allocated object of each type. Instances of subclasses are represented by the values of non-overlapping sub-ranges. In our example, the allocated objects of types A, B, and C are given by the subranges $[1, .., idxA]$, $[m+1, .., idxB]$, and $[m+n+1, .., idxC]$, respectively, where idx constants are constrained as follows:

```
(declare-fun idxA () (_ BitVec s))
(declare-fun idxB () (_ BitVec s))
(declare-fun idxC () (_ BitVec s))
(assert (and (bvule idxA m)
 (or (= idxB (_ bv0 s))
  (and(bvule idxB(bvadd m n))(bvugt idx_B m)))
 (or (= idxC (_ bv0 s))
  (and(bvule idxC w)(bvugt idxC(bvadd m n))))))
```

The valid ranges of A and B are $[0, .., idxA]$ and $[0, m+1, .., idxB]$, respectively. The valid range of C also includes the valid ranges for A and B, and is defined as $[0, .., idxA, m+1, .., idxB, m+n+1, .., idxC]$.

The Java expression *(o instanceof T)* evaluates to true if o is not null, and is in the valid range of T. Casting an object o to a class T is allowed if $o = null$ or *(o instanceof T)* holds. Overridden methods and fields are resolved via a sequence of nested tests on the actual object type using *instanceof*.

4) Object Allocation: In order to allocate an object of type A, we increment[2] $idxA$. The new value of $idxA$ denotes the allocated object. More precisely, the statement *A a = new A* is encoded by the following constraints where $idxA_i$ and $idxA_{i+1}$ represent the value of $idxA$ before and after the allocation, respectively, and m represents the number of bits for A.

```
(assert (and
 (= idxA_{i+1} (bvadd idxA_i (_ bv1 m)))
 (= a idxA_{i+1})
 (bvuge idxA_{i+1} idxA_i)
 (bvuge idxA_{i+1} (_ bv1 m))))
```

The last two constraints ensure that the expression $idxA_i + 1$ does not overflow by constraining the result to be greater than both $idxA_i$ and 1. Furthermore, the final number of allocated objects in the post-state, denoted by $idxA'$, will be constrained not to exceed A's bound.

5) Fields: Fields are translated to arrays over bit-vectors. Read accesses to the fields are encoded using the `select` operator, and updates are encoded using the `store` operator. Each update to a field requires a new array to represent the result. A field f of type B declared in a class A is encoded as `(declare-fun f () (Array (_ BitVec m) (_ BitVec n)))` where m and n denote the number of bits for A and B, respectively.

To ensure that the initial value of the field f is valid, we constrain that f maps all values of the valid range of A (except null) to a value in the valid range of B in the pre-state. For simple ranges of $[0, .., idxA]$ and $[0, .., idxB]$ for A and B, respectively, the following constraint is used:

```
(assert (forall (x (_ BitVec m))
 (=> (and (not (= x nullA)) (bvule x idxA))
     (bvule (select f x) idxB))))
```

Since each field update either uses an object from the pre-state or one that is allocated, the above constraint along with the constraints on allocated objects (see Section III-B4) ensure that all field updates throughout the code are valid.

6) Arrays: In Java, each array is an object that holds its contents. We encode array objects of type *A[]* by introducing a new type $ArrayObjA$ as `(_ BitVec t)`. Furthermore, we introduce $RefA$ as a reference from array objects of type *A[]* to their contents:

```
(declare-fun RefA ()
 (Array (_ BitVec t)
        (Array (_ BitVec i) (_ BitVec m))))
```

where t, i, and m denote the number of bits for $ArrayObjA$, integers, and the class A, respectively.

The length of an array in Java is initialized upon its allocation, and remains unchanged over its lifetime. Thus, it can be efficiently modeled using an uninterpreted SMT function from array objects to integers. The length of any array of type *A[]* is given by `(declare-fun LenA (_ BitVec t) (_ BitVec i))` where t and i denote the number of bits for $ArrayObjA$ and integers, respectively.

Array Allocation: Upon array allocation, its length is determined and its elements are initialized to 0 for primitive types, and null for others. Similar to other types, allocated array objects are modeled using an idx counter. The following constraint initializes the array attributes for an allocation statement *A[] ao = new A[length]*, where i is the number of bits for integers and $RefA_k$ represents the array reference before this statement.

```
(assert (and (= (LenA ao) length)
 (forall (x (_BitVec i))
  (=> (bvult x length)
  (= (select (select RefA_k ao) x) nullA)))))
```

Array Access: As shown above, a read access to an array object requires two nested `select` operations. Similarly, a write access requires nested `store` operations. That is, a Java statement *ao[j] = a* is encoded as follows, where $RefA_k$ and $RefA_{k+1}$ represent the array references before and after this statement, respectively.

```
(assert (let (RA (select RefA_k ao))
(= RefA_{k+1}(store RefA_k ao (store RA j a)))))
```

C. Encoding JML Specifications

JML specifications can refer to fields, the receiver object, the input arguments, and the return value that are accessible in the pre- or post-state of the analyzed method. We replace these references with the appropriate symbolic constants that are used in our translation.

[2]This is slightly more involved for the case of inheritance.

1) Valid instances: Encoding types as bit-vectors has the side effect that the accessible range of a type can be bigger than its valid range. Therefore, our translation modifies every JML formula that uses a variable x of type A to incorporate an additional constraint that x belongs to the valid range of A. This becomes particularly important for the variables used in quantification.

2) Reachability: Transitive closure (the reachability operator) over arbitrary domains cannot be axiomatized in pure first-order logic [22]. For finite domains, however, an axiomatization has been given by Claessen [6]. Inspired by his approach, we introduce the following axioms to compute the transitive closure of a (homogeneous) field f of type A declared in a class A. For a more concise syntax, we pretend that f is a function of type $A \rightarrow A$.

```
(assert (forall ((x A)(y A))
 (= (= (f x) y) (= (P x y) 1))))
(assert (forall ((x A)(y A)(z A))
 (=>(and (>(P x y)0)(>(P y z)0))(>(P x z)0))))
(assert (forall ((x A)(y A))
 (=>(> (P x y) 1) (exists (w A) (and
  (= (P x w) 1) (= (P x y) (+ 1 (P w y))))))))
```

The auxiliary function $P : A \times A \rightarrow int$ is defined to represent the smallest number of steps required to reach from one object to another (via f). The first constraint sets $P(x, y)$ to 1 if and only if $x.f = y$. The second constraint ensures transitivity, and the third constraint defines a partial order over all the objects reachable from a single source. This constraint is crucial for soundness when f is cyclic. Converting the axioms to use arrays and bit-vectors is straightforward. It should be noted that the maximum value of P is at most the number of objects in A. Therefore, in converting to bit-vectors, the int type used in the declaration of P above can use the same number of bits as A. Furthermore, the addition operator must be constrained not to overflow.

We use P to rewrite the reachability construct of JML. That is, an object y is in the set $\backslash reach(x, T, f)$ iff it is a valid object of type T, and $(x = y)$ *or* $((P\ x\ y) > 0)$.

3) Call Site Specifications: If the user provides pre- and post-conditions for a method *foo*, they will be used to substitute any call to *foo*. Otherwise, the body of *foo* will be inlined using the arguments of each call site. In order to substitute specifications, *foo*'s pre-conditions will be asserted (checked) to hold in the pre-state of the call, and post-conditions will be assumed to hold in a fresh post-state in which a fresh SMT constant is introduced for the return value and any field that may be modified by *foo*, and a fresh idx constant is introduced for any type of which *foo* may allocate an object. These constants will be constrained further to belong to the valid ranges of their corresponding types as described before.

IV. Evaluation

We have implemented our technique in a prototype tool, InspectJ, that uses the Jimple 3-address intermediate representation provided by the Soot optimization framework [32] to preprocess Java Bytecode, the Common JML Tools package (ISU) [18] to preprocess JML specifications, and Z3 [10] as the underlying SMT solver. We have checked a large-scale implementation of Dijkstra's shortest path algorithm that forms the basis of several optimized routing algorithms in graphs with millions of nodes. Furthermore, we have compared the runtime of InspectJ against JForge [12], a well-known SAT-based bounded verification tool that can handle data-structure-rich Java programs.

A. An implementation of Dijkstra's Shortest Path Algorithm

Dijkstra's Shortest Path Algorithm computes single-source shortest paths in graphs with non-negative edge weights. The optimized version [11] that we target makes heavy use of a priority queue backed by a binary heap. The original codebase is written in C++, which the second author has manually ported to Java for another project. Porting required mostly syntactic conversions but also two significant changes:

1) All C++ templates were removed and the actual types were substituted.
2) All occurrences of *STL:vector* in the C++ code were replaced with arrays, as our prototype does not handle Java library (such as *List*) at the moment.

The resulting Java code consists of 7 classes with a total of 37 methods and 346 Java source lines, excluding whitespace and specifications. To our knowledge, all previous verifications of the Dijkstra algorithm were performed on either a very abstract, or a very basic implementation [20, 24]. Our target code optimizes the memory layout and cache effects through sophisticated interconnections of data structures.

B. Formal Specifications

In order to check a method *foo* in a class C, formal specifications of *foo* are needed. Unlike full verification tools, our tool does not need specifications for *foo*'s called methods or loop invariants.

In the absence of any formal specifications, we provided JML specifications for the Dijkstra code by consulting its developers. The specifications of the analyzed methods consist of 27 lines and mostly constrain the internal integrity of the binary heap data structure. A sketch of the binary heap along with some of its invariants is shown in Figure 4.

Since JForge expects specifications in the JFSL language [40], we also converted JML specifications to JFSL. This conversion required only simple syntactic changes.

C. Detected Bugs

So far we have checked 10 out of a total of 19 public methods. Our analysis revealed 3 previously-unknown bugs in the Java implementation of the binary heap data structure, two of which represented the same problem in two different

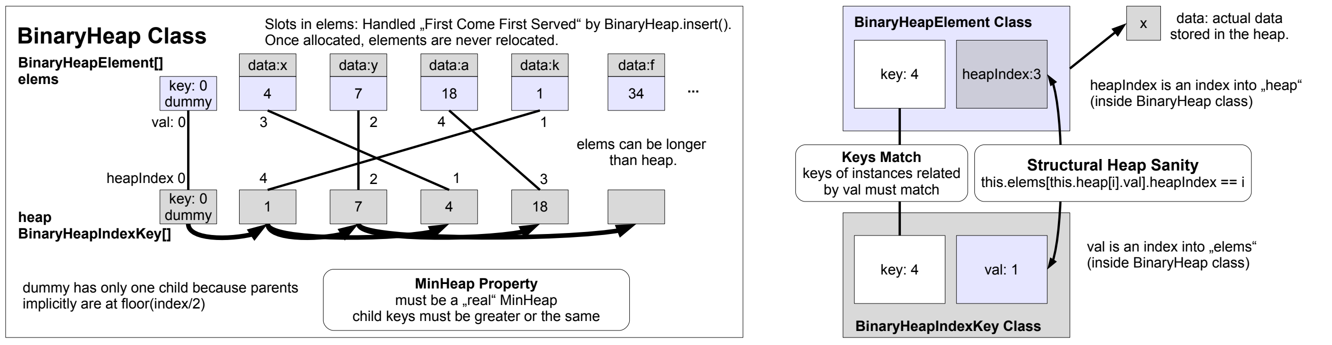

BinaryHeap Class
Slots in elems: Handled „First Come First Served“ by BinaryHeap.insert(). Once allocated, elements are never relocated.
BinaryHeapElement[] elems
key: 0 dummy
data:x
4
data:y
7
data:a
18
data:k
1
data:f
34
...
val: 0
3
2
4
1
elems can be longer than heap.
heapIndex 0
4
2
1
3
key: 0 dummy
1
7
4
18
heap BinaryHeapIndexKey[]
dummy has only one child because parents implicitly are at floor(index/2)
MinHeap Property
must be a „real“ MinHeap
child keys must be greater or the same
BinaryHeapElement Class
key: 4
heapIndex:3
x
data: actual data stored in the heap.
heapIndex is an index into „heap“ (inside BinaryHeap class)
Keys Match
keys of instances related by val must match
Structural Heap Sanity
this.elems[this.heap[i].val].heapIndex == i
key: 4
val: 1
val is an index into „elems“ (inside BinaryHeap class)
BinaryHeapIndexKey Class

Figure 4: A sketch of the binary heap data structure and a sampling of its invariants.

methods. All bugs required small bounds (max. 3) and could be found in under two seconds by both InspectJ and JForge.

The first problem (repeated twice) was introduced when the C++ code was ported to Java. Assigning a *struct* in C++ copies all of its fields by value, while assigning an object in Java only copies it by reference. Mimicking the C++ code, we used a simple assignment to get copies of the elements of the binary heap, which caused inconsistencies. The bug and its fix are shown in Listing 1. The JML specification that caught the bug is also given. It constrains the keys of the *elems* array to match the keys of the *heap* array. In C++, line 8 modifies a fresh copy of *heap[index1]*, whereas in Java, it modifies the same object (*key* and *val* are integer values). This unintended modification to *heap* causes inconsistencies between *heap* and *elems* which causes the specification to fail.

```
1  /*@ invariant
2  @(\forall int i; i >= 0 && i < this.heap.len
3  @ ==> this.elems[this.heap[i].val].key ==
4  @       this.heap[i].key)
5  @*/
6  // VERSION WITH BUG
7  heap[index2] = heap[index1];
8  heap[index2].key = k;
9
10 // VERSION WITHOUT BUG
11 heap[index2].key = heap[index1].key;
12 heap[index2].val = heap[index1].val;
13 heap[index2].key = k;
```

Listing 1: Invalid Copy Semantics

The second problem was present in the original C++ code that was already heavily tested. It involved a memory location that was freed, but under certain conditions, was referenced again. InspectJ detected this as a null pointer exception since our Java code marks freed locations by null. The bug and its fix are shown in Listing 2. Line 2 removes the last element of the heap, but Line 3 accesses the first element without checking whether the heap has become empty (*heap[0]* is a dummy element). Swapping Lines 2 and 3 produces the intended behavior and fixes the bug in this case. We have already reported the problem and it has been fixed by the original developers.

```
1  // VERSION WITH BUG
2  this.dropHeap();
3  x = heap[1];
4  ....
5
6  // VERSION WITHOUT BUG
7  x = heap[1];
8  this.dropHeap();
9  ....
```

Listing 2: Invalid Memory Access

D. Runtime Evaluation

In order to evaluate the performance of our SMT-based technique, we compared the runtime of InspectJ with JForge. Apart from different specification languages, both JForge and InspectJ operate on the same inputs. To evaluate scalability, we set both tools to inline called methods in all the experiments, and increased the bounds until both tools timed out. All the experiments are performed on an Intel Xeon 2.53 GHz with 16 GB of RAM using OpenSUSE 11.3 64bit. We used JForge 0.2.6, SAT4J 2.3.0.v20110329, and Z3 3.2.

The evaluation results are given in Table I. The *Bits*, *Objs*, and *Loop* columns give the bounds on the integer bitwidth, objects of each class, and loop iterations, respectively. The runtime of each tool is given in seconds, and is split into the time spent in the preprocessing phase (denoted by *PrePro*), in which the code and its specifications are translated to SAT or SMT formulas, and in the solving phase, which is performed by the underlying solver. By default, JForge uses an old version of SAT4J which we replaced with its most recent version. Furthermore, for a fair comparison, we also used Z3 (in SAT solving mode) as the backend solver for JForge. The *Total* column gives the sum of the preprocessing time and the best solving time. Any runtime beyond our threshold of 600 seconds is denoted by *TO*.

The methods *insert*, *decreaseKey*, and *deleteMin* provide the main functionality of the binary heap. They are used to insert, update, and delete heap elements, respectively. The method *minElement* returns the root element of the heap. Checking these methods returns *unsat*, meaning that the solver cannot find a counterexample, and thus the specifications hold for the given bounds. All the bugs previously described were fixed prior to this comparison. We also experimented with some satisfiable cases by underspecifying the *run* method, the functional entry point of the Dijkstra code. Calling this method typically involves calling all the above methods multiple times.

As shown in Table I, when checking some methods with respect to very small bounds (see first rows for *deleteMin*, *insert*, and *minElement*), InspectJ is slower than JForge. This is caused by the slow startup of the Soot and ISU libraries used in InspectJ preprocessing. However, in all other cases, InspectJ is significantly faster, and is capable of checking bounds that JForge cannot. An interesting case is *minElement* for which the runtime of InspectJ is independent of the bounds. This is because the SMT solver can deduce unsatisfiablity of the formula using high-level simplifications. Increasing the bounds beyond 10 in this case, causes an internal error in JForge due to the big sizes of the formulas.

As shown in the table, InspectJ preprocessing is independent of the analyzed bounds. This is because it only makes a few passes over the input program and specification to generate the SMT formulas. JForge preprocessing, however, generates boolean formulas through bit-blasting. It incorporates low-level optimizations such as symmetry breaking and sharing detection, and thus depends on the analyzed bounds.

During the experiments, we noticed that InspectJ covers some paths in the code that JForge does not. For example, *deleteMin* removes the root of the heap and restructures the resulting tree to remain balanced. Covering all distinct cases requires at least 5 tree nodes, but JForge times out in that scope, and thus cannot analyze certain paths of the code.

Although more scalable than JForge, InspectJ still does not deliver required scalability in all cases. The *run* method, for example, cannot be checked beyond 2 loop iterations due to its complexity (nested loops with various method calls). We are currently investigating some abstraction ideas to further improve the scalability of our approach.

V. Related Work

Many bounded verification approaches (e.g. Jalloy [36], JForge [12], TACO[27], Miniatur[13], Karun[33], and MemSAT[34]) have been developed that target data-structure-rich programs. Similar to our technique, these approaches are exhaustive in the analyzed domain and produce non-spurious counterexamples (wrt. the analyzed bounds). However, unlike our technique that translates the code and specifications to an SMT logic to allow high-level simplifications, they use propositional logic (via a relational logic) with only local simplifications at boolean level. Thus, scalability is their key issue due to bit-blasting problems.

Scalability of bounded program verification can be improved by partitioning the set of all program executions based on the program's control-flow or data-flow properties, and analyzing each partition separately [28, 29]. Another possibility is to introduce a CEGAR framework (see e.g. Karun [33]), to iteratively analyze only the necessary parts of the code. Such ideas are independent of the underlying solver and can be incorporated into our approach in future.

ESC/Java[14] and ESC/Java2[8] analyze JML specifications of Java programs where loops are bounded, but Java classes are not. They support various SMT solvers and theorem provers, but due to quantification over infinite types, their target logics are undecidable. Thus the solver may not terminate with a conclusive outcome.

TestEra [25] and Korat [5] also check Java programs against data-structure properties with respect to a bounded heap. However, they perform the check dynamically. That is, they generate all nonisomorphic input structures that satisfy the pre-conditions within the given bounds, run the program on each input, and check the results against an oracle (or a post-condition). For checking code that involves a single data structure (e.g. a linked list or a tree), these approaches would suffice; they would achieve the same results as bounded program verification. However, for checking code that involves several data structures, the number of possible inputs can become too large to enumerate and execute explicitly.

Model checkers such as FSoft [17], CBMC[7], and SLAM [2] focus on checking temporal safety properties, provide a fully automatics analysis, and can produce sound counterexamples. However, they require the entire program; no modular analysis is supported. They have been successfully used in checking large programs against control properties, but they are not suitable for checking the kind of data-structure properties that we aim.

Several model checkers (e.g. [1, 9, 15, 30, 38]) incorporate SMT solvers as their underlying engines. Similar to our approach, they translate a program and its property to an SMT logic that consist of bitvectors and/or arrays. Unlike our approach, their logics are quantifier-free. To our knowledge, all of these model checkers target C programs (thus no object-oriented features are supported), and their translations are highly tuned for checking memory layout and finite-state-machine properties; no data-structure properties (beyond simple array accesses) can be checked.

Java PathFinder [37], model checks Java programs by explicitly traversing their state spaces. Originally it only checked temporal safety properties, but then it was integrated with Korat to handle data structure properties [19]. Similar to Korat, explicitly enumerating all (nonisomorphic) inputs can limit the applicability of this approach when the input consists of several data structures or is weekly constrained.

KeY [4] and LOOP [35] use theorem provers to verify

Table I: Evaluation Results

Method	Bits	Objs	Loop	JForge					InspectJ			
				PrePro	SAT4J	Z3	Total	Result	Result	PrePro	Z3	Total
decreaseKey	3	3	3	0.6	TO	61.8	62.4	unsat	unsat	1.5	0.4	1.9
	4	4	4	0.7	TO	82.5	83.2	unsat	unsat	1.5	8.7	10.3
	5	5	5	1.8	TO	TO	TO	-	unsat	1.5	31.3	32.8
	6	6	6	8.7	TO	TO	TO	-	unsat	1.5	117.1	118.6
	7	7	5	63.5	TO	TO	TO	-	unsat	1.5	357.6	359.1
	7	7	6	66.0	TO	TO	TO	-	unsat	1.6	507.5	509.1
deleteMin	3	3	3	0.5	3.4	0.6	1.1	unsat	unsat	1.7	0.2	1.9
	4	4	4	1.5	414.8	36.4	37.9	unsat	unsat	1.7	3.4	5.0
	5	5	5	4.8	TO	TO	TO	-	unsat	1.7	52.5	54.2
	6	6	6	29.5	TO	TO	TO	-	unsat	1.7	133.4	135.1
insert	3	3	3	0.5	1.6	0.5	1.0	unsat	unsat	1.6	0.4	1.9
	4	4	4	0.8	69.8	14.8	15.6	unsat	unsat	1.6	5.4	7.0
	5	5	5	2.1	TO	409.8	411.9	unsat	unsat	1.6	86.8	88.4
	6	6	6	11.3	TO	TO	TO	-	unsat	1.6	110.0	111.6
	7	7	6	71.2	TO	TO	TO	-	unsat	1.6	311.4	313.0
minElement	4	4	4	0.5	0.3	0.2	0.7	unsat	unsat	1.4	0.0	1.4
	6	6	6	6.4	19.5	6.9	13.3	unsat	unsat	1.4	0.0	1.4
	7	7	7	49.5	70.0	16.6	66.1	unsat	unsat	1.4	0.0	1.4
	8	8	8	TO	-	-	TO	-	unsat	1.4	0.0	1.4
	10	10	10	TO	-	-	TO	-	unsat	1.4	0.0	1.4
	11	11	11	FAIL	-	-	-	-	unsat	1.4	0.0	1.4
run	3	3	1	9.6	1.5	2.2	11.8	sat	sat	3.2	0.7	3.9
	4	4	1	16.7	9.5	4.3	21.0	sat	sat	3.2	6.9	10.0
	7	7	1	371.1	TO	299.0	TO	-	sat	3.2	0.2	3.4
	10	10	1	TO	-	-	TO	-	sat	3.2	2.4	5.6
	3	3	2	TO	-	-	TO	-	sat	5.0	52.7	57.7

rich properties of Java programs in a modular way, without bounding domains. However, they often need user interaction to prove their generated proof obligations. Boogie [3], on the other hand, uses the Z3 SMT solver to fully verify Java programs. Although Z3 is fully automatic, since Boogie targets an undecidable SMT logic, it does not always terminate with a conclusive result. Furthermore, these approaches require the user to provide loop invariants. Shape analysis techniques also provide full verification. TVLA [23], for example, uses a 3-valued logic to analyze certain data structures such as singly- and doubly-linked lists. However, it is not easily extensible to arbitrary data structures.

VI. Conclusion

We presented an SMT-based, bounded verification technique for finding bugs in data-structure-rich programs. Programs are checked with respect to user-provided bounds on the number of loop iterations and the number of elements of each type. All found counterexamples are guaranteed to be sound in the analyzed domain and any counterexample within that domain is guaranteed to be found. Lack of a counterexample, however, does not constitute a proof of correctness beyond the analyzed domain. The novelty of the approach is to exploit the quantified bitvector theory (QBVF) of recent SMT solvers, which allows high-level simplifications. To our knowledge, this is the first attempt to use an SMT solver for bounded program verification.

We described how object-oriented features such as class hierarchies, fields, dynamic object allocations, and array objects can be efficiently encoded in QBVF. We also reported on applying our prototype tool, InspectJ, to a large-scale implementation of the Dijkstra's shortest path algorithm. The results were encouraging; we found 3 previously-unknown bugs, and witnessed significantly better scalability over JForge—a compatible SAT-based engine. Checking programs with respect to bigger bounds allowed us to cover some execution paths that JForge could not cover.

In addition to improving InspectJ to handle more of Java (e.g. exceptions and library methods), we will investigate incorporating several optimizations such as slicing, partitioning, and CEGAR (see Sec. V) to reduce the burden of the underlying solver. Furthermore, although we described a translation of reachability specifications to QBVF, this feature was not used in our case study. Therefore, the efficiency of this translation has to be evaluated in the future.

An important question in bounded verification is the relationship between the number of objects and loop unrollings. Increasing one without increasing the other is not always meaningful; it may only cause dead code or unused objects. We will investigate such relationships in the future.

Acknowledgement

We thank Emina Torlak for helping us with Kodkod, Greg Dennis and Kuat Yessenov for helping us with JForge, and Julian Dibbelt for explaining the Dijkstra code. This work was funded in part by DFG grant TA764/1-1.

References

[1] A. Armando, J. Mantovani, and L. Platania, "Bounded model checking of software using smt solvers instead

of sat solvers," *Int. J. Softw. Tools Technol. Transf.*, vol. 11, pp. 69–83, 2009.

[2] T. Ball and S. Rajamani, "Automatically validating temporal safety properties of interfaces," in *SPIN Workshop*, 2001, pp. 103–122.

[3] M. Barnett, B. Chang, R. Deline, and et.al., "Boogie: A modular reusable verifier for object-oriented programs," in *FMCO*, 2006, pp. 364–387.

[4] B. Beckert, R. Hähnle, and P. Schmitt, *Verification of object-oriented software: The KeY approach*. Springer-Verlag, 2007.

[5] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated testing based on java predicates," in *ISSTA*, 2002, pp. 123–133.

[6] K. Claessen, "Expressing transitive closure for finite domains in pure first-order logic," Chalmers University of Technology., Tech. Rep., 2008.

[7] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *TACAS*, 2004, pp. 168–176.

[8] D. R. Cok and J. R. Kiniry, "Esc/java2: Uniting esc/java and jml - progress and issues in building and using esc/java2," in *CASSIS*. SpringerVerlag, 2004.

[9] L. Cordeiro, B. Fischer, and J. Marques-Silva, "Smt-based bounded model checking for embedded ansi-c software," in *ASE*, 2009, pp. 137–148.

[10] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *TACAS*, 2008, pp. 337–340.

[11] D. Delling, "Engineering and augmenting route planning algorithms," Ph.D. dissertation, Universitat Karlsruhe (TH), 2009.

[12] G. Dennis, F. Chang, and D. Jackson, "Modular verification of code with sat," in *ISSTA*, 2006, pp. 109–120.

[13] J. Dolby, M. Vaziri, and F. Tip, "Finding bugs efficiently with a sat solver," in *FSE*, 2007, pp. 195–204.

[14] C. Flanagan, R. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata, "Extended static checking for java," in *PLDI*, 2002, pp. 234–245.

[15] M. Ganai and A. Gupta, "Accelerating high-level bounded model checking," in *ICCAD*, 2006, pp. 794–801.

[16] J. Harrison, *Handbook of Practical Logic and Automated Reasoning*, 1st ed. Cambridge University Press, 2009.

[17] F. Ivančić, Z. Yang, M. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar, "F-soft: Software verification platform," in *CAV*, 2005, pp. 301–306.

[18] "Jml reference manual." [Online]. Available: http://www.eecs.ucf.edu/ leavens/JML/

[19] S. Khurshid, C. Pasareanu, and W. Visser, "Generalized symbolic execution for model checking and testing," in *TACAS*, 2003, pp. 553–568.

[20] V. Klasen, "Verifying dijkstra's algorithm with key," in *Diploma Thesis, Universitat Koblenz-Landau*, 2010.

[21] D. Knuth and P. Bendix, "Simple word problems in universal algebra," in *Computational problems in abstract algebra*, 1970, pp. 263–297.

[22] T. Lev-ami, N. Immerman, T. Reps, M. Sagiv, and et.al., "Simulating reachability using first-order logic with applications to verification of linked data structures," in *In CADE-20*, 2005, pp. 99–115.

[23] T. Lev-Ami and M. Sagiv, "Tvla: A system for implementing static analyses," in *SAS*, 2000, pp. 280–301.

[24] R. Mange and J. Kuhn, "Verifying dijkstra algorithm in jahob," 2007, student project, EPFL.

[25] D. Marinov and S. Khurshid, "Testera: A novel framework for automated testing of java programs," in *ASE*, 2001, pp. 322–342.

[26] L. Moura and N. Bjørner, "Efficient e-matching for smt solvers," in *CADE*, 2007, pp. 183–198.

[27] B. Parrino, J. Galeotti, D. Garbervetsky, and M. Frias, "A dataflow analysis to improve sat-based bounded program verification," in *SEFM*, 2011, pp. 138–154.

[28] D. Shao, S. Khurshid, and D. Perry, "An incremental approach to scope-bounded checking using a lightweight formal method," in *FM*, 2009.

[29] D. Shao, D. Gopinath, S. Khurshid, and D. Perry, "Optimizing incremental scope-bounded checking with data-flow analysis," in *ISSRE*, 2010, pp. 408–417.

[30] C. Sinz, S. Falke, and F. Merz, "A precise memory model for low-level bounded model checking," in *SSV*, 2010.

[31] "Smt-lib: The satisfiability modulo theories library." [Online]. Available: http://www.smtlib.org/

[32] "Soot - a java bytecode optimization framework." [Online]. Available: http://www.sable.mcgill.ca/soot/

[33] M. Taghdiri, "Automating modular program verification by refining specifications," Ph.D. dissertation, Massachusetts Institute of Technology, 2008.

[34] E. Torlak, M. Vaziri, and J. Dolby, "Memsat: checking axiomatic specifications of memory models," in *PLDI*, 2010, pp. 341–350.

[35] J. van den Berg and B. Jacobs, "The loop compiler for java and jml," in *TACAS*, 2001, pp. 299–312.

[36] M. Vaziri, "Finding bugs in software with a constraint solver," Ph.D. dissertation, MIT, 2004.

[37] W. Visser, K. Havelund, G. Brat, S. Part, and F. Lerda, "Model checking programs," in *ASE*, 2000, pp. 3–11.

[38] M. Vujošević-Janičić and V. Kuncak, "Development and evaluation of LAV: an SMT-based error finding platform," in *VSTTE*, 2012.

[39] C. Wintersteiger, Y. Hamadi, and L. de Moura, "Efficiently solving quantified bit-vector formulas," in *FMCAD*, 2010, pp. 239–246.

[40] K. Yessenov, "A lightweight specification language for bounded program verification," in *MEng Thesis, MIT*, 2009.