# An Incremental Approach to Scope-Bounded Checking Using a Lightweight Formal Method

Danhua Shao, Sarfraz Khurshid, and Dewayne E. Perry

Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712, USA
{dshao,khurshid,perry}@ece.utexas.edu

**Abstract.** We present a novel approach to optimize *scope-bounded* checking programs using a relational constraint solver. Given a program and its correctness specification, the traditional approach translates a bounded code segment of the *entire* program into a declarative formula and uses a constraint solver to search for any correctness violations. Scalability is a key issue with such approaches since for non-trivial programs the formulas are complex and represent a heavy workload that can choke the solvers. Our insight is that bounded code segments, which can be viewed as a *set* of (possible) execution paths, naturally lend to *incremental* checking through a *partitioning* of the set, where each partition represents a sub-set of paths. The partitions can be checked independently, and thus the problem of scope-bounded checking for the given program reduces to several sub-problems, where each sub-problem requires the constraint solver to check a less complex formula, thereby likely reducing the solver's overall workload. Experimental results show that our approach provides significant speed-ups over the traditional approach.

**Keywords:** Scope-bounded checking, Alloy, first-order logic, SAT, lightweight formal method, computation graph, white-box testing.

## 1 Introduction

*Scope-bounded* checking [1, 4, 5, 9, 11, 17], i.e., systematic checking for a bounded state-space, using off-the-shelf solvers [7, 21, 24], is becoming an increasingly popular methodology to software verification. The state-space is typically bounded using bounds (that are iteratively relaxed) on input size [1], and length of execution paths [9].

While existing approaches that use off-the-shelf solvers have been used effectively for finding bugs, scalability remains a challenging problem. These approaches have a basic limitation: they require translating the bounded code segment of the *entire* program into *one* input formula for the solver, which solves the complete formula. Due to the inherent complexity of typical analyses, many times solvers do not terminate in a desired amount of time. When a solver times out, e.g., fails to find a counterexample, typically there is no information about the likely correctness of the program checked or the coverage of the analysis completed.

This paper takes a *divide-and-solve* approach, where smaller segments of bounded code are translated and analyzed—even if the encoding or analysis of some segments time out, other segments can still be analyzed to get useful results. Our insight is that bounded code segments, which can be viewed as a *set* of (possible) execution paths, naturally lend to *incremental* checking through a partitioning of the set, where each partition represents a sub-set of paths. The partitions can be checked independently, and thus the problem of scope-bounded checking for the given program reduces to several sub-problems, where each sub-problem requires the constraint solver to check a less complex formula, thereby likely reducing the solver's overall workload.

We develop our approach in the context of the Alloy tool-set [14]—a lightweight formal method—and the Java programming language. The Alloy specification language is a first-order logic based on sets and relations. The Alloy Analyzer [13] performs scope-bounded analysis for Alloy formulas using off-the-shelf SAT solvers.

Previous work [5, 6, 15, 29] developed translations for bounded execution fragments of Java code into Alloy's relational logic. Given a procedure *Proc* in Java and its pre-condition *Pre* and post-condition *Post* in Alloy, the following formula is solved [15, 29]:

$$Pre \wedge translate(Proc) \wedge \neg Post$$

Given bounds on loop unrolling (and recursion depth), the *translate()* function translates the bounded code fragments of procedure *Proc* from Java into a first order logic formula. Using bounds on the number of objects of each class, the conjunction of *translate(Proc)* with *Pre* and *Post* is translated into a propositional formula. Then, a SAT solver is used to search solutions for the formula. A solution to this formula corresponds to a path in *Proc* that satisfies *Pre* but violates *Post*, i.e., a counterexample to the correctness property.

In our view, the bounded execution fragment of a program that is checked represents a set of possible execution paths. Before translating the fragment into relational logic, our approach implicitly partitions the set of paths using a partitioning *strategy* (Section 4), which splits the given program into several *sub*-programs—each representing a smaller bounded execution fragment—such that

$$path(Proc) = \bigcup_{i=1}^{n} path(Sub_i)$$

Function *path(p)* represents the set of paths for a bounded execution segment *p*. $Sub_1$, …, $Sub_n$ are sub-programs corresponding to path partitioning. To check the procedure *Proc* against pre-condition *Pre* and post-condition *Post*, we translate bounded execution fragment of each sub-program into a first order logic formula and check correctness separately.

$$Pre \wedge translate(Proc) \wedge \neg Post \Leftrightarrow$$
$$\{Pre \wedge translate(Sub_1) \wedge \neg Post\} \wedge .... \wedge \{Pre \wedge translate(Sub_n) \wedge \neg Post\}$$

Thus, the problem of checking *Proc* is divided into sub-problems of checking smaller sub-programs, $Sub_1$, …, $Sub_n$. Since the control-flow in each sub-program is less complex than the entire procedure, we expect the sub-problems to represent easier SAT problems.

This paper makes the following contributions:

- *An incremental approach*. To check a program against specifications, we propose to divide the program into smaller sub-programs and check each of them individually with respect to the specification. Our approach uses path partitioning to reduce the workload to the backend constraint solver.
- *Implementation*. We implement our approach using the Forge framework [5] and KodKod model finder [28].
- *Evaluation*. Experiments using Java library code show that our approach can significantly reduce the checking time.

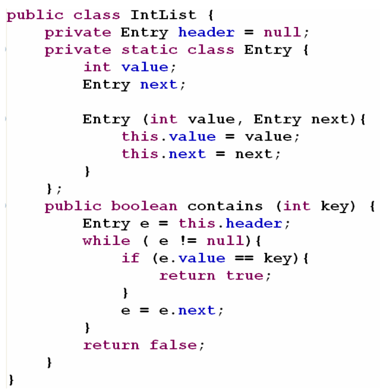
## 2 Example

This section presents a small example to illustrate our path partitioning and program splitting algorithm. Suppose we want to check the `contains()` method of class `IntList` in Figure 1 (a):

An object of `IntList` represents a singly-linked list. The `header` field points to the first node in the list. Objects of the inner class `Entry` represent list nodes. The `value` field represents the (primitive) integer data in a node. The *next* field points to the next node in the list. Figure 1 (b) shows an instance of `IntList`.

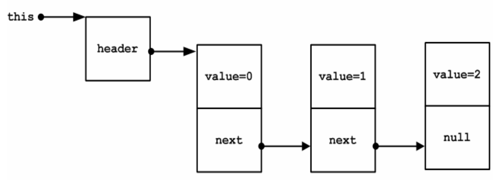
Consider checking the method `contains()`. Assume a bound on execution length of one loop unrolling. Figure 2 (a) shows the program and its *computation graph* [15] for this bound.

Our program splitting algorithm uses the computation graph and is *vertex-based*: Given a vertex in the computation graph, we split the graph into two sub-graphs—(1) *go-through* sub-graph and (2) *bypass* sub-graph. The go-through sub-graph has all the paths that go through the vertex and the bypass sub-graph has all the paths that bypass the vertex. Given the computation graph in Figure 2 (a), splitting based on vertex 11 generates the go-through sub-graph shown in Figure 2 (b) and the bypass sub-graph



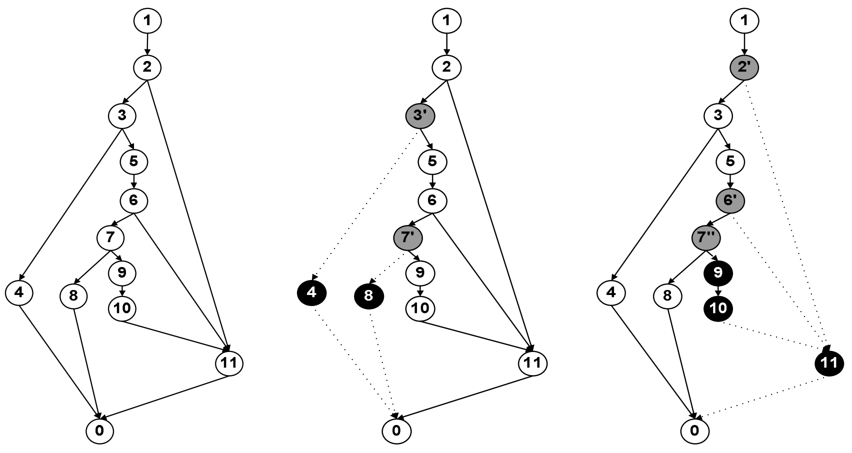
```
public class IntList {
    private Entry header = null;
    private static class Entry {
        int value;
        Entry next;

        Entry (int value, Entry next){
            this.value = value;
            this.next = next;
        }
    };
    public boolean contains (int key) {
        Entry e = this.header;
        while ( e != null){
            if (e.value == key){
                return true;
            }
            e = e.next;
        }
        return false;
    }
}
```

(a)



(b)

**Fig. 1.** Class `IntList` (`contains()` method and an instance)



```
public boolean
constains(int key)
 {
1 : Entry e = this.header;
2 : if (e != null){
3 :  if (e.value == key){
4 :   return true;
    }
5 :  e = e.next;
6 :  if (e != null){
7 :   if (e.value == key){
8 :    return true;
     }
9 :   e = e.next;
    }
10:  assume(e == null);
   }
11: return false;
0 :}
```

(a)

```
public boolean
go-through(int key)
 {
1 : Entry e = this.header;
2 : if (e != null){
3': assume !(e.value==key);
4 :

5 :  e = e.next;
6 :  if (e != null){
7':  assume!(e.value==key);
8 :

9 :   e = e.next;
    }
10:  assume(e == null);
   }
11: return false;
0 :}
```

(b)

```
public boolean
bypass(int key)
{
1 : Entry e = this.header;
2': assume(e != null);
3 : if (e.value == key){
4 :   return true;
  }
5 : e = e.next;
6': assume (e != null);
7": assume(e.value == key);
8 : return true;

9 :

10:

11:
0 :}
```

(c)

**Fig. 2.** Splitting of program `contains()` based on Vertex 11. *Broken lines* in sub-graph indicate edges removed constructing this sub-program during splitting. *Gray nodes* in a sub-graph denote that a branch statement in original program has been transformed into an assume statement. In programs below computation graph, the corresponding statements are show in Italic. *Black nodes* denote the statements removed during splitting.

shown in Figure 2 (c). Thus the problem of checking the contains method using a bound of one loop unrolling is solved using two calls to SAT based on the two computation sub-graphs.

## 3 Background

The goal of our computation graph splitting algorithm is to optimize traditional bounded exhaustive checking of programs using constraints in relational logic. The traditional approach [5] [15] [29] translates the entire bounded Java code segment into one relational logic formula. The conjunction of the code formula and the negation of correctness specifications are passed to a relational logic constraint solver. Solutions are translated back to executions that violate the specification.

## 3.1 Relational Logic and SAT-Based Analysis

Traditional approaches use a subset of Alloy [14] as the relational logic for Java code translation. Alloy is a first-order declarative language based on *relations*. A relation is a collection of tuples of atoms. A relation can have any finite arity. A set is viewed as a unary relation, and a scalar is a singleton set.

In Alloy, *expressions* are formed by relations combined with a variety of operators. The standard set operators are union (+), intersection (&), difference (-). Unary relational operators are transpose (~), transitive closure (^), and reflexive transitive closure (*), which have their standard interpretation. Binary relational operators are join (.), product (->), and update (++).

Expression quantifiers turn an expression into a *formula*. The formula 'no e' is true when e denotes a relation containing no tuples. Similarly, 'some e', and 'one e' are true when e has some, and exactly one tuple respectively. Formulas can also be made with relational comparison operators: subset (in), equality (=) and inequality (!=). So 'e1 in e2' is true when every tuple in (the relation denoted by the expression) e1 is also a tuple of e2. Alloy provides the standard logical operators: conjunction (&&), disjunction (||), implication ($\Rightarrow$), bi-implication ($\Leftrightarrow$), and negation (!).

A *model* of an Alloy formula is an assignment of relational variables to sets of tuples built from a universe of atoms within a given *scope*. So Alloy Analyzer is a model finder for Alloy formula.

The Alloy Analyzer uses a *scope*, i.e., a bound on the universe of discourse, to perform *scope-bounded* analysis: the analyzer translates the given Alloy formula into a propositional satisfiability (SAT) formula (w.r.t. the given scope) and uses off-the-shelf SAT technology to solve the formula.

## 3.2 Java to Relational Logic Translation

A relational view of the program heap [15] allows translation of a Java program into an Alloy formula using three steps: (1) encode data, (2) encode control-flow, and (3) encode data-flow.

Encoding data builds a representation for classes, types, and variables. Each class or type is represented as a set, *domain*, which represents the set of object of this class or values of this type. Local variables and arguments are encoded as singleton sets. A field of a class is encoded as a binary, functional relation that maps from the class to the type of the field. For example, to translate the program in Figure 2 (a), we define four domains: List, Entry, integer, and boolean. Field header is a partial function from List to Entry, and field next is a partial function from Entry to Entry. Field value is a function from Entry to integer.

Data-flow is encoded as relational operations on sets and relations. Within an expression in a Java statement, field deference is encoded as relational join, and an update to a field is encoded as relational override. For a branch statement, predicates on variables or expressions are encoded as corresponding formulas with relational expressions. Method calls are encoded as formulas that abstract behavior of the callee methods.

Given a program, encoding control-flow is based on computation graph. Each edge ($v_i \rightarrow v_j$) in the computation graph is represented as a boolean variable $E_{i,j}$. True value of edge variable means the edge is taken. The control flow from one statement to its sequential statement another is viewed as relational implication. For example, code segment `{A; B; C;}` is translated to '$E_{A,B} \Rightarrow E_{B,C}$'. Control flow splits at a *branching* statement—the two branch edges are viewed as a relational disjunction. For each branch edge, a relational formula is generated according to the predicate. Only data that satisfied the relational formula can take this edge. In Figure 2 (a), control flow at vertex 3 is translated into '($E_{2,3} \Rightarrow E_{3,4}$ || $E_{3,5}$) and ($E_{3,4} \Rightarrow$ e.value = key) and ($E_{3,5} \Rightarrow$ !(e.value = key))'. If the control flow takes `then` branch $E_{3,4}$, the constraint '($E_{3,4} \Rightarrow$ e.value = key)' should be satisfied. An `assume` statement is translated into the formula for its predicate. For example, at vertex 10 of Figure 2 (a), the control-flow is encoded as '($E_{10,3} \Rightarrow$ no e)'. This constraint restricts that this edge is taken only when `e` is `null`.

In our splitting algorithm, sub-graphs are constructed by removing branch-edges at selected branch statements. According to the translation scheme, a branch statement is equivalent to two assume statements with complementary predicates. So removing one branch can be implemented as transforming the branch statement into an assume statement. In Figure 2 (a), removing the `then` branch of vertex 3, branch statement `'if(e.value == key)'` will be transformed to 'assume !(e.value == key)'. Its relational logic translation is '($E_{2,3} \Rightarrow E_{3,4}$) and ($E_{3,5} \Rightarrow$ !(e.value = key))'. The semantics of `else` branch is preserved after the transformation to an `assume` statement.

With encoding of data-flow and control-flow, the conjunction of all generated formulas is the formula for the code segment under analysis. A model to this code formula corresponds to a valid path of the code fragment.

## 4 Algorithm

The goal of our splitting algorithm is to divide the complexity of checking the program while preserving its semantics (w.r.t. to the given scope). This paper presents a *vertex-based* splitting algorithm. Splitting a program into two sub-programs partitions paths in the program based on a chosen vertex: one sub-program has all paths that go through the vertex and the other sub-program has all paths that bypass that vertex. Our vertex-based path splitting guarantees the consistency between the original program and sub-programs. For a heuristic measure of the complexity of checking, we propose to use the number of branches. Our strategy is selecting a vertex so that the number of branches in each of sub-programs is minimized.

Our approach checks a given program *p* as follows.

1. Translate *p* into *p'* where *p'* represents the *computation graph* [15] of *p*, i.e., the loops in *p* are unrolled and method calls in-lined to generate *p*;
2. Represent *p'* as a graph $CG = (V, E)$ where $V$ is a set of vertices such that each statement in *p'* has a corresponding vertex in $V$, and $E$ is a set of edges such that each control-flow edge in *p'* has a corresponding edge in $E$. For each edge $e = (u, v)$, $u = e.from$, and $v = e.to$;
3. Apply the splitting heuristic to determine a likely optimal splitting vertex $v$;

4. Split *CG* into two sub-graphs $CG_1$ and $CG_2$;
5. Recursively split $CG_1$ and $CG_2$ if needed;
6. Check the set of sub-programs corresponding to the final set of sub-graphs.

Recall a computation graph has one *Entry* vertex and one *Exit* vertex for the program. *Entry* has no predecessor and *Exit* has no successor. A vertex *v* representing a branch-statement has two successors: vertex *v.then* and vertex *v.else*. Vertices that do not represent branch-statements have only one successor, *v.next*. The computation graph of a program is a *DAG* (Directed Acyclic Graph). An *execution path* in the computation graph is a sequence of vertices from *Entry* to *Exit* through edges in *E*.

**Definition.** Given a CG = (V, E) and a set of edges S $\subset$ E,

*then-branch-predecessor*(*S*) =

{ *u* | *u*$\in$ V, *u* is a branch-statement, at least one edge in *S* is reachable from *u.then*, but no edge in *S* is reachable from *u.else*}

*else-branch-predecessor*(*S*) =

{ *u* | *u*$\in$ V, *u* is a branch, at least one edge in *S* is reachable from *u.else*, but no edge in *S* is reachable from *u.then*}

*Sub-CG(S)* = ($V'$, $E'$)

$V' = V$, and

$E' = E - \{e \mid e \in E,\ e = u \rightarrow u.else$ if $u$ is in *then-branch-predecessor*(*S*), or $e = u \rightarrow u.then$ if $u$ is in *else-branch-predecessor*(*S*)$\}$ .

**Theorem 1.** Given a computation graph *CG*=(*V*, *E*) and a set of edges $S \subset E$, an execution path *p* visits at least one edge in *S* if and only if *p* is a path in *Sub-CG*(*S*) = ($V'$, $E'$).

**Proof.** *Sub-CG*(*S*) is a sub-graph of *CG* with fewer edges. An execution path in *CG* is still in *Sub-CG*(*S*) if and only if this path does not contain any edge in $E - E'$.

$\Rightarrow$ Assume that there is a path *p* in CG that visits an edge in S but is not in the sub-graph. Suppose *p* visits an edge $v_i \rightarrow v_j$ that has been removed. According to the definition of the sub-graph, none of the edge in *E* is reachable from $v_i \rightarrow v_j$, i.e., a contradiction.

$\Leftarrow$ Assume *p* is a path in *CG* that does not visit any edge in *S*. Let $P = \{q \mid q$ is a path of $CG$, $q$ visits $S\}$. Since *S* is not empty, *P* is not empty. For each path *q* in *P*, match *p* and *q* according to their vertex sequence. Let $v_i$ be the last vertex in *p* that matches a vertex in P. $v_i$ must be a branch vertex. Let edge $v_i \rightarrow v_j$ and edge $v_i \rightarrow v_j'$ be the two branches from $v_i$. Suppose edge $v_i \rightarrow v_j$ is in *p*. $v_i \rightarrow v_j$ cannot reach any edge in *S*. Since $v_i$ is the last vertex-match with paths in *P*, $v_i \rightarrow v_j'$ can reach an edge in *S*. So $v_i \rightarrow v_j$ should be removed according to definition of *then-branch-predecessor* (*S*) or *else-branch-predecessor* (*S*). So any path that does not visit an edge in *S* will be removed from sub-graph. ■

Since the computation graph of the program is a DAG after loop unrolling, we can linearly order the vertices using a topological sort.

Given a computation graph *CG*, let *order* represent *topological-sort*(*CG*) such that *order*[*Exit*] =0; *order*[*Entry*] = *n*-1; and *n* is number of vertices in *CG*.

**Definition.** Let $u$ be a vertex in *CG*. Define the set *go-through-edge*($u$) = $\{e \mid e \in E, e.to = u\}$.

**Theorem 2.** Given a vertex $u$ in *CG*, a path visits $u$ if and only if the path visits an edge in *go-through-edge*($u$) .

**Proof.** Since *CG* is a directed graph, all paths visiting vertex $u$ will go through an edge whose end point is $u$. ■

**Definition.** Given a vertex $u$ in *CG*, *bypass-edge*($u$) = $\{e \mid e \in E$, order[$e.from$] > order[$u$] and order[$e.to$] < order[$u$]$\}$.

**Theorem 3.** Given a vertex $u$, a path $p$ bypasses $u$ if and only if $p$ visits an edge in *bypass-edge*($u$).

**Proof.** $\Rightarrow$ Let path $p$: $v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_m$ be a path in *CG*, $v_0$ = *Entry* and $v_m$ = *Exit*. According to the topological sort, order[$v_0$] > order[$v_1$] > order[$v_2$] > … > order[$v_m$]. For vertex u, if $u \neq$ *Entry* and $u \neq$ *Exit*, then order[$u$] < order[*Entry*] and order[$u$] >order[*Exit*]. Since $u$ is not in $p$, there must be two vertices $v_i$ and $v_j$ in $p$ such that order[$v_i$] > order[$u$] and order[$v_j$] <order[$u$]. By definition, edge $v_i \rightarrow v_j$ is in *bypass-edge*($u$) .

$\Leftarrow$ Let path $p$: $v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_m$ be a path in *CG*, $v_0$ = *Entry* and $v_m$ = *Exit*. Since $p$ visits a path in *bypass-edge*($u$) there exist an edge $v_i \rightarrow v_j$ such that order[$v_i$] > order[$u$] and order[$v_j$] <order[$u$]. According to the topological sort, order[$v_k$] > order[$v_i$] > order[$u$] if k < i, and order[$v_k$] < order[$v_j$] < order[$u$] if k>j. So path $p$ can not visit $u$. ■

**Theorem 4.** Given a vertex $u$ in *CG*, a path visits $u$ if and only if the path is in *Sub-CG (go-through-edge(u))*.

**Proof.** Follows from Theorem 2 and Theorem 1. Since paths that visit u are the paths that visit *go-through-edge(u)* and paths that visit go-through-edge($u$) are the paths in *Sub-CG(go-through-edge(u))*, therefore paths that visit $u$ are the paths in *Sub-CG(go-through-edge(u)*. ■

**Theorem 5.** Given a vertex $u$ in *CG*, a path bypasses $u$ if and only if the path is in *Sub-CG(bypass-edge(u))*.

**Proof.** Follows from Theorem 3 and Theorem 1. Since paths that visit $u$ are the paths that visit *bypass-edge(u)* and paths that visit *bypass-edge*($u$) are the paths in *Sub-CG(bypass-edge(u))*, therefore paths that visit $u$ are the paths in *Sub-CG (bypass-edge(u))*. ■

**Algorithm**

Figure 3 shows our splitting algorithm. The method *branches*() returns the branch vertices of a given computation graph. If one edge of a branch vertex has been removed in a sub-graph, this branch vertex will not be counted as a branch in that sub-graph.

```
List<CG> Split-CG(CG cg)
{
    List<CG> sub = new List<CG>();
    for (Vertex u : cg.vertices){
        branch1 = branches(Sub-CG(go-through-edge(u))).size();
        branch2 = branches(Sub-CG(bypass-edge(u))).size();
        split-complexity = max(branch1, branch2);
        if (split-complexity < current-complexity){
            v=u;
            current-complexity = split-complexity;
        }
    }
    sub.add(Sub-CG(go-through-edge(v)));
    sub.add(Sub-CG(bypass-edge(v)));
    return sub;
}
```

**Fig. 3.** Program splitting algorithm

To illustrate, consider the example form Section 2. The split-complexity of $v_{11}$ can be calculated in the following steps:

1) For go-through sub-graph,
   *go-through-edge*($v_{11}$) = { $v_2 \rightarrow v_{11}$, $v_6 \rightarrow v_{11}$, $v_{10} \rightarrow v_{11}$};
   *branches*(*Sub-CG*(*go-through-edge*($v_{11}$)))= {$v_2$, $v_6$};
2) For bypass sub-graph,
   *bypass-edge*($v_{11}$) = { $v_3 \rightarrow v_4$, $v_7 \rightarrow v_8$)};
   *branches*(*Sub-CG*(*bypass-edge*($v_{11}$)))= {$v_3$};
3) For split-complexity,
   *Split-complexity*($v_{11}$) = max {|{ $v_2$,$v_6$}|, |{$v_3$}|} = 2.

According to the definition of *Sub-CG*, some branch edges will be removed to construct a sub-graph. Given a branch, edge removing is implemented by transforming the branch statement into an **`assume`** statement. The semantic consistency of this transformation is discussed in the background section.

## 5 Experiments

To evaluate our approach, we compare performance of our sub-program-based *incremental* analysis and the traditional entire program analysis. We select the Forge tool-set [6] as the baseline, since it is the most recent implementation of the traditional approach from the Alloy group at MIT. We piggyback on Forge to implement our incremental approach.

Experimental evaluation is based on checking four procedures in Java library classes: `contains`() of `LinkedList` (a singly-linked acyclic list), `contains`() of `BinarySearchTree`, `add`() of `BinarySearchTree`, and `topologicalsort`() of `Graph` (directed acyclic graph).

In relational logic based bounded verification, the bound specifies the numbers of loop unrolling, scope, and bit-width—the number of bits used to represent an integer value. While translating integer into propositional logic, we set the bit-width to 4 in all the four experiments. *Scope* defines the maximum number of nodes in a list, tree, or graph. *Unrolling* specifies the number of unrollings for a loop body. For `contains`() of `LinkedList` and `contains`() of `BinarySearchTree`, we check them with fixed scope and varied unrolling. For `add()` of `BinarySearchTree`, and `topologicalsort()` of `Graph`, we check them with fixed unrolling and varied scope.

For each bound, we run our *incremental* analysis and the traditional entire program analysis to check a procedure against its *pre-condition* and *post-condition*, which represent the usual correctness properties including structural invariants. In incremental analysis, we did two round splitting and generated four sub-programs. While checking each sub-program, we record the checking time, number of branches, variables and clauses of CNF formula. The *total* time is the sum of the checking time of all sub-programs. For the traditional analysis, we similarly record the checking time, number of branches, variables and clauses of CNF formula. The speedup is the ratio of the checking time of the traditional analysis to the total checking time of our incremental analysis.

We ran experiments on a Dual-Core 1.8GHz AMD Opteron processor with 2 GB RAM. We selected MiniSAT as the SAT solver. We run each experiment three times and use the average as the final result. The results are showed in the tables that follow.

Table 1 and Table 2 show the performance comparison for different loop unrollings. Table 3 and Table 4 show the comparison for different scopes. Results from the four experiments showed that our splitting algorithm gave at least a 2.71X performance improvement over the traditional approach, whereas the maximum speed-up was 36.73X.

The results also show that our splitting algorithm scales better. As unrolling increasing from 5 to 8, speedup of checking `contains()` of `LinkedList` increases from 3.99X to 36.73X. As scope increasing from 4 to 7, speedup of checking `add()` of `BinarySearchTree` increases from 4.78X to 12.6X.

**Table 1.** `LinkedList.contains()` (bit-width = 4, scope = 8)

| unrolling | | sub0 | sub1 | sub2 | sub3 | total | entire | speedup |
|---|---|---|---|---|---|---|---|---|
| 5 | time (sec.) | 2 | 1 | 82 | 1 | 86 | 343 | 3.99X |
| | # branch | 1 | 2 | 2 | 2 | | 10 | |
| | # variable | 4655 | 4149 | 4731 | 3969 | | 4740 | |
| | # clauses | 10081 | 8353 | 10167 | 7563 | | 14271 | |
| 6 | time(sec.) | 8 | 1 | 173 | 7 | 189 | 653 | 3.46X |
| | # branch | 2 | 2 | 2 | 3 | | 12 | |
| | # variable | 4911 | 4149 | 4985 | 4226 | | 4996 | |
| | # clauses | 10945 | 8353 | 11031 | 8436 | | 15213 | |
| 7 | time(sec.) | 66 | 1 | 428 | 3 | 498 | 4541 | 9.12X |
| | # branch | 2 | 3 | 3 | 3 | | 14 | |
| | # variable | 5165 | 4406 | 5242 | 4226 | | 5252 | |
| | # clauses | 11809 | 9218 | 11904 | 8436 | | 16155 | |
| 8 | time(sec.) | 179 | 1 | 359 | 44 | 583 | 21414 | 36.73X |
| | # branch | 3 | 3 | 3 | 4 | | 16 | |
| | # variable | 5422 | 4406 | 5496 | 4484 | | 5508 | |
| | # clauses | 12674 | 9218 | 12768 | 9310 | | 17097 | |

**Table 2.** `BinarySearchTree.contains()` (bit-width = 4, scope = 7)

| unrolling | | sub0 | sub1 | sub2 | sub3 | total | entire | speedup |
|---|---|---|---|---|---|---|---|---|
| 4 | time(sec.) | 564 | 552 | 390 | 388 | 1894 | 6468 | 3.42X |
| | # branch | 4 | 4 | 3 | 4 | | 12 | |
| | # variable | 7776 | 7369 | 6961 | 6724 | | 7808 | |
| | # clauses | 20734 | 19185 | 17635 | 16726 | | 21193 | |
| 5 | time(sec.) | 1 | 2427 | 1745 | 301 | 4474 | 15015 | 3.36X |
| | # branch | 7 | 7 | 5 | 4 | | 15 | |
| | # variable | 8151 | 8151 | 7376 | 6724 | | 8224 | |
| | # clauses | 22170 | 22178 | 19300 | 16726 | | 22859 | |
| 6 | time(sec.) | 698 | 1879 | 546 | 936 | 4059 | 18982 | 4.68X |
| | # branch | 7 | 5 | 5 | 6 | | 18 | |
| | # variable | 8599 | 8192 | 7539 | 6976 | | 8640 | |
| | # clauses | 23941 | 22400 | 19822 | 17861 | | 24525 | |
| 7 | time(sec.) | 1 | 2535 | 2834 | 686 | 6056 | 28435 | 4.71X |
| | # branch | 11 | 11 | 6 | 7 | | 21 | |
| | # variable | 8975 | 8975 | 7784 | 7140 | | 9056 | |
| | # clauses | 25386 | 25394 | 20850 | 18392 | | 26191 | |
| 8 | time(sec.) | 794 | 1085 | 1289 | 623 | 3791 | 18703 | 4.94X |
| | # branch | 13 | 13 | 7 | 7 | | 24 | |
| | # variable | 9384 | 9384 | 7948 | 7140 | | 9472 | |
| | # clauses | 26945 | 26953 | 21381 | 18392 | | 27857 | |

The relative lower speedup in `contains()` of `BinarySearchTree` and `topologicalsort()` of `Graph` show a limitation of our approach. Compared with traditional approach which checks correctness against specifications only once, our *divide-and-solve* approach requires multiple correctness checking, one checking for one sub-program. In case the complexity of specification formula is much heavier than code formula, the benefit from dividing the code formula will be reduced largely by the overhead from multiple checking specification formulas. However, even with specification of complex data structure invariants, our approach still shows 5X speedup in `contains()` of `BinarySearchTree` with 8 unrollings and 7 nodes.

The results show that our splitting heuristic is effective at evenly splitting the branches. Moreover, the smaller number of variables and clauses for the incremental approach shows the workload to SAT has been effectively divided by splitting entire program into sub-programs using our approach.

**Table 3.** `BinarySearchTree.add()` (unrolling = 3, bit-width = 4)

| scope | | sub0 | sub1 | sub2 | sub3 | total | entire | speedup |
|---|---|---|---|---|---|---|---|---|
| 4 | time(sec.) | 2 | 3 | 1 | 3 | 9 | 43 | 4.78X |
| | # branch | 5 | 6 | 6 | 7 | | 11 | |
| | # variable | 4878 | 5092 | 4692 | 5083 | | 9686 | |
| | # clauses | 16132 | 17393 | 15079 | 17397 | | 36929 | |
| 5 | time(sec.) | 13 | 15 | 3 | 9 | 40 | 249 | 6.23X |
| | # branch | 5 | 6 | 6 | 7 | | 11 | |
| | # variable | 6705 | 7038 | 6446 | 6653 | | 12837 | |
| | # clauses | 22457 | 24308 | 20990 | 22973 | | 49623 | |
| 6 | time(sec.) | 140 | 316 | 30 | 19 | 505 | 4339 | 8.59X |
| | # branch | 5 | 6 | 6 | 7 | | 11 | |
| | # variable | 8689 | 9161 | 8349 | 8340 | | 16335 | |
| | # clauses | 29414 | 31943 | 27487 | 28965 | | 63809 | |
| 7 | time(sec.) | 1675 | 6409 | 863 | 76 | 9023 | 109730 | 12.16X |
| | # branch | 5 | 6 | 6 | 7 | | 11 | |
| | # variable | 11030 | 11661 | 10601 | 10247 | | 20380 | |
| | # clauses | 37703 | 40998 | 35270 | 35738 | | 80187 | |

**Table 4.** `Graph.TopologicalSort` () (unrolling = 7, bit-width = 4)

| scope | | sub0 | sub1 | sub2 | sub3 | total | entire | speedup |
|---|---|---|---|---|---|---|---|---|
| 7 | time(sec.) | 183 | 152 | 118 | 1 | 454 | 1436 | 3.16X |
| | # branch | 1 | 1 | 1 | 1 | | 7 | |
| | # variable | 269908 | 197682 | 125456 | 53230 | | 269962 | |
| | # clauses | 1073479 | 785037 | 496595 | 208153 | | 1084273 | |
| 8 | time(sec.) | 210 | 199 | 114 | 1 | 524 | 1422 | 2.71X |
| | # branch | 1 | 1 | 1 | 1 | | 7 | |
| | # variable | 299104 | 219070 | 139036 | 59002 | | 299158 | |
| | # clauses | 1197974 | 875832 | 553690 | 231548 | | 1210304 | |
| 9 | time(sec.) | 214 | 278 | 157 | 1 | 650 | 2113 | 3.25X |
| | # branch | 1 | 1 | 1 | 1 | | 7 | |

**Table 4.** (*continued*)

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | # variable | 357978 | 262236 | 166494 | 70752 | | 358032 | |
| | # clauses | 1457783 | 1065867 | 673951 | 282035 | | 1471649 | |
| 10 | time(sec.) | 357 | 255 | 187 | 2 | 801 | 2844 | 3.55X |
| | # branch | 1 | 1 | 1 | 1 | | 7 | |
| | # variable | 402696 | 295010 | 187324 | 79638 | | 402750 | |
| | # clauses | 1659106 | 1212870 | 766634 | 320398 | | 1674508 | |
| 11 | time(sec.) | 611 | 341 | 263 | 2 | 1217 | 3694 | 3.04X |
| | # branch | 1 | 1 | 1 | 1 | | 7 | |
| | # variable | 439783 | 322189 | 204595 | 87001 | | 439837 | |
| | # clauses | 1829579 | 1337181 | 844783 | 352385 | | 1846517 | |
| 12 | time(sec.) | 558 | 519 | 247 | 2 | 1326 | 4372 | 3.31X |
| | # branch | 1 | 1 | 1 | 1 | | 7 | |
| | # variable | 476935 | 349417 | 221899 | 94381 | | 476989 | |
| | # clauses | 2003834 | 1464198 | 924562 | 384926 | | 2022308 | |

## 6 Related Work

Our work is based on previous research [15] that models a heap-manipulating procedure using Alloy and finds counterexamples using SAT. Jackson et al. [15] proposed an approach to model complex data structures with relations and encode control flow, data flow, and frame conditions into relational formulas. Vaziri et al. [29] optimized the translation to boolean formulas by using a special encoding of functional relations. Dennis et al. [5] provided explicit facilities to specify imperative code with first-order relational logic and used an optimized relational model finder [28] as the backend constraint solver. Our algorithm can reduce the workload to the backend constraint solver by splitting the computation graph that underlies all these prior approaches and dividing the procedure into smaller sub-programs.

DynAlloy [8] is a promising approach that builds on Alloy to directly support sequencing of operations. We believe our incremental approach can optimize DynAlloy's solving too.

Bounded exhaustive checking, e.g., using TestEra [16] or Korat [1] can check programs that manipulate complex data structures. Testing, however, has a basic limitation that running a program against one input only checks the behavior for that input. In contrast, translating a code segment to a formula that is solved allows checking all (bounded) paths in that segment against all (bounded) inputs.

The recent advances in constraint solving technology have led to a rebirth of symbolic execution [17, 18]. Guiding symbolic execution using concrete executions is rapidly gaining popularity as a means of scaling it up in several recent frameworks, most notably DART [10], CUTE [26], and EXE [2]. While DART and EXE focus on properties of primitives and arrays to check for security holes (e.g., buffer overflows), CUTE has explored the use of white-box testing using preconditions, similar to Korat [1]. Symbolic/concrete execution can be viewed as an extreme case of our approach where each sub-program represents exactly one path in the original program. As the number of paths increases, the number of calls to the constraint solver increases in symbolic execution. Our approach is motivated by our quest to find a sweet spot between checking all paths at once (traditional approach) and each path one-by-one (symbolic/concrete execution).

Model checkers have traditionally focused on properties of control [12, 22]. Recent advances in software model checking [9, 30] have allowed checking properties of data. However, software model checkers typically require explicit checking of each execution path of the program under test.

Slicing techniques [27] have been used to reduce workload of bounded verification. Dolby et al. [6] and Saturn [31] perform slicing at the logic representation level. Millett et al. [23] slice Promela programs for SPIN model checker [12]. Visser et al. [30] and Corbett et al. [2] prune the parts that are not related to temporal constraints and slice at the source code level. Since slicing is based on constraints, the effectiveness depends on the properties to be checked. Statements that do not manipulate any relations in properties will not be translated into formula for checking. If constraints are so complex that all the relations show up, no statements will be pruned. Our program-splitting algorithm can still reduce workload to backend constraint solvers because our path partitioning algorithm is independent of constraints to be checked.

Sound static analyses, such as traditional shape analysis [25, 19] and recent variants [20], provide correctness guarantees for all inputs and all execution paths irrespective of a bound. However, they typically require additional user input in the form of additional predicates or loop invariants, which are not required for scope-bounded checking, which provides an under-approximation of the program under test.

## 7 Conclusions

Scalability is a key issue in *scope-bounded* checking. Traditional approaches translate the bounded code segment of the *entire* program into *one* input formula for the underlying solver, which solves the complete formula in one execution. For non-trivial programs, the formulas are complex and represent a heavy workload that can choke the solvers.

We propose a *divide-and-solve* approach, where smaller segments of bounded code are translated and analyzed. Given a vertex in the control-flow graph, we split the computation graph of the program into two sub-graphs: go-through sub-graph and bypass sub-graph. The go-through sub-graph has all the paths that go through the vertex and the bypass sub-graph has all the paths that bypass the vertex. Our vertex-based path partitioning can guarantee the semantic consistency between the original program and the sub-programs. We propose to use the number of branch statements as a heuristic to compute an analysis complexity metric of a program. To effectively divide the analysis complexity of a program, the heuristic selects a vertex so that the number of branch statements in each of sub-programs is minimized.

We evaluated our *divide-and-solve* approach by comparison with the traditional approach by checking four Java methods against pre-conditions and post-conditions defined in Alloy. The experimental results show that our approach provides significant speed-ups over the traditional approach.

The results also show other potential benefits of our program splitting algorithm. Because all sub-graphs are independent, they can be checked in parallel. Since our program splitting algorithm can effectively divide the workload, parallel checking the sub-programs would likely introduce significant speedups. Incremental compilation and solving are likely to provide further optimizations.

In ongoing work, we are exploring novel strategies for dividing the workload. We aim to leverage concepts from traditional dynamic and static analysis. For example, notions of code coverage in software testing lend to division strategies.

## Acknowledgments

We thank Greg Dennis for his help on the Forge framework. This work was supported in part by NSF grants IIS-0438967, CCF-0702680, and CCF-0845628, and AFOSR grant FA9550-09-1-0351.

## References

1. Boyapati, C., Khurshid, S., Marinov, D.: Korat: Automated Testing Based on Java Predicates. In: Proc. of ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA (2002)
2. Cadar, C., Ganesh, V., Pawlowski, P., Dill, D., Engler, D.: EXE: Automatically Generating Inputs of Death. In: Proc. of the 13th ACM Conference on Computer and Communications Security (CCS) (2006)
3. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Pasareanu, C.S., Robby, Zheng, H.: Bandera: extracting finite-state models from Java source code. In: Proc. of International Conference on Software Engineering, ICSE (2000)
4. Darga, P., Boyapati, C.: Efficient software model checking of data structure properties. In: Proc. of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA (2006)
5. Dennis, G., Chang, F.S.H., Jackson, D.: Modular verification of code with SAT. In: Proc. of the International Symposium on Software Testing and Analysis, ISSTA (2006)
6. Dolby, J., Vaziri, M., Tip, F.: Finding Bugs Efficiently with a SAT Solver. In: Proc. of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE (2007)
7. Eén, N., Sörensson, N.: An extensible SAT solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
8. Frias, M.F., Galeotti, J.P., López Pombo, C.G., Aguirre, N.M.: DynAlloy: upgrading alloy with actions. In: Proc. of International Conference on Software Engineering, ICSE (2005)
9. Godefroid, P.: Model Checking for Programming Languages using VeriSoft. In: Proc.of ACM Symposium on Principles of Programming Languages, POPL (1997)
10. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. In: Proc. of ACM SIGPLAN conference on Programming language design and implementation, PLDI (2005)

11. Heitmeyer, C., James Kirby, J., Labaw, B., Archer, M., Bharadwaj, R.: Using abstraction and model checking to detect safety violations in requirements specifications. IEEE Transactions on Software Engineering 24(11), 927–948 (1998)
12. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley, Reading (2004)
13. Jackson, D.: Automating first-order relational logic. In: Proc. of the International Symposium on Foundations of Software Engineering, FSE (2000)
14. Jackson, D.: Software Abstractions: logic, language, and analysis. MIT Press, Cambridge (2006)
15. Jackson, D., Vaziri, M.: Finding bugs with a constraint solver. In: Proc. of the International Symposium on Software Testing and Analysis, ISSTA (2000)
16. Khurshid, S., Marinov, D.: TestEra: Specification-based Testing of Java Programs Using SAT. Automated Software Engineering Journal 11(4) (October 2004)
17. Khurshid, S., Pasareanu, C., Visser, W.: Generalized Symbolic Execution for Model Checking and Testing. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 553–568. Springer, Heidelberg (2003)
18. King, J.C.: Symbolic execution and program testing. Communications of the ACM 19(7) (July 1976)
19. Klarlund, N., Møller, A., Schwartzbach, M.I.: MONA implementation secrets. In: Yu, S., Păun, A. (eds.) CIAA 2000. LNCS, vol. 2088, p. 182. Springer, Heidelberg (2001)
20. Kuncak, V.: Modular Data Structure Verification. Ph.D. thesis, EECS Department, Massachusetts Institute of Technology (2007)
21. Leonardo, M., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
22. McMillan, K.: Symbolic Model Checking. Kluwer Academic Publishers, Dordrecht (1993)
23. Millett, L.I., Teitelbaum, T.: Slicing Promela and its applications to model checking. In: Proc. of the 4th International SPIN Workshop (1998)
24. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: Proc. of 39th Design Automation Conference, DAC (2001)
25. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Transactions on Programming Languages and Systems (TOPLAS) 24(3), 217–298 (2002)
26. Sen, K., Marinov, D., Agha, G.: CUTE: A concolic unit testing engine for C. In: Proc. of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, ESEC/FSE (2005)
27. Tip, F.: A survey of program slicing techniques. Journal of Programming Languages 3(3), 121–189 (1995)
28. Torlak, E., Jackson, D.: Kodkod: A Relational Model Finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007)
29. Vaziri, M., Jackson, D.: Checking properties of heap-manipulating procedures with a constraint solver. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 505–520. Springer, Heidelberg (2003)
30. Visser, W., Havelund, K., Brat, G.P., Park, S., Lerda, F.: Model checking programs. In: Proc. of International Conference on Automated Software Engineering, ASE (2000)
31. Xie, Y., Aiken, A.: Saturn: A scalable framework for error detection using boolean satisfiability. ACM Transactions on Programming Languages and Systems (TOPLAS) 29(3) (2007)